AN ADAPTATION OF THE HERSHEY DIGITIZED CHARACTER SET FOR USE IN COMPUTER GRAPHICS AND TYPESETTING

Patrick Michael Doyle

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

An Adaptation of the Hershey Digitized
Character Set For Use In
Computer Graphics and Typesetting

by

Patrick Michael Doyle

June 1977

Thesis Advisor: G. L. Barksdale, Jr.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) An Adaptation of the Hershey Digitized Character Set For Use In Computer Graphics and Typesetting	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis: June 1977
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Patrick Michael Doyle	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE June 1977
	13. NUMBER OF PAGES 170
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer typesetting, digitized fonts, text processing, typeface, fonts, vectors, Hershey

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Font definitions of 1377 characters of various styles developed by Allen V. Hershey were used as an initial data base. His character definitions were first put into a form suitable for use by vector graphics display processors, and then these vectors were converted into dot matrix form in a variety of point sizes. This conversion and digitization process was done using the C programming language; the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

host computer was a PDP-11/50 with the UNIX operating system, and the computerized typesetting was done on a VERSATEC 1200-A printer/plotter.

As a result, a large data base for use in computerized typesetting has been developed. In addition, the computerized typesetting system at the Naval Postgraduate School has been improved and adapted to make use of the large number of fonts now available.

Approved for public release; distribution unlimited.

An Adaptation of the Hershey Digitized Character Set For Use In Computer Graphics and Typesetting

by

Patrick Michael Doyle
Lieutenant, United States Navy
B.S., United States Naval Academy, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
JUNE 1977

ABSTRACT

Font definitions of 1377 characters of various styles developed by Allen V. Hershey were used as an initial data base. His character definitions were first put into a form suitable for use by vector graphics display processors, and then these vectors were converted into dot matrix form in a variety of point sizes. This conversion and digitization process was done using the C programming language; the host computer was a PDP-11/50 with the UNIX operating system, and the computerized typesetting was done on a VERSATEC 1200-A printer/plotter.

As a result, a large data base for use in computerized typesetting has been developed. In addition, the computerized typesetting system at the Naval Postgraduate School has been improved and adapted to make use of the large number of fonts now available.

CONTENTS

I. INTRODUCTION

A. BEGINNINGS

Early in the 1960's, as computer technology began to develop more and more rapidly, the influence of computers expanded into many new areas. As computers became more sophisticated, more available, and easier to use, many different groups began to search for computer applications within their fields. One field in which several uses were found for computers was in the publishing industry.

Early systems used by newspaper and book publishers involved various methods for character generation and mechanical positioning of those characters; while these systems were faster than typesetting by hand, they still left room for considerable improvement. Current systems electronically generate and position their characters, greatly improving the speed of the process.

In his recent book on the subject of electronic composition, N. Edward Berg states that:

> Although the effort to set type by computer has been underway since the early 1960's, it has not yet reached the age of maturity.... Many exciting new developments have already taken place which are but a prelude to what will unfold in the future. Proper computerization of typesetting now offers very significant cost advantages over hot metal....

A technology (that of computers) and a technological art (that of typesetting) are being blended together in a way that is particularly challenging to the computer technology since the typesetting art must be maintained. The computer must assist the art--not dictate or attempt to eliminate it.

When two disciplines come together there is always a need for good communications and standardizations -- standardization not in terms of the art or expression, but in terms of electronic techniques. The development of these standards allows an orderly application of computer technology and will not detract even minutely from the needs of the art and free expression.

The computer technology will not replace creative human expression but will enable that expression to have enlarged horizons, and leave the mundane and repetitious to the computer. [Ref. 3, p. vii]

B. EVOLUTION

In addition, Mr. Berg also mentions a "generation" classification which was developed to create a rational subdivision of machines into classes as follows:

1. First Generation. Machines evolved from their hot metal ancestors but adapted to the photographic process.

2. Second Generation. Machines not evolved from previous concepts embodied in hot metal machines but based on the new technology of setting type from photographic masters.

3. Third Generation. Machines designed to work in con-

junction with computers at high speed (greater than 100 characters per second) and expose the character image via a cathode ray tube (CRT).

Since the early experiments with computerized typesetting, the computer has played a more and more important part in the process. The main direction of this paper has been to provide these "third generation" machines with a large collection of type styles in a variety of sizes. One of the largest data bases available was that digitized by Allen V. Hershey in 1967. However, this data was available only in vector form and current graphics display processors and typesetters usually require information for their character displays to be in dot matrix rather than in vector form.

The first step in the conversion process involved obtaining the raw data base, and then converting into a form that was usable for generating the appropriate vectors. This process is described in Appendix C. An interesting by-product of these initial efforts was the program written for use on the TEKTRONIX 4014 display processor and described in Appendix D. This program allowed the user to select a particular font and then to draw a character from that font on the CRT; the appearance of the characters allowed the verification of the vector data base, and provided a check on procedures used to that point.

After the data base was confirmed, the next step was the conversion of the vector data into bit patterns that would

allow the use of these fonts in a dot matrix environment on raster scan CRTs. The goal was to produce a program that could convert a standard size vector definition of a character into a dot matrix definition in the size desired by the user. Anyone using this program gained access to the Hershey data base and increased the character set available for his use by a significant amount.

The next sections provide the background on some early experiences with computerized typesetting and on some current methods used by "third generation" machines.

C. FONT FUNDAMENTALS

A font is a collection of different characters, all of the same style and height, which are mapped onto a character set. On the PDP-11/50, the 7-bit ASCII set of 128 character codes is used. Some fonts have generic names, such as the Bodoni fonts; others have lost their origins but are named for their appearance, like the Gothic English fonts. Some fonts are recent creations, and have received more mundane names; SAIL10, for example, is a 10 point font created at the Stanford Artificial Intelligence Laboratory (SAIL). The most useful fonts are those that contain both upper and lower case English letters, Arabic numerals, and a minimal set of punctuation marks. The more exotic fonts contain mathematical symbols, characters from foreign languages, and, occasionally, homemade symbols for very special purposes.

Some characteristics of fonts which should be mentioned before proceeding are:

1. Character width

A font is either fixed or variable width. When a font is fixed width, each character, whether it is a 'M' or an 'i', will have the same widths. In a variable width font, on the other hand, each character may have a unique width.

2. Typeface

Fonts are generally classified by the style of the typeface used. Bodoni, Nonie, Complex, Triplex, and so on, are typical examples of styles.

3. Size

Together with typeface, size makes up one of the most noticeable characteristics of a font, and provides one of the most useful methods of classification. Font size is most often referred to in "point" size, a measure of the font's height. A point is a traditional printer's measure, and is approximately 1/72 inch. On the VERSATEC, the unit of measure used is the pixel, the smallest unit of resolution possible on the machine. The picture element (pixel or pel) is 1/200 inch, about four times the resolution of most CRTs. At 200 pixels per inch, point size and raster height may be converted using the following formula :

$$\text{raster height} = (\text{point size} * 2.8) + 1$$

One character width of pixels represents one raster line holding the "1s", which are dots which must be black, and the "0s", which are blank spots; together these binary digits make up a horizontal slice of a character picture. The character's height is determined by the point size that is required, and the widths are proportional to the heights. Appendix F contains a more complete description of font and character dimensions.

4. Style

Fonts that use the same typeface may appear different because they have been altered slightly; a standard font may be regular, it may be slanted to the right (italicized), or it may be thickened (bold face).

D. EARLY COMPUTERIZED TYPESETTING

1. Phototypesetters

a. Background

Early in 1961, Michael P. Barnett, the Director of the Cooperative Computing Laboratory (C.C.L.) at the Massachusetts Institute of Technology, encountered a tape-operated phototypesetting machine and became interested in the possibility of producing operating tapes for these machines from the output of a digital computer. Early programming efforts produced some interesting results, but none that were especially useful. Following the award of a

research grant in 1962, however, the staff at the C.C.L. was enlarged and a system of computer programs was completed. These programs were used in 1963 and 1964 to set many hundreds of pages of material for a variety of reports, papers, pamphlets, and other publications of interest to Mr. Barnett and his staff.

b. Equipment

The equipment used at the C.C.L. included an IBM 709/90 computer with 32K of memory which produced output tapes for a PHOTON 560 phototypesetter. Text material was prepared for the computer using a FRIDEN FLEXOWRITER.

The FLEXOWRITER had a conventional keyboard and produced copy that had the appearance of standard typewritten material; the type was in a single typestyle and size, and lines were not justified. A paper tape punch unit was a part of the FLEXOWRITER, and striking any key on the keyboard, whether it was a printing key or not, caused a pattern of holes to be punched in the tape and then the tape was automatically advanced. The paper tape was then run through the digital computer to translate the 8-bit FLEXOWRITER codes into bit patterns on magnetic tapes that could be used to control the phototypesetter. These input tapes were usually internally coded to select type fonts, type size, and so on, in much the same way that input to current text processing systems or text formatters such as TPS [Ref. 1] and NROFF [Ref. 7] is done.

The PHOTON machine operated using a glass character disk, a small electronic flash unit, a lens turret and prism, and a disk level selection cam. Each glass disk contained photographic negatives of 1440 characters arranged in eight concentric rings of 180 uniformly spaced characters; the disk rotated in a vertical plane in front of the flash unit, and the spindle in which the disk was mounted rested in a cradle which could occupy eight parallel positions. Changing the position of the cradle with the cam moved the disk a small amount in the vertical plane; as the ring of characters moved past the flash unit, the unit flashed at the appropriate character and the image of that character was focused onto the film passing beneath the typesetter. Different disks were used to provide the different type fonts required, and type size was changed by rotating the lens turret to change the size of the lens. The film was then cut into pages and printed using standard offset printing techniques.

c. Lessons Learned at M.I.T.

As personnel at the C.C.L. gained experience in computerized typesetting, the advantages of that system became obvious. First of all, tape-operated typesetting machines could set computerized output more rapidly than human operators could, and it could be done without the intervention of keyboard operators and the inevitable human errors that occur. Computers could also sort, update, and perform other clerical operations on almost any form of in-

put. Computers were also used to simplify the keyboard work involved in setting type from a manuscript by introducing typographical details and styles that did not force the keyboard operators to attend to the smallest details as they had had to do when using conventional techniques.

2. Hot Metal Machines

a. Background

In the middle 1960's, the interaction of computers and various typesetting devices was gaining more and more attention in the publishing industry. Computers were being installed in typesetting environments for use in the newspaper and book publishing industries. As early as July of 1963, several newspapers had begun to use computers for production purposes. THE WASHINGTON STAR, for example, used their general purpose computer for normal hyphenation and justification of news copy, and also expanded its use to include the generation of volume and production statistics and other accounting functions.

b. Equipment

Using an IBM 1620/1 with 40K of memory and a 1311 disk file, the WASHINGTON STAR was able to run an applications program that accepted internally coded input and that could justify every typeface size and line width that their linecasting equipment could produce. The computer stored the widths of the brass mats and the lengths of space

band travel; as each character was read by the computer, its brass width was subtracted from the previously set line length. This continued until the line was within justification range. The computer then searched to see if the next word space fell within range; if it did, then the line was filled and sent to the appropriate punch. If the word space was too long, the computer could attempt to insert extra fixed space inter- or intra-word, depending on the desired hyphenation frequency.

The paper used their disk files to store both type sizes and widths and to store their hyphenation dictionary; when the justification routine could not work, then the hyphenation routine was called. THE WASHINGTON STAR maintained an extensive hyphenation dictionary and a set of programs that attempted to hyphenate any words not located in the dictionary.

c. Expansion and Development

Because justification and hyphenation took up very little of the computer's time, THE WASHINGTON STAR also used their computer to provide production statistics, to schedule linecaster operations, and to gather statistics for editors and compositors to help them balance their presentation of the news and to help them lay out the paper. In addition, they had completed the development of a program that enabled them to take wire service copy as input, run it through the computer to store it and reprint it, and then

edit the computerized print-out of the story; the stored version was then re-edited and sent to the linecasting routine. That method eliminated the need to cast a dummy page that would then have to be broken up and re-cast after editing.

In addition to wire service editing, THE WASHINGTON STAR also used the same keyboard and computer to output photo-composed display advertising using a program developed by IBM and THE MIAMI HERALD. THE WASHINGTON STAR's use of their computer for "hot metal" typesetting and for peripheral accounting and editing tasks demonstrated an effective use of the equipment available at the time.

3. Computer Generation of Characters

Most early uses of computers in typesetting involved computer generation of code to drive either a phototypesetting machine or a "hot metal" linecaster. These processes generally used commands embedded within the text to be printed. These commands performed such functions as selecting type font and size, and positioning the characters on the output medium. In the late 1960's, interest grew in increasing the capabilities of computerized typesetting; there were usually severe limitations on the character sets available to computer output devices, normally line printers, but typographers had a wide variety of type styles and sizes to choose from. What was needed was a system that would make the advantages of typography available to the

computer; such a system would combine the speed of the computer with the versatility of the linecaster, and would make the result available to both machines.

In 1967, Allen V. Hershey, a mathematical physicist at the U.S. Naval Weapons Laboratory in Dahlgren, Virginia, developed a set of 1377 occidental characters and hundreds of oriental characters by hand using only graph paper to assist his work [Ref. 16]. He also developed FORTRAN typographic and cartographic systems that used his character library to compose finished pages of text, maps, drawings, and mathematical equations. This was one of the earliest efforts made to use the computer to take over the functions formerly performed by slower mechanical devices, so that both character generation and position could be handled at computer speed.

E. IMPROVED COMPUTER TECHNIQUES

1. Introduction

None of the "generations" of typesetting machines mentioned earlier is now totally distinct, since even the simplest devices in use today may host a mini-computer or a micro-computer. Therefore, the general opinion is that machines should now be classified based on the techniques used to store a master character and to generate that character for recording on the output medium. The classifications used are:

* Photographic/Optical (Photo/Optic).

* Photographic/Scanning (Photo/Scan).

* Digital/Scanning (Digital/Scan).

These classifications include the various graphics display terminals and CRT terminals familar to most people who use or who have seen computers.

The Photo/Optic method is fairly well known; it is the oldest of the three and was the method used by both the Cooperative Computing Laboratory at M.I.T. and THE WASHINGTON STAR in their offset printing procedures. The two "scanning" methods are less well known and generally consist of generating dots or lines on an output medium using a CRT or some kind of drum and light arrangement. The CRT is perhaps the most familar and its use involves generating a narrow beam of light and then deflecting the beam so that it will illuminate a very small area on the screen of the CRT. As some areas are "turned on" against a dark background, the character or pattern desired can be displayed on the screen as a dot pattern.

2. Photo/Optic Machines

This category includes the majority of phototypesetting devices available today. Usually the master character is stored photographically and is then generated optically for recording on the output medium. Most devices store the master characters for various fonts in negative form, since

the character must be illuminated after it has been selected. As it is illuminated, the ootical system, using a variety of lenses, can produce the required point size and position the character image on the output medium. The disadvantage inherent in this system is the mechanical movement required to position the characters and pages; this movement is very slow when compared to the speed with which characters can be selected and generated, even when the mechanical equipment is operating at its fastest. Even the character selection and generation is slow when compared with a fully computerized system, since this system must still use some kind of mechanical apparatus to select the characters.

3. Photo/Scan Machines

These machines again store the master characters photographically, but they generate the selected character using a dot or line generating mechanism to record the character on the output medium. These devices operate much like Photo/Optic devices until the output stage is reached; from that time on, Photo/Optic devices treat the character as a unit and Photo/Scan devices treat the character as a collection of "scan lines" which are built up to form the completed character.

The character is built up on the output medium using a series of closely spaced lines or dots which together form the character, usually using a CRT and an arrangement of

mirrors. This speeds up the typesetting process greatly because the characters are positioned electronically. Because the characters can also be sized electronically, there is no time lost while a lens turret is moved to position a different lens.

4. Digital/Scan Machines

Devices in this category store their character sets digitally in memory and use a dot or line generating mechanism to produce the characters on the output medium. This method allows the character definitions to be stored as binary digits in the computer's memory, providing rapid access to and display of the character information; however, it does require a large amount of storage for each character. For example, as characters become more complex and/or larger, more information about beam positioning and switching is required. Mr. Berg estimates that "for 100 printing characters at 10 point size, approximately 8000 (16 bit) words of storage are required....Only 35 characters at 72 point size can be stored in 8000 (16 bit) words. The precise storage requirement is dependent on typeface, point size, and character design." [Ref. 3, p. 6:10]

F. CURRENT CHARACTER DISPLAY TECHNIQUES

Digital/Scan techniques are most familar to computer scientists because alphanumeric CRT terminals and most graphics display processors use this method of character generation. Both the DATAMEDIA terminals (1500,1520,2500) and the RAMTEK GX-100 [Ref. 9] display processor, for example, use a bank of ASCII characters stored digitally in 7x12 dot matrices to generate visual displays. Characters for these devices all fit within a 5x7 dot matrix and the extra dot positions provide spacing between characters and between lines. The screen image is renewed 40 times a second; the electron gun is moved across the rear of the CRT in a side-to-side, line by line "raster" scan, and each individual dot is either illuminated or skipped to provide the required display. Figure 1 is an example of a character represented digitally and suitable for use by raster scan devices.

```
................
................
..00000000......
.00......0......
.0......0.......
.0.....00.......
.......0........
......00........
.....00.........
.....0..........
....00..........
....0.....0.....
...0......0.....
..000000000.....
................
................
................
```

FIGURE 1. Dot Matrix Representation

Another example of a raster scan device used as a graphics display unit is the CONOGRAPHIC-12 Interactive Display System discussed in Reference 13. This device supports a set of printable characters corresponding to the standard ASCII character set; standard size characters are drawn on a grid measuring 22x16 raster units and situated in the lower left corner of a character block measuring 40x24 raster units. The character block determines inter-column and inter-line spacing, normally 85 characters per line and 38 lines per page. Figure 2 provides an example of this technique. While the size of the characters on the screen may be changed, all characters are still drawn from the standard size definition.

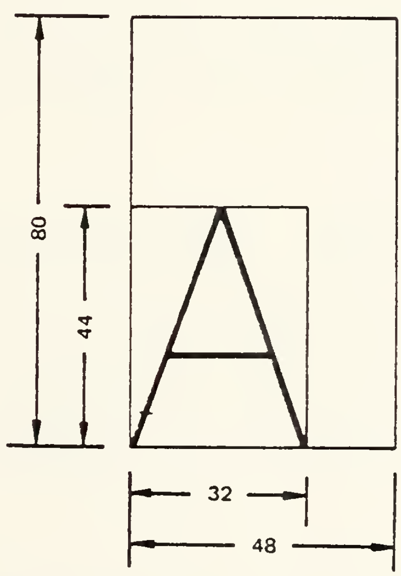

FIGURE 2. Character Block

The VECTOR GENERAL, AGT-10, and TEKTRONIX graphics display processors are examples of "refresh graphics" machines which store character sets digitally, but which

operate differently from the raster scan devices to generate visual displays. The VECTOR GENERAL [Ref. 12] is a highly sophisticated machine with many interesting capabilities, including the ability to draw curves. That capability allows the VECTOR GENERAL to store information about each character in its memory as a sequence of strokes which create character shapes. Each character is composed from a set of basic image elements [Ref. 12, p. 1-20], or draw figures, and the characters are drawn from these images as a series of arcs and vectors using that information.

In addition, the VECTOR GENERAL can display several fonts in four sizes; however, the sizes are all scaled from the standard size character definition, and the only fonts available are the standard ASCII character set and a font consisting largely of Greek characters and special mathematical symbols.

The ADAGE display processor (AGT-10) stores and generates its character set in a manner similar to that of the VECTOR GENERAL. However, because the AGT-10 does not have circle and arc hardware, all curves must be approximated by straight lines.

Both of these refresh graphics processors must re-draw the entire screen image approximately 40 times a second to prevent the image from fading or flickering.

TEKTRONIX [Ref. 14] display terminals (4010,4012,4014) are also Digital/Scan machines, but they differ in some ways

from the other refresh terminals. The character set is stored internally in dot matrix form, but when the characters are drawn on the CRT by the electron gun, the beam illuminates slightly more of the screen than the precise location required. Because of that, most characters appear as lines rather than as individual dots. The characters can be drawn in four sizes, but each size is based on a common character definition which is enlarged to the size required; the beam from the electron gun is then intensified so that an even larger spot is illuminated on the CRT, and the characters aopear to grow both larger and wider.

II. NATURE OF THE PROBLEM

A. COMPUTERIZED TYPESETTING AT NPS

This thesis was undertaken as part of an effort to improve the computerized typesetting capabilities at the Naval Postgraduate School in 1976-1977. Until that time, these facilities had been fairly limited and were rarely used. The programs used were written in the programming language C and were designed to be run under the UNIX operating system on the Computer Science Department's PDP-11/50 computer. The documents set in computer type are produced on a VERSATEC plotter/printer.

The original software to set type under UNIX was designed and written by Professor G.L. Barksdale, Jr. and was based on four fixed width fonts with common dimensions. The information to be set in these fonts was the output from TROFF, a text processor already available under UNIX. The actual typesetting was done by another program, a virtual typesetter. Professor Barksdale had also designed a "font editor" that was intended to allow a user to create new fonts or to modify existing fonts, in a manner similar to that used by most text editors. However, this font editor was not appropriate for use in the large scale digitization of fonts.

In an attempt to improve this situation, 48 additional fonts were obtained from external sources. Thirty-four of these fonts were already in digitized form and as a result were limited in point sizes available. They also required a great deal of storage in that form. Because the 14 Hershey fonts were available in vector form rather than dot matrix, they were acquired in the hopes that they could be adapted for use in computerized typesetting in a form that required less storage. The 34 digitized fonts, for example, required 643 512-byte blocks of storage while the Hershey fonts, stored in vector form, required only 193 blocks.

This thesis was directed toward finding an algorithm that would allow the Hershey fonts to remain in memory in vector form but convert them to a digitized form in any point size required by the user.

B. INITIAL CONVERSION

1. Original Format

The vector definitions of the 14 Hershey fonts were obtained from a tape available through the National Bureau of Standards [Ref. 16]. The original tape contained approximately 360K bytes of data representing 8-bit EBCDIC character codes. The tape contained just over 4600 card images, where each card image contained a character identification number, a card sequence number, and coordinate pairs. As a result, the data was essentially stored as a stream of

numbers.

Hershey's original definitions used integers between -49 and +49 to represent the endpoints of his vectors, with a (50,00) coordinate pair representing a "lift pen" command and a (50,50) representing "end of character". So that all of the coordinate pairs would fit into four bytes, negative values were subtracted from 100 and stored as two-digit numbers greater than 50 so that they could be differentiated from a positive integer. For example, (10,10) was stored as "1010" but (-10,10) was stored as "9010".

2. Converted Format

The initial steps required to read the tape, convert the records from EBCDIC to ASCII, strip away unnecessary characters, and so on, are contained in Appendix C. Once the input files had been properly prepared, they were put into a vector form which made it easier to access the vector definitions for a given character. A header table consisting of 256 16-bit words was established; each even numbered word from 0 to 254 corresponded to the appropriate ASCII octal codes and contained the character width of the character at that code location, while the odd numbered words contained pointers to the character definitions.

Within the character definitions, each coordinate pair was stored in a word of storage with the x-coordinate in the left byte and the y-coordinate in the right byte. Even the (50,00) and (50,50) pairs were stored in this

fashion rather than as "move/draw" and "endlist" bits in the conventions used by some graphics display processors. Since each integer used by Hershey could be represented in seven bits, the initial inclination was to use a format with the x,y coordinate pairs stored in two bytes, but with six bits used for the integer, one bit for the sign, and the extra bit used for the "move/draw" or "endlist" bits. This would have decreased the present storage requirements for a font by approximately 25%. That method was not used, however; it was decided that the amount of storage that would be saved was not worth the extra effort that would be required to manipulate the bits satisfactorily. In addition, the time would increase slightly, which is only a minor concern since this is usually done only once to a font, but the risk of introducing or failing to detect errors arising from the bit operations would also increase greatly.

III. DIGITIZING A HERSHEY FONT

A. FONT FILE FORMAT

All digitized font files at NPS follow a modified SAIL format [Ref. 4] that offers several advantages in memory requirements and that is tailored to 16-bit processing. The NPS format is displayed in FIGURE 3 on the next page. The first 256 16-bit words of each file contain a header table. Each of the 128 possible characters in a font has two words in this table which contain its character width and accessing information. Character 000 octal uses the first two words, character 001 uses the next two words, and so on. This arrangement provides an easy character accessing formula: twice the character code gives the location of the first word of information about that character in the header table. For each character defined, the first header table word contains the character width in the rightmost byte and a block counter in the leftmost byte. The maximum character width permitted is 255 pixels. The block counter contains a number between 0 and 255; it is a file offset in 512 byte blocks. The second word contains a byte offset, an unsigned integer between 0 and 65535, which is added to the block offset.

The character definition is accessed by seeking the required block offset, if any, and then seeking the byte

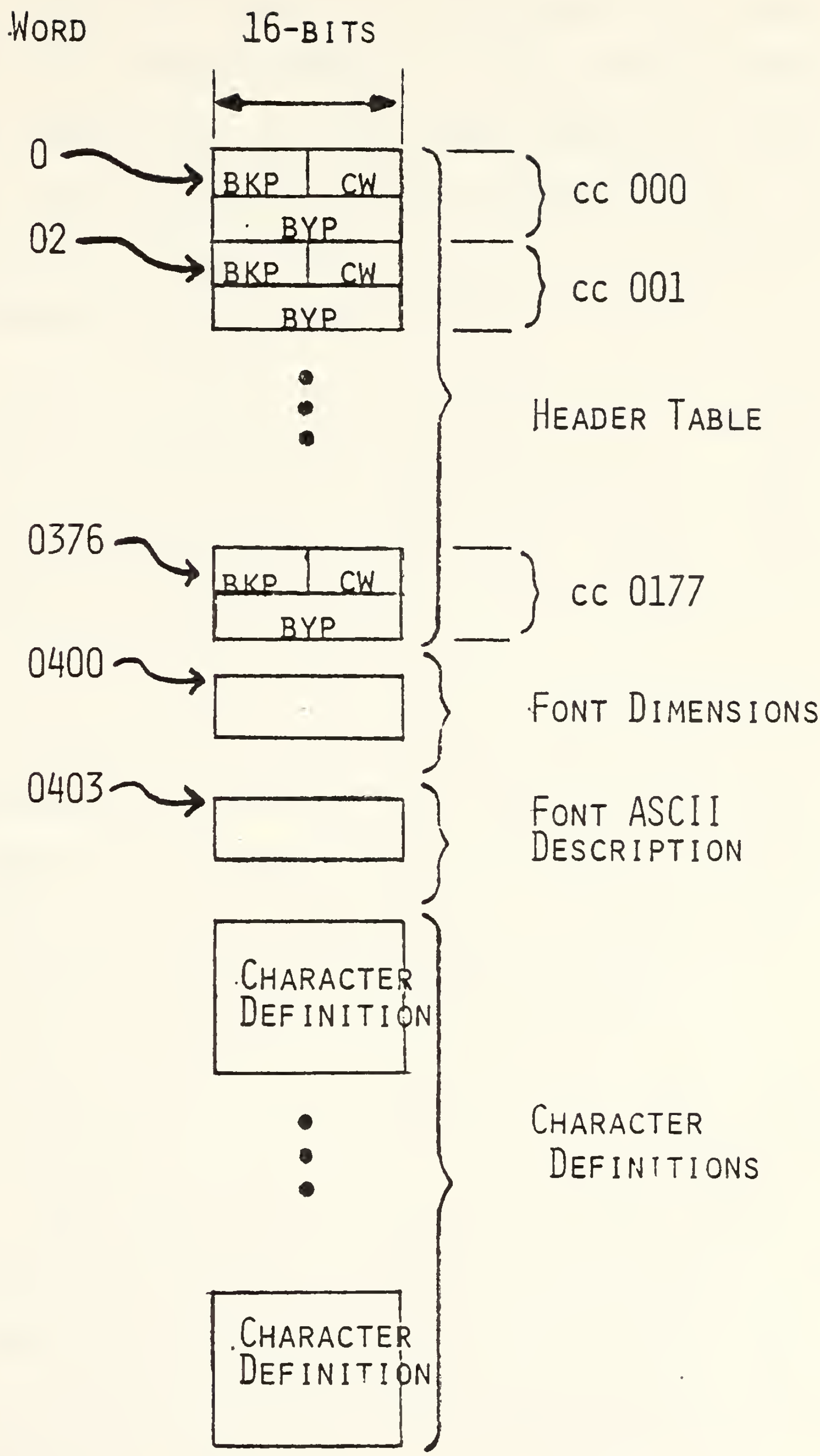

FIGURE 3. NPS Font File Format

offset. When accessing any character, a zero width and a zero pointer imply the character is not defined in the particular font. The dynamic aspects of the pointer structure in the header table allow for individual character accessing and for font files up to approximately 200K in size. However, a limitation in the "seek" system call limits the addressable storage to approximately 160K.

This situation is ideal in a minicomputer environment where core is limited and where large quantities of data reside on direct access devices. The three words following the header table in the font file contain information on the font height, on the width of the widest character in the font, and on the logical height of the characters. All dimensions are measured in pixels. An ASCII description of the font begins in word 260 and continues until an end-of-string delimiter ('\0') is encountered. No description is normally provided with any of the Hershey fonts.

The remainder of the file is composed of the character definitions pointed to by the information stored in the header table. Each definition follows the same format, and there are no requirements for definitions to begin on word boundaries. Each character definition is divided into two parts, the character dimensions and the character bit picture, as indicated in FIGURE 4.

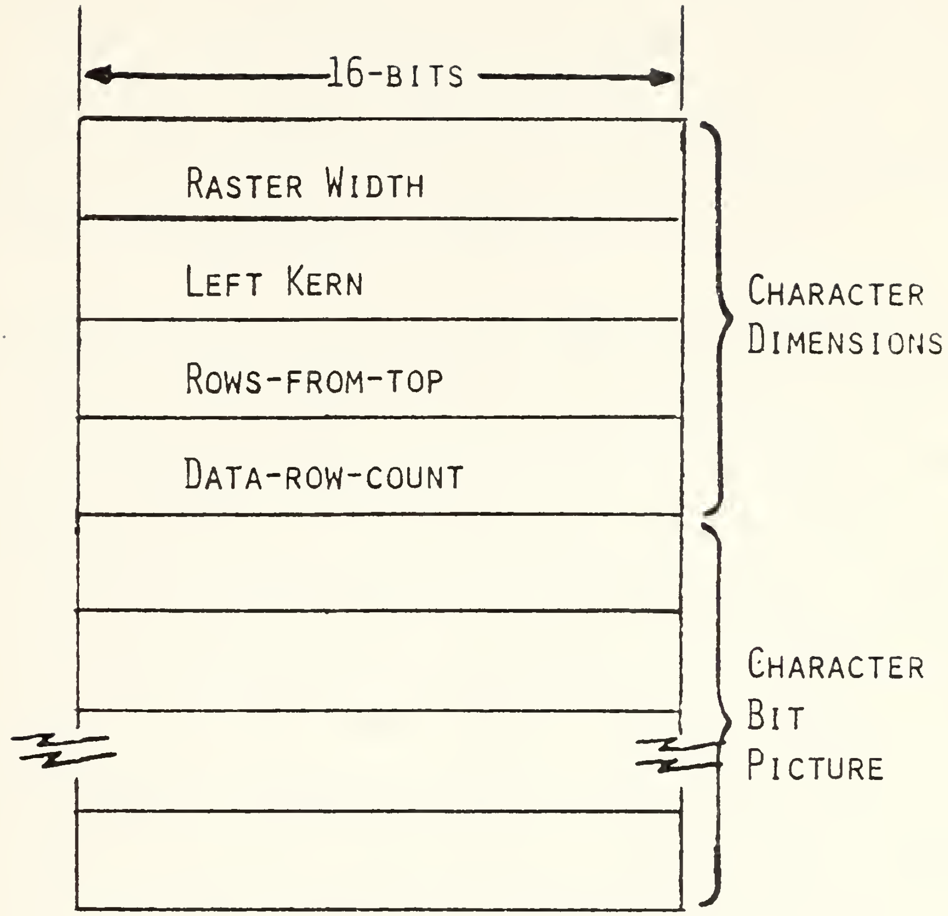

FIGURE 4. NPS Character Definition

First, there are eight bytes which hold the raster width, left kern, rows-from-top (rft), and the data-row-count (drc). These terms are defined in Appendix F. Next, a portion of the character picture is stored in consecutive bytes, raster line by raster line. Bits that are "on" (1's) represent space to be inked in, and bits that are "off" (0's) represent white space. Each character in a font is conceptually set in a rectangular frame which is as wide as the character's raster width and as high as the font's

height; hence, a great many characters have blank raster lines close to the top and near the bottom of the frame. These blank raster lines are not stored in the character definition. While the rft defines the number of blank lines at the character top, the drc specifies the number of nonblank raster lines stored in the definition, and the number of blank lines at the bottom is computed.

As an example, the process which "edf" would perform to display a character would be to access the character definition through the header table and to read in the four character dimensions. Now, if, for example, the raster width was 17, then 3 bytes would be required to store a single raster line, the third byte having its rightmost 7 bits wasted. The next three bytes hold the next raster line, and so on. "Edf" must display a number of blank lines equal to rft. It must then read and display the nonblank raster lines stored in the definition, and, finally, "edf" completes the picture by filling out the character height with blank lines. This process is similiar to the Stanford method. A more detailed explanation and some statistics can be found in Reference 6 and Appendix A. Appendix F illustrates character dimensions in more detail.

B. THE DIGITIZATION ALGORITHM

The digitization algorithm used was based on the standard slope/intercept formula for a line, y = m*x + b . After determining the logical top and bottom of a character, the end-points of each line in the vector definition were read into the program and the slope and intercept were determined. Then the line was scanned from top to bottom and from one side to the other using a "for" loop within a "for" loop. These integer values were converted to floating point with an assignment statement; if those values were within the required tolerance of the line being scanned, then that unique bit was changed from 0 to 1.

C. CONSIDERATIONS

1. Storage Requirements

An important consideration in designing the computer typesetting system was the amount of storage that would be required to hold the digitized fonts. All of the vector definitions, for example, were in the 5-7K bytes range; the comparative figures in Appendix B reveal that a 10 point digitized font requires approximately that much storage. At smaller point sizes less storage is required for digitized fonts than for the vectors, but as point sizes increase the storage requirements rise dramatically.

To minimize the storage requirements, all programs designed for this system used the convention mentioned in

paragraph A, where only the rows actually containing data were stored in memory. All rows containing zeros were added by the various programs as they executed. This technique reduced the storage requirements significantly, especially where most punctuation and lower case letters were concerned.

In addition, only one array of 4K words was used to hold each character individually as it was being digitized; this size allowed the digitization of the largest characters allowed, but was considerably smaller than an array that would hold the entire font during digitization would have been. As one character was completed, its bit picture was written to the designated file and the array was zeroed out in preparation for the next character. After the last character in the font had been digitized and written out, the blank (octal 040) was added to the font and the header table was written at the front of the file. This method used a minimum of storage, since only 519 extra bytes (used as a place-holder for the header table) were stored at any one time.

2. Sizing

Every effort was made to make all necessary variables proportional to the size of the font being digitized. Since Hershey's vector definitions were equivalent to a 10 point font, that raster height (29 pixels) was used as a base for determining the proportionality constant for modi-

fying the widths of the characters approoriately.

Two steps were necessary to determine font and character heights. First, the tallest upper case letter and one of the lower case descenders were scanned to obtain a base line and a logical height for the font. Then the largest characters in the font were scanned to dètermine a constant which would adjust the character heights to fit the desired raster height. The logical height and base line were adjusted by this amount, and the program could begin the digitization process.

3. Programming Techniques

One important consideration was to be able to address locations in memory up to the maximum font size allowed. Since even a "char *ptr" declaration allowed only 65K addressable bytes and permitted the possibility of the left-most bit being interpreted as a sign bit in arithmetic operations, the address pointer was declared as a long integer. The 32 bits were not all necessary because other limitations allowed the use of only 18 bits, but it did prevent unusual occurrences during mathematical operations.

Shifting operations were done in many places rather than a normal arithmetic operation, especially where the long integer was involved, for just that reason. Some bit masking was also necessary, normally to prevent a sign bit from propagating across a byte.

4. Types of Lines

After the slope of a line was determined, the execution flow carried the line into four possible sections of code. Because of the wav that the algorithm was arranged, it was necessary to treat vertical lines, horizontal lines, and lines with positive or negative slopes each somewhat differently.

It was difficult to arrive at a group of tolerances for lines with different slopes that would allow the lines to mesh smoothly to form a character. These tolerances were used to determine whether or not a particular bit in the character picture lay close enough to the line being digitized to be switched from 0 to 1. A step function was used to determine the tolerances to be used for lines with slope values between certain limits; as a result, there is some overshoot at points where slopes change enough to pass from one set of tolerances to another.

At first, nearly horizontal lines near the tops and bottoms of curved characters (0, Q, C, etc.) tended to either overshoot significantly or to vanish completely. Then horizontal and vertical lines grew out of proportion to the rest of the character. Some of these problems are illustrated in FIGURE 5. Eventually, the characters became more and more recognizable. The method used to smooth out the digitization involved studying the characters digitized with one set of tolerances with "edf", then graphing the charac-

ter from the vector definition, deciding how much tolerance was required for slope values between certain limits, and beginning the loop over.

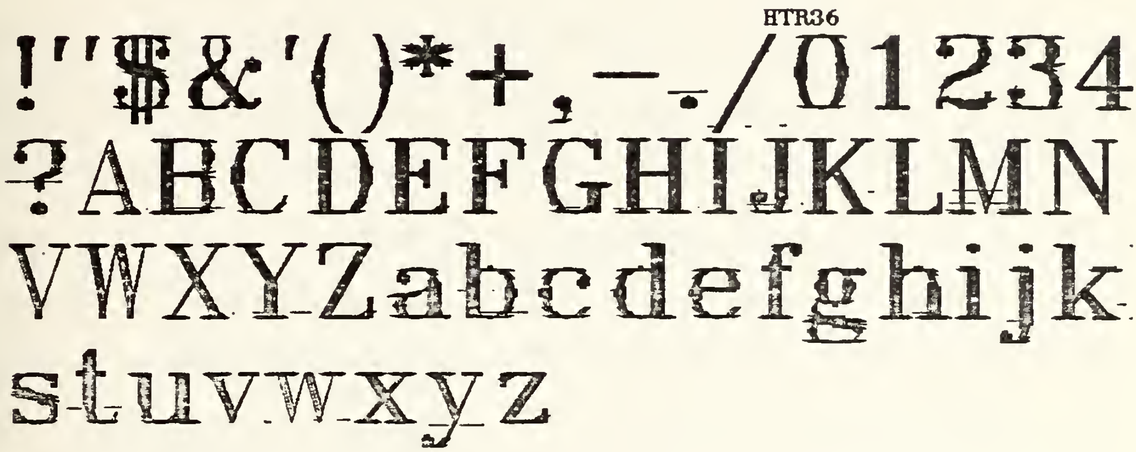

FIGURE 5. Problems in Digitization

Horizontal and vertical lines tended to grow thicker when digitized, so their widths were reduced programmatically by approximately half. The tolerances necessary for these lines were approximately one-half those of the tightest tolerances used for sloping lines. Sloping lines had to be thickened by the same means, but even here there was a difference; lines with a slope that was very close to horizontal required an even larger assist than did other lines. Lines with slopes between 0.5 and -0.5 (nearly horizontal) required very tight tolerances to keep them from thickening excessively, while lines between 0.5 and 3.0 and between -0.5 and -3.0 received somewhat larger tolerances. Lines with slopes from 3.0 to 7.0 and from -3.0 to -7.0 were

essentially left alone, but lines with slopes greater than 7.0 or less than -7.0 (nearly vertical) required very loose tolerances.

In general, characters such as "A", "M", "Z", and others that were essentially composed of straight lines, no matter what their slopes, transitioned from vector to raster form clearly and were very clean. This resulted largely because the same tolerance was used by the algorithm throughout the line and the character. In other words, there were very few breaks in the continuity of the lines that defined the character. There were minor problems such as notching in the base of the "M" or in the point of the "A" and a thickening in the right foot of the "A" and the "X"; these were not immediately obvious, especially at point sizes that would normally be used for typesetting.

Characters such as "O", "Q", "d", "c", and others that required the use of many small lines to approximate curves were usually ragged in places after digitization. Because different tolerances were used on lines that were linked, the effect was not as smooth as it was for the straight line characters. As a result, characters of this type sometimes appear somewhat ragged, especially at larger point sizes where this effect is easily discernible.

5. Floating Point

Floating point arithmetic was used extensively in the digitization process. While this made the program

slightly slower, it had been decided beforehand that floating point was necessary to achieve the accuracy required to prevent holes or extraneous lines and bits from appearing in the dot matrix.

D. LIMITATIONS

1. Time

One of the lesser limitations imposed upon the user in this area is the time required to digitize a Hershey font. While the time required sometimes seems out of proportion, especially with larger or more complex fonts, many of the reasons for this seeming slowness have been explained previously. In addition, the time required to digitize the largest fonts possible is still on the order of approximately 15 minutes at the worst. The times can be improved by digitizing fonts at times when system usage is low, and by digitizing fonts only once and storing them between uses. This should be the normal mode of operation when using Hershey fonts.

2. Appearance

The appearance of most fonts at larger sizes has already been discussed to some extent and a comparison of the Duplex Roman font at 10, 20, 30, and 40 point sizes is available in Appendix B. On the whole, the program will digitize fonts fairly well up to the size limitations discussed in the next section. Fonts with more vectors in the

character definition will not be as ragged as those with only a few lines.

3. Size

An initial design decision was made to limit the fonts to a raster height of 255 pixels, which is equivalent to 91 point. As a result, the array declared in "makehf" to hold each digitized character definition as it is converted is designed to hold one character 255 pixels high by 255 pixels wide at its maximum.

An additional constraint is imposed by the structure of the font files. Because the character width and a block count, if present, each occupy a byte, the maximum value for the block counter is 255. As the block counter approaches that figure, specifically at 253 blocks, the program will switch modes and use the same block counter from that point on, but the byte counter will be reset and will increase up to 65535. This will permit the user to approach 200K bytes for the digitization.

The size of a character that can be edited by the font editor is arbitrarily set at 42 point, the size of the largest already digitized font available, SIGN41. Therefore, Hershey fonts larger than this can be created, but they cannot be edited. However, they are still usable by "prfont" and "signmkr".

IV. CONCLUSIONS

A. A COMPUTERIZED TYPESETTING SYSTEM

The initial computerized typesetting capability at NPS has been expanded considerably as a result of thesis efforts described in this paper and in Reference 6. Specifically, 48 variable width fonts in a variety of sizes and styles have been added. These efforts are incomplete in that a virtual typesetter that sets variable width fonts has not yet been implemented; however, an additional program has been written which will set these fonts and which performs a limited number of text formatting functions.

At the present time, this expanded typesetting system is designed to use four programs. The user has "edf" and "makehf" available to create or modify fonts, and "prfont" and "signmkr" are available to display his efforts. The font editor, "edf", has been expanded and modified considerably; it is documented in Appendix A. The program "makehf", which is described in the previous chapter, was the end result of the author's thesis efforts and provided a substantial contribution to the increased capability of the NPS computerized typesetting system. This program allowed the user to convert Hershey's vector definitions into dot matrix representations that could be used by the computer; these definitions could be converted to a variety of sizes,

subject only to a few limitations.

The display routines developed for the system, "prfont" and "signmkr", are described in Appendix D, together with the vector display routine "drawhf". "Prfont" is designed to display one font at a time by examining the header table and printing all defined characters in the desired font. "Signmkr" is more sophisticated, and allows the user to specify a limited set of text processing commands to set type to his specifications.

B. ADVANTAGES/DISADVANTAGES

1. Advantages

The adaptation of the Hershey fonts for use in computerized typesetting has improved both the quality and the variety of fonts available for use. It is now possible for a user to access more elaborate fonts, or to access fonts in several different alphabets. These could now be used for special purpose applications or for accenting or highlighting standard printing applications.

This scheme also allows the creation of fonts at larger sizes than are available through the SAIL set. The algorithm holds up well at large sizes for most fonts and leaves very few holes, especially on Triplex or Gothic fonts where a large number of vectors are used to make up the character definition.

For most purposes, the Hershey fonts digitize extremely well. There are usually only a few holes, even at very large point sizes, in most fonts. They tend to break up at 8 point or smaller (due to pixel size). Above 50 point (because of line spread) some small extraneous lines may appear. In the range that would include most normal uses the digitized Hershey fonts are serviceable, with the exotic fonts looking especially good.

2. Disadvantages

The vector digitization method has several disadvantages over and above the current lack of a virtual typesetter previously mentioned. First of all, it is slow, especially for larger and/or more complex fonts. Therefore, it is not suitable for on-line digitization of individual characters. However, this is easily overcome by deciding beforehand which fonts will be required and then digitizing them before beginning the typesetting process.

Secondly, the algoritnm is somewhat inefficient. A large portion of the overhead is incurred through the use of floating point arithmetic and this was deemed necessary. However, some time is also lost in array accessing; the conversion from arrays to pointers could increase the digitization speed somewhat.

In addition, the algorithm begins to leave holes in the digitization as fonts become extremely large. An exception is the Duplex Roman font, which begins to break up at a

very small size because of the arrangement of its component vectors. In general, this is not a significant problem with most fonts.

C. PERFORMANCE EVALUATION

1. Testing the Algorithm

To determine which parts of the algorithm required the most execution time, an execution profile was run on the program under a variety of conditions. A "monitor" system call was inserted into the beginning of the digitization algorithm so that the entire program could be profiled, and the program was then compiled using the shell command "cc -c -f -O -S makehf.c"; the object file resulting from that command was loaded using "ld /lib/fcrt0.o makehf.o -la -lc". The "a.out" file produced by the load was then used to digitize the Simplex Roman font at multiples of 10 points between 10 and 70 points. These profiles provided the test data used below; other fonts were digitized for comparison ourposed as noted in paragraph 3.

2. The Execution Profile

The execution profile revealed that one section of the program required, as a minimum, approximately 60% of the program execution time. This section consisted of the four "for" loop pairs previously described in Chapter III. These loops are for horizontal and vertical lines and lines with positive or negative slopes. The majority of the floating

point arithmetic was used in these loops to scan each line in the character defintion and to turn on the appropriate bits in the character picture.

The table below can be used to compare three quantities: the point size of the digitized font, the time required to digitize the font to that point size, and the total time that the program spent in the four digitization loops together. The "real" time required to digitize a font versus the point size is shown is FIGURE 6, as is the "user" (CPU) time versus point size. The point size versus percentage of time spent in the digitization loops is shown in FIGURE 7.

	% spent in digitization	Real	TIME User	System
HSR10	63.3	0:26.0	0:10.8	0:08.7
HSR20	73.7	0:59.0	0:35.5	0:08.2
HSR30	78.0	1:27.0	1:12.9	0:09.0
HSR40	80.3	2:44.0	2:04.6	0:12.5
HSR50	80.7	5:22.0	3:14.3	0:19.5
HSR60	82.1	9:33.0	4:34.6	0:30.3
HSR70	81.8	14:02.0	6:14.6	0:34.8

3. Different Fonts

Several more complex fonts were digitized at various point sizes to determine whether or not the performance of the algorithm would be affected. While the percentages of time spent in the different digitization loops determined by

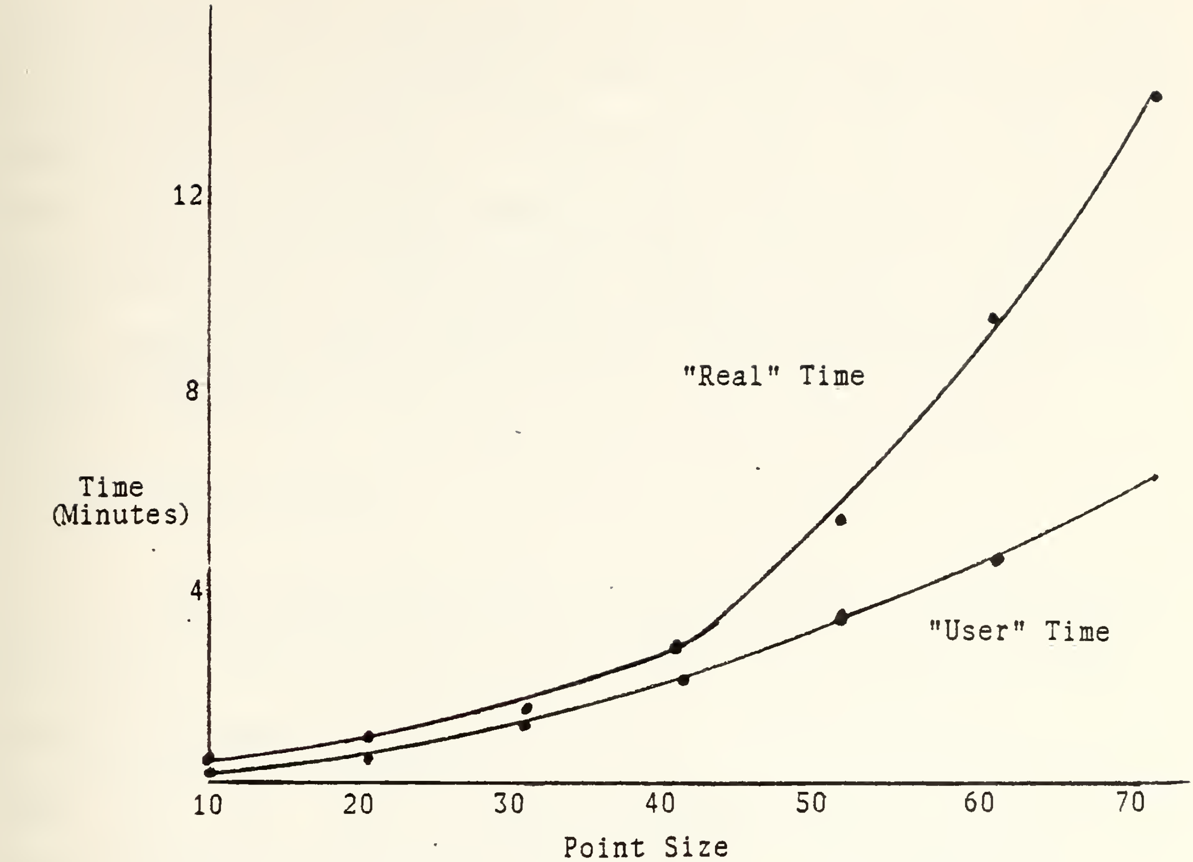

FIGURE 6. Digitization Time vs. Point Size

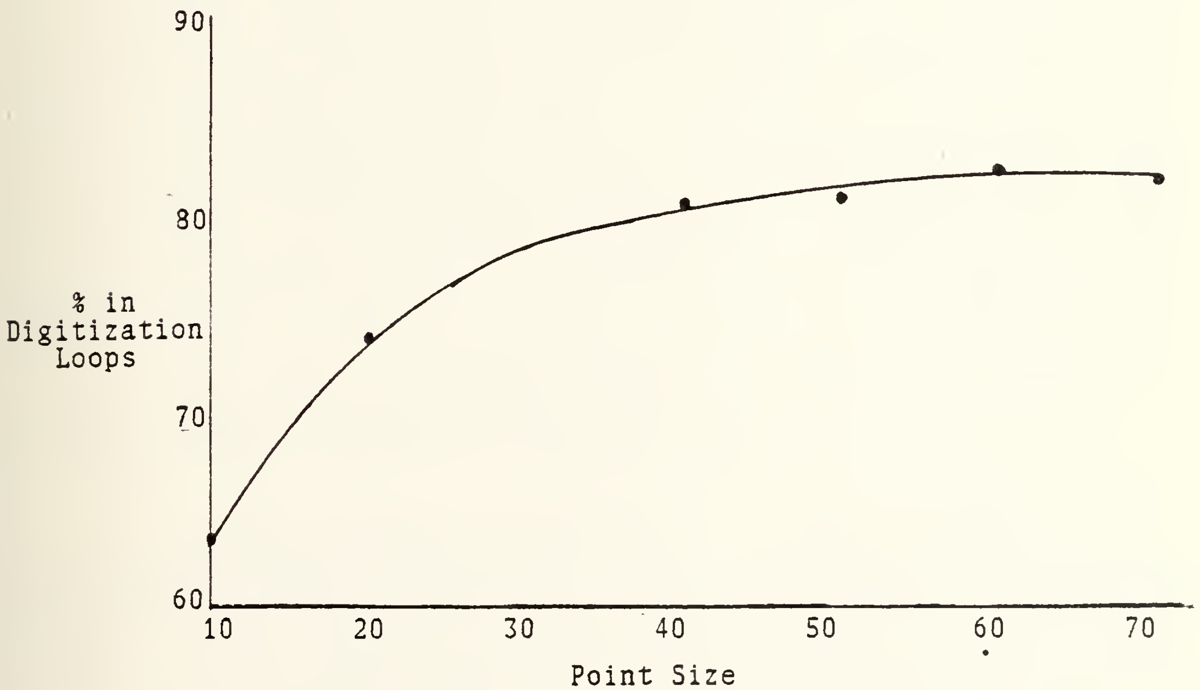

FIGURE 7. Point Size vs. Percentage Of Time In Digitization Loops

the slopes of the lines were different, the overall amount of time spent in those portions of the algorithm remained approximately the same. If anything, the total times for the digitization loops were slightly less for the more complex fonts than for the simpler fonts; however, the times in the "read" portions were slightly higher because more lines had to be read in.

4. Conclusions

From the table above and the figures, it is possible to arrive at two conclusions. One conclusion is that as the point size increases, the "real" time required to digitize the font also increases; this increase is non-linear and is very slow at lower point sizes, but begins to increase dramatically between 30 and 40 point. This reflects the time that the user must wait at a terminal for his digitized font file to be created; a second time correlation, not quite so dramatic as the "real" time required but just as important, is the corresponding rise in "user" (CPU) time as point size increases. This indicates that larger fonts incur a non-linear increase in CPU time that is reflected as an even larger increase in "real" time.

A second possible conclusion is that one section of the algorithm contributes significantly to the time required for the program execution. The percentage of time required in the digitization loops was never less than 56 and seemed to level out at just over 80 for the larger fonts; if this

portion of the algorithm could be speeded up the time required for digitization, especially digitization of the larger fonts, could be improved.

It should also be noted that as the point sizes grow larger and the percentage of time spent in the digitization loops increases, the relative amount of time spent in the "read" portion of the program decreases until it becomes inconsequential at the larger point sizes. Therefore, the improvement of the digitization process becomes the central problem in making the algorithm faster.

D. POSSIBLE IMPROVEMENTS

1. Better Digitization

While the present digitization algorithm is fairly effective, it could be improved in some places. Procedures to eliminate extraneous bits or overshoots that extend outside of the main character definition, or to detect and fill in small holes or odd bits within the character defintion, are possible.

Some of the raggedness and overshooting in curved characters may be minimized or eliminated by changing the tolerance function used. If a function that allowed for gradual changes in the slope (such as a sinusoid) were used in place of the step function currently being used, the approximations of curves could be improved and any remaining raggedness would be more difficult to see. Rather than use

such a function in the program itself, the values should be computed once and then put into table form for program use.

An additional alternative might be to use a means other than the slope/intercept formula for a line to control the digitization. Cubic splines are one possible choice; the use of splines should minimize round-off error, and they are perhaps better suited for digitizing the curves that have presented the majority of problems during this research. Since splines provide a smoother fit over sparse data, they may be ideally suited to font digitization.

2. A Faster Algorithm

Several means to increase the efficiency of the algorithm have already been mentioned, including minimizing floating point arithmetic, switching from arrays to pointers, and so on. In addition, since the vectors are read in one point (two bytes) at a time, one "read" operation that brought in the whole character definition would somewhat decrease the time required for system calls.

The "for" loops used for digitization are arranged so that one goes from the logical top of the character to the bottom, but the other runs from 0 to the font width. Since all of the font width is not usually required, this inner loop could scan only the character width.

E. FONTS AVAILABLE

1. The SAIL Fonts

The 34 digitized fonts were acquired from the Artificial Intelligence Laboratory at Stanford University and were converted to a file format compatible with the PDP-11 [Ref. 6]. These fonts were either designed at Stanford or acquired by them through the ARPA net from other artificial intelligence centers. SAIL fonts use a 7-bit code similiar to ASCII; however, the SAIL set uses many of the ASCII control codes for additional printable characters. There are some additional minor differences in character usage. The complete SAIL character set is listed in Appendix G with a complete listing of all SAIL fonts converted for use at NPS.

2. The Hershey Fonts

The 14 fonts available in vector form were converted for NPS use from a set of fonts created by Allen V. Hershey in 1967 [Ref. 16]. These fonts offer several type faces in Roman, italic, and script, as well as complete alphabets in Greek and Cyrillic, and in Gothic English, German, and Italian. A complete listing of the Hershey fonts is available in Appendix E, together with sample listings of the fonts in digitized form.

The Hershey fonts are stored in vector form and are not suitable for use by typesetting programs until they are

converted to dot matrix form by the user. This can be done by using either the Hershey font conversion program "makehf" or the font editor. These fonts may be digitized in any size desired by the user, subject to some limitations on the programs involved. The programs required and their limitations are discussed in Chapter III and Appendixes A and B.

APPENDIX A. FONT EDITOR

A. USING THE FONT EDITOR

1. Basic Structure

"Edf" is an interactive program which allows a user to create new fonts or to modify or maintain existing ones. It was originally designed by Professor Barksdale to create and manipulate the fixed width, 20 x 16 pixel fonts. The current version of "edf" is considerably larger than its predecessor, a growth resulting from the addition of modules to manipulate the more complex and more dynamic format of the new font files.

Creating a font may be accomplished by one of several means. First, a call to "edf" with no arguments indicates that the user desires to create a font from scratch. The user must specify the characteristics of the new font and then use the "a" (add) command to create specific characters at each character position. Repeating this process for 128 characters can become exceedingly tedious. A more efficient option is to create only a few new characters and to then use the "i" (include) command to include other characters from a compatible font. "Compatible", in this case, means that both fonts have identical heights and logical heights and that the characters being included are no wider

than the maximum character width of the font being created. A third option, somewhat similar to the second, is to use the "d" (delete) command to remove unwanted characters from a selected base font.

To edit an existing digitized font file, "edf" requires an argument consisting of either a font file name or a complete path name. In the first case, the font editor assumes that the font is located on the directory "/.fonts.01/font/" and prepends that string to the argument before issuing a system call to open that file. If a complete path name is used, "edf" will open that font file. If the font file is missing or if the font file contains invalid information, then "edf" will exit with an appropriate error message.

A Hershey font, digitized to any desired size and subject to the limitations discussed later, can also be created using the font editor. This is done by calling "edf" with at least one argument. The first argument must be of the form "-HXY", where the minus sign informs the editor that it must digitize a Hershey font and "HXY" is a valid font from the list of fonts available found in Appendix E. This argument must contain those four characters. The point size desired may be input as a second argument. The default point size used is 10 point, and the editor can edit up to only 42 point. Whether the newly digitized Hershey font is written to another directory or not, the most recently created Hershey font is normally left on

directory "/.fonts.01" and is named HFONT.

Some examples of valid calls to "edf" are listed below:

a) edf

This indicates that the user desires to create his own font. He may give it any name desired when he writes it out, ending the edit session.

b) edf SIGN41

The user wants to edit font file SIGN41, which had better exist (and SIGN41 does) on directory "/.fonts.01/font/SIGN41".

c) edf /usr/doyle/fonts/HTR42

The user wants to edit an existing Hershey font file called HTR42, a Triplex Roman font at 42 point, on directory "/usr/doyle/fonts/".

d) edf HSR20

The user wants to edit an existing Hershey font file called HSR20, a Simplex Roman font at 20 point, on directory "/.fonts.01/font/".

e) edf -HGE 36

The user wants to create a Hershey font file in the Gothic English type at 36 point. He may write it to any

directory after it has been digitized.

f) edf -HCS

The user wants to create a Hershey font file in Complex Script type. The point size defaults to 10 point, and the font may be written to any directory at the conclusion of the edit session.

In the edit mode, the header table, the font dimensions, and the font description, if any, are read into the program variables. When a specific character definition is required by the program, the bytes containing the dot matrix definition of that character are read into a character buffer, and blank lines are inserted at the top and bottom of the definition if required. A character definition leaves the character buffer and is put on a linked list if it has been modified during the current edit session. As a new character definition is required, it is read from either the font file or from the linked list if it has been changed previously. Characters which are not defined in the font, such as the control characters below octal code 040 in the Hershey fonts, or which are non-printable, such as the blank, are flagged and may not be displayed with the font editor.

Changing the current character code will not cause a character definition to be read into the buffer unless it is followed by a command which requires the definition; for example, "-" or "056" will change the current character code,

but no definition is read into the buffer until a command like "l" (list) or "e" (edit) is given.

Once a character has been modified, its new definition will not be read from the character buffer to the linked list until the current character is changed or until the user gives the "w" (write) command. An attempt to end the edit session without writing out a file containing changes will generate one warning. The user must specify the name of the file that he is writing to. The editor will not allow the user to write to the same file that he is editing from or to write to "HFONT"; hence, no font file is inadvertantly destroyed. The editor writes to the specified file, incorporating character definitions from the linked list and from the font file, updating the header table as necessary. As a final gesture, the editor writes out the size of the file in decimal. Renaming the new font file or replacing an old file with a new one remains the responsibility of the user.

When using "edf" it is most efficient to complete all desired modifications to one character before proceeding to another.

2. Commands

The basic command line consists of three parts: the current character selector, the command itself, and arguments, if any, to the command. The current character may be considered a pointer to a code position in a font. For exam-

ple, when the current character is 0101, then any character listing or editing will be directed toward "A" which has the code 0101. Whenever a character picture, or a portion thereof, is displayed, each raster line is composed of whole bytes. For example, if the raster width is 17, then all 3 bytes required to hold the 17 bits will be displayed. Changing the character picture to the right of the 17th bit is a superficial change, since modifications made outside the raster width are ignored.

a) <number>

Change the current character to <number>. The number may be octal (preceded by a zero) or decimal. Any number greater than 127 is converted to 0, and anything less than 0 is converted to 127. Any command may appended to <number>. The effect is to change the current character first and then to execute the appended command.

Examples: 0176, 0, 161, 78c 0 25, 16a.

b) +¦-

Increment (decrement) the current character. Wraparound occurs as in <number> above. Either <+> or <-> may be used but not both on the same command line. Any command may be appended to either, and the effect is to increment (decrement) the current character first and then execute the

command. Only one "+" or "-" may be used on a command line.

Examples: +1, -, +, +e, +c0 40.

c) [<number>]¦[+]¦[-]a

Add a new character to the font at the current character position. The "a"(add) command is complex. A "p"(parameter) command is executed automatically. Follow the displayed instructions to input the dimensions of your new character. Remember that your new character is being defined at the current character. After exiting the parameter command loop, you may use the "c"(change), "e"(edit), "s"(shift), or "l"(list) commands to form the desired character picture. The character buffer has previously been zeroed. If you use <number>, "+", or "-" to change the current character before you are satisfied with the new character picture, the unsatisfactory picture gets stored! If this happens, list the character and continue.

Examples: +a, -a, 056a, 19a, a.

d) [<number>]¦[+]¦[-]c[<number>] [<number>]

Change lines "s" thru "e", prompting for each line. "c" alone sets "s" to 0 and "e" to "height-1". "c" followed by one number sets both "s" and "e" to that number. "c" with

two numbers sets "s" and "e" accordingly. The numbers may be octal or decimal, and a space is required between two numbers.

Examples: +c, -c0 10, 077c 1 044, c, +c 10.

e) d[<number>] [<number>] [font file]

Delete characters "s" thru "e". "d" alone sets "s" to 0 and "e" to 127, effectively deleting the entire font. "d" with a single number deletes that character code. "d" with two numbers deletes "s" thru "e" inclusive. Numbers may be octal or decimal, and a space is required between two numbers.

Examples: d, d5, d 0176, d 0 057.

f) [<number>]|[+]|[-]e[<number>] [<number>]

Edit lines "s" thru "e", prompting for each line. "s" and "e" are set as in "c"(change). While editing a line, "cntl-d" completes the line as it was. This command uses the NPS line-editor functions in the terminal handler.

Examples: e, 077e0 10, +e 3 5, -e, 017e 12.

g) f

Turn on (off) a flag controlling the display of character dimensions. Once turned on, character dimensions are displayed every time a character definition is fetched. Displaying is turned off by a subsequent "f". "f" may be prepended to any command.

Examples: f, f1, +fe 0 10, 0176f1 0 10.

h) i[<number>] [<number>] filename

Include characters "s" thru "e" from the font file "filename". "s" and "e" are set as in the "d"(delete) command. If the font file being edited or created and "filename" are not compatible, then the include will not occur. Subsequent uses of "i" do not require "filename"; unless, of course, you wish to include from another font file.

Examples: i 0 057 BDJ8, i HCS20, i.

i) [<number>]¦[+]¦[-]1 [<number>] [<number>]

List lines "s" thru "e" of the current character. "s" and "e" are set as in "c"(change).

Examples: +1 0 10, -1, 1, 0761, 1 12.

j) n

Display the font description and a table reflecting the status of the edit session. The description tells you what you're editing, if you've forgotten. The table is a handy way to keep track of how much you've accomplished.

Example: n.

k) p

The "p"(parameter) command executes an interactive module of "edf" which allows you to modify character and font dimensions and description. A set of instructions will be displayed and may be recalled if required. This module is quite versatile. Keep in mind that character and font dimensions are being changed, not character pictures.

Example: p

l) q

Quit warns you if you've made changes and have forgotten to write them out; otherwise, it exits, closing any open files.

Example: q.

m) [<number>]¦[+]¦[-]sl¦r¦u¦d [<number>] [<number>]

Shift lines "s" thru "e" one pixel left(l), right(r), up(u), or down(d). The resulting lines are automatically displayed. "s" and "e" are set as in "c"(change).

Examples: +sl0 10, 044su 10, sr, -sd.

n) w filename

Write out the font file being edited or created to "filename". "w" must have a "filename" and will not allow you to write to the font file being edited. "w" displays the byte size, in decimal, of "filename" and then performs a "q"(quit). Be patient! Writing out a font file takes longer than writing out a normal file.

Examples: w temp, w /.fonts.01/font/HCI20.

o) <rubout>¦<break>

Either key causes an interrupt which is trapped. Whatever was going on is stopped, the previous environment restored (the command loop is reentered), and you may continue. Neither key undoes anything; they merely give a mechanism for killing commands without killing the program.

3. Limitations

There are two types of limitations to "edf". First, there are some commands implemented in the original version which are not available in the current version. They included "nice to have" commands such as folding character pictures, italicizing fonts, and producing bold fonts. These commands were not included due to time constraints but could easily be added in the future. Second, "edf" has not had a thorough testing. There are many checks throughout the program which were included to detect bad font files and to prevent the program from "crashing". "Edf" is good at screening commands and at flagging bad ones. Although it is possible to string some commands together on one command line, some combinations are bound to produce strange results. It is safe to combine commands only as described in the preceding section. Despite its limitations, "edf" is an extremely useful tool. It was developed early in the thesis research and used extensively to purge and inspect fonts.

NAME

edf -- font editor

SYNOPSIS

edf < -Hershey font [point size] > ¦ < SAIL font > ¦
< Hershey font >

DESCRIPTION

"Edf" is an interactive font editor that provides a means of creating and maintaining fonts. If called with no arguments it will enter the "create" mode. If given just the font name, it will prepend "/.fonts.01/font/". The editor will also accept a full path name. "Edf" also digitizes Hershey fonts to a specified point size.

Because of the size of the buffer used, "edf" can be used only for characters below 120 pixels (42 point) in size. All of the digitized fonts are less than that size, and the editor will not create Hershey fonts over that size.

Command Summary:

< number >	Change the current character to <number>
+ ¦ -	Increment/decrement the current character
a	Add a character
c	Change a line
d	Delete a character or a font
e	Edit a line
f	Turn on/off character dimensions
i	Include a character
l	List the current character
n	Display the status of the edit session
p	Modify character and font dimensions
q	Quit, end the edit session
s	Shift [l]¦[r]¦[u]¦[d]

w — Write to a file

<rubout> — Reenter the command loop

<break> — Reenter the command loop

FILES

/.fonts.01/HFONT
/.fonts.01/makehf
/.fonts.01/font/<SAIL font> ¦ <Hershey font>

SEE ALSO

makehf

BUGS

A call to the font editor must contain the correct name of the font file desired. No input checking is done; the only errors that will be detected are those that occur when trying to open a non-existant file.

"Edf" tries to tell you that the Hershey Complex Cyrillic (HCC) font has characters at octal codes 000 and 003, when the characters are in reality at 001 and 004.

```
                    /*                      */

                    /*          edf.c       */

                    /*                      */

#define error return(1);

int readfp, writefp;    //file descriptors
int ptsize;             //Hershey font point size
int pid;                //Child process id
int freenode;           //ptr to next free node in llist
int infont;             //current character
int wrflag;             //initially, 0. incremented on
                        //any change to flag a quit without
                        //writing
int wr;                 //flag to turn off displaying of
                        //diagnostics during file writing
int max;                //32677 used to denote base node
int ht, maxw, lht;      //font dimensions
int blkc; char *bytc;   //block,byte counters
int edit;               //set to 1 when in edit mode
int delete;             //flag in checking for empty fontfiles
int tht, tmaxw, tlht;   //temp font dimensions
int dim;                //char dim diplay control switch
int include;            //flag preventing access to llist
                        //during an include command
int rw, lk, rft;        //character dimensions
int bot, bytes, drc;    //    "          "
int s, e;               //command arguments
int in;                 //1 if current character definition is
                        //in character buffer, 0 otherwise
int c, peekc;           //characters on the command line
int first, last;        //line ptrs in character buffer
int chmod;              //1 if char in  buffer was modified
int *n;                 //integer pointer
                        //0, otherwise
int sgtty[3];           //buffer for gtty(II)
int savetty;            //terminal status
int onintr();           //address of interrupt trap
int *chardef, *p;       //character pointers
char cstat;             //holds status of char in char buffer
char des[80];           //holds font description
char ibuf[36];          //buffer for read(II)
char tbuf[4000];        //character buffer
int hdr[256];           //hdr table of edited/created font
int fhdr[256];          //temp hdr table during an include
struct node     {       //a node holds info on a single
                        //character stored on the llist
   int code;            //character code
   char *def;           //ptr to char definition
   int nsize;           //size of new definition
   char stat;           //status of modification
   struct node *next;   //ptr to next node in llist
}  llist[129];
```

```
struct node *head;     //ptr to head of llist
struct node *avail;    //ptr to next free node
struct node *current; //ptr to node found in FIND
struct node *insert();//node returned by INSERT
char rfontfile[40];    //fontfile being included from
char wfontfile[40];    //file being written to
char sfontfile[40]{"/.fonts.01/font/"};
                       //pathname header of fontfile to
                       //be edited
char hfsize[5]{"10"}; //default pt size for Hershey font

main(argc,argv)
   int argc; char **argv;    {
   int i;
   if (argc > 1) {//arguments->edit mode
     if (argv[1][0] == '-') {//digitize Hershey font
       if (argc == 3) {//check any point size
           if ((ptsize = atoi(argv[2])) > 42)    {
              printf("point size exceeds 42");
              exit();
           }
           p = hfsize;
           for(i=0;(*p++ = argv[2][i]) != '\0';i++);
       }
       pid = fork() ;
       if ( pid != 0 )
         while ( pid != wait() ) ;
       else  //create process to digitize Hershey font
          execl("makehf","makehf",argv[1],hfsize,0);
        readfp = open("/.fonts.01/HFONT",0);
      }
      else if ( argv[1][0] == '/' ) {//full pathname
         readfp = open(argv[1],0);
      }
     else   {
         p = argv[1];
         for(i=16;(sfontfile[i] = *p++) != '\0';i++);
         readfp = open(sfontfile,0);
      }
   edit = 1;
   }
   init();
   signal(2,onintr);    //set interrupt trap
   while (1)      {
      setexit();
      printf("\n%3o> ",infont);
      peekc = (peekc == '\n') ? 0 : peekc;
      if (command())      {
         printf("?\n");
         if (peekc != '\n') while((c=getc()) != '\n') ;
      }
   }
}

init()    {
```

```
    int i;
    if (edit)     {
        if (readfp > 0) fonthdr();
        else     {
            printf("fontfile not found\n");
            exit();
        }
    }
    else {//create mode
        zhdr(hdr);
        printf("\nfont height ? ");
        while((ht=getnum()) < 0 || ht > 120)     {
            peekc = 0; printf("height ?  ");     }
        printf(" %d !\n",ht);
        peekc = 0;
        printf("maximum character width ?  ");
        while((maxw=getnum()) < 0 || maxw > 256)     {
            peekc = 0; printf("Maxwidth ?  ");     }
        printf(" %d !\n",maxw);
        peekc = 0;
        printf("logical height above baseline ?  ");
        while((lht=getnum()) < 0 || lht > ht)     {
            peekc = 0; printf("lht ?  ");     }
        printf(" %d !\n",lht);
        peekc = 0;
        printf("Type in any one-line");
        printf(" font description, if desired.\n");
        getname(des);
    }
    max = 32677; wrflag = 0;
    head->code = max;
    head->next = 0; chmod = 0;
    include = 1; freenode = 1;
    infont = 0; wr = 1;
    head = llist; avail = &llist[1];
}

zhdr(h)  //zero a hdr table
    int h[];     {
    register int i;
    n = h;
    for(i=0;i<256;i++) *n++;
}

int getc() {//return next char in command line
    if (peekc)     {
        c = peekc;
        peekc = 0;
    }
    else    {
        c = getchar();
        if (c != ' ') peekc = c;
    }
    return(c);
}
```

```
fonthdr() {//read hdr table and font dimensions
    int i; char t;
    read(readfp,hdr,512);
    read(readfp,&ht,2);
    printf("\nHeight %d  ",ht);
    if (ht > 120 || ht < 0)    {
        printf("too high"); exit(); }
    read(readfp,&maxw,2);
    printf("Maximum character width %d  ",maxw);
    if(maxw > 256 || maxw < 0)     {
        printf("too wide"); exit();}
    read(readfp,&lht,2);
    printf("Logical height %d\n",lht);
    if(lht > ht || lht < 0)    {
        printf("too high"); exit();}
    seek(readfp,518,0);
    p = des; t = 1;
    for(i=0; t != '\0';i++)    {
        read(readfp,&t,1);
        *p++ = t;
    }
}

int getnum() {//convert numeric string and rtrn value
    int i,base;
    i = 0;
    while((c = getc()) == ' ')    ;
    if (c >= '0' && c <= '9')    {
        base = (c-'0') ? 10 : 8;
        peekc = c;
        if (base == 10) while((c=getc()) >='0' && c<='9')  {
            peekc = 0;
            i = i*base + c - '0';
        }
        else while((c=getc()) >='0' && c<='7')   {
            peekc = 0;
            i = i*base + c - '0';
        }
        peekc = c;
        return(i);
    }
    else{//there was no numeric string
        peekc = 0;
        if (c == '+') return(-2);
        if (c == '-') return(-3);
        peekc = c; //c will be processed later
        return(-1);
    }
}

int command()    {
    /* Process the command line:
            update infont
            check command arguments
            execute command
```

```
   Any problems ? return a 1; otherwise, return a 0 */
register i,j;
int temp, k, h, hb, lb;
switch(temp = getnum())     {

   case -2:   //increment infont
      if (chmod) putdef();
      infont++;
      in = 0; chmod = 0;
      break;

   case -3:   //decrement infont
      if (chmod) putdef();
      infont--;
      in = 0; chmoc = 0;
      break;

   case -1: break;   //no change

   default:   //infont gets temp
      if (chmod) putdef();
      infont = temp;
      in = 0; chmod = 0;
      break;

}
if (infont < 0) infont = 127;    //check for wraparound
if (infont > 127) infont = 0;
while((c = getc()) == ' ')    ;
switch (c)    {

  case 'a':   //add a character
    instr(); c=getchar(); getdim(); p=tbuf;
    for(i=0;i<4000;i++) *p++ = 0 ;
     bytes = (rw%8 == 0) ? rw/8 : rw/8 + 1;
    in++; wrflag++; chmod++; break;

   case 'c':   //change lines s thru e
      if (gchardef(readfp))     {
         if (setse(ht)) error;
         sbase();
         for(i=s; i < e;i++)
            for(j=first; j < last+first; j++)
               tbuf[i*bytes+j] = 0;
         for(i=s; i <= e; i++)     {
            printf("%3d ",i);
            for(j=first; j < last+first;j++)
               tbuf[i*bytes+j] = getdef();
         }
         in++; cstat = 'm';
         wrflag++; chmod++;
      }
      else error;
      break;
```

```
case 'd':   //delete char's s thru e
    if (setse(128)) error;
    cstat = 'd';
    for(infont=s; infont<=e;infont++)     {
        if(hdr[infont*2] == 0) continue;
        hdr[infont*2] = 0; putdef();
    }
    in = 0; wrflag++; break;

case 'e':   //edit lines s thru e
    if(gchardef(readfp))     {
        if(setse(ht)) error;
        sbase();
        gtty(1,sgtty); savetty = sgtty[1];
        for(i=s; i<=e;i++)     {
            printf("\n%3d ",i);
            sgtty[1] =| 03; stty(1,sgtty);
            for(j=first;j<first+last;j++)
                list("%c%c%c%c%c%c%c%c",tbuf[i*bytes+j]);
            sgtty[1] = savetty; stty(1,sgtty);
            printf("\n    ");
            for(j=first;j<first+last;j++)
                tbuf[i*bytes+j] = getdef();
        } in++; wrflag++; chmod++; cstat = 'm';
    } else error; break;

case 'f':   //switch char dimension flag
    dim =(dim) ? 0 : 1 ;
    break;

case 'i':  //include char's s thru e from rfontfile
    if (setse(128)) error;
    getname(rfontfile);
    ppend(rfontfile,"/.fonts.01/font/");
    if((temp=open(rfontfile,0)) < 0)     {
        printf("cannot open %s",rfontfile); error;
    }
    cpy(hdr,fhdr); read(temp,hdr,512);
    read(temp,&tht,2); read(temp,&tmaxw,2);
    read(temp,&tlht,2);
    if (reject())     {
        printf("compatible ");
        cpy(fhdr,hdr); error;
    }
    in = include = 0;
    cstat = 'i'; wr = 0; drc = 1;
    for(infont=s; infont<=e; infont++)     {
        if (gchardef(temp)) putdef();
        else if(drc == 0) putdef();
    }
    close(temp); wr = 1;
    for(i=0;i<s;i++)     {
        hdr[i*2] = fhdr[i*2]; hdr[i*2+1] = fhdr[i*2+1];
    }
    for(i=e+1;i<128;i++)     {
```

```
            hdr[i*2] = fhdr[i*2]; hdr[i*2+1] = fhdr[i*2+1];
        }
        include = 1; wrflag++; break;

    case 'l':   //list lines s thru e
        if (gchardef(readfp))    {
            if (setse(ht)) error;
            sbase();
            for(i=s; i <= e;i++)     {
                printf("\n%3d ",i);
                for(j=first;j < last + first; j++)
                    list("%c%c%c%c%c%c%c%c",tbuf[i*bytes+j]);
            }
            in++;
        }
        else error;
        break;

    case 'n':   //display font description and table
        p = des;
        if(*p == '\0') printf("no description\n");
        else for(i=0;*p != '\0'; i++)
          putchar(*p++);
        putchar('\n');
        printf("     0    1    2    3    4");
        printf("    5    6    7");
        for(i=0; i<128;i++)     {
            if(i%8 == 0)     {
                if (i == 0)printf("\n000");
                else if (i < 0100)printf("\n0%o",i);
                else printf("\n%o",i);
            }
        pstat(i);
        }
        printf("\n\n' ' undefined  'X' unmodified  ");
        printf("'I' included  ");
        printf("'D' deleted  'M' modified");
        break;

    case 'p':   //modify font/char dimensions
        instr(); c = getchar();
        getdim(); break;

    case 'q':   //quit, warn if not written
        if (wrflag)    {
            wrflag = 0;
            printf("write??");
            error;
        }
        exit();


    case 's':   //shift lines s thru e once
        if(gchardef(readfp))    {
            peekc=0; temp=getc();
```

```
if (setse(ht))  error;
sbase();
switch (temp)    {

   case 'r':   //right
    for(i=s; i<=e; i++)    {
       lb = 0;
       for(j=first; j < first+last; j++)    {
          hb = lb; p = &tbuf[i*bytes+j];
          if (*p & 01) lb = 1; else lb = 0;
          *p =>> 1;
          if(hb) *p =| 0200;else *p =& 0177;
       }
    } break;

   case 'l':   //left
    for(i=s;i<=e;i++)    {
       hb = 0; lb = 0;
       for(j=first+last-1;j>=first;j--)    {
          p = &tbuf[i*bytes+j];
          if((*p&0200)>>7) hb = 1; else hb = 0;
          *p =<< 1; if(lb) *p =| 01; lb = hb;
       }
     } break;

    case 'u':   //up
     for(i=s; i<=e; i++)    {
        if(i == 0) sontinue;
        for(j=first; j<first+last;j++)
          tbuf[(i-1)*bytes+j] = tbuf[i*bytes+j];
      }
     for(j=first; j<first+last;j++)
        tbuf[e*bytes+j] = 0;
     break;

   case 'd':   //down
     for(i=e;i>=s;i--)   {
       if (i == ht-1 ) continue;
       for(j=first;j<first+last;j++)
         tbuf[(i+1)*bytes+j] = tbuf[i*bytes+j] ;
       }
     for(j=first;j<first+last;j++)
       tbuf[s*bytes+j] = 0 ;
     break;


   default: error;

}  //list the shift
for(i=s; i <= e; i++)    {
   printf("\n%3d ",i);
   for(j=first;j < first+last; j++)
      list("%c%c%c%c%c%c%c%c",tbuf[i*bytes+j]);
}
in++; wrflag++; chmod++; cstat = 'm';
```

```
    } else error; break;

case 'w':    //write to wfontfile and quit
    if (chmod) putdef(); wr = 0;
    getname(wfontfile);
    //no writing to file being edited
   if ( cmpr(wfontfile,sfontfile) ||
        cmpr(wfontfile,"HFONT") )   {
      printf("writing to existing file "); wr=1; error;
   }
    if((writefp=creat(wfontfile,0666)) < 0)     {
       printf("file "); error;
    }
    zhdr(fhdr);
    write(writefp,fhdr,512); //write blank hdr table
    write(writefp,&ht,2);
    write(writefp,&maxw,2);
    write(writefp,&lht,2);
    blkc = 1; bytc = 6; p = des;
    for(i=0; *p != '\0';i++)     {
       write(writefp,p++,1); bump(1);
    }
    write(writefp,p,1); bump(1); in = 0;
    for(infont=0; infont< 128; infont++)     {
       if (hdr[infont*2] == 0) continue; //no char here
       else if (find(infont)) {//get it from llist
          if (current->nsize == 0) continue;
          fhdr[infont*2]=(hdr[infont*2]&0377)|(blkc<<8);
          fhdr[infont*2+1] = bytc;
          write(writefp,current->def,current->nsize);
          bump(current->nsize);
          free(current->def);
       }
       else if (edit) {//get it from file
          i = gchardef(readfp);
          p = tbuf;
          fhdr[infont*2]=(hdr[infont*2]&0377)|(blkc<<8);
          fhdr[infont*2+1] = bytc;
          write(writefp,p,8); bump(8);
          p =+ bytes*rft + 8;
          write(writefp,p,bytes*drc);
          bump(bytes*drc);
       }
      else error;
    }
    seek(writefp,0,0);
    write(writefp,fhdr,512);
    delete = 1;
    //remove any empty fontfile
    for(i=0;i<256;i=+ 2) if(fhdr[i] > 0) delete = 0;
    if (delete){blkc = bytc = 0; unlink(wfontfile);}
    printf("%l\n",blkc*512+bytc);
    exit();
```

```
        case '\n': break;          // sync

        default:
           printf("%c ",c);
           error;

    }
    return(0);
}

bump(i) //running count wfontfile size
         //in blocks and bytes
    int i;     {
    if (bytc+i >= 512)     {
        if ((blkc + (bytc+i)/512) < 255)     {
           blkc =+ (bytc+i)/512;
           bytc = (bytc+i)%512;
        }
        else if (bytc+i > 32768)   {
           printf("file too big"); exit();
        }
        else bytc =+ i;
    }
    else bytc =+ i;
}

int cmpr(p1,p2)  //rtn 1 if p1 != p2; otherwise, 0
    char *p1,*p2;     {
    for( ; ; )     {
        if (*p1 != *p2++) return(0);
        if (*p1++ == '\0') return(1);
    }
}

cpy(n1,n2)  //copy p1 to p2
    int *n1,*n2;     {
    int i;
    for(i=0;i<256;i++) *n2++ = *n1++;
}

ppend(p1,p2)  //prepend p2 to p1
    char p1[], p2[];     {
    char *b1, *b2, t[40];
    b1 = p1; b2 = t;
    while((*b2++ = *b1++) != '\0') ;
    b2 = p2; b1 = p1;
    while((*b1++ = *b2++) != '\0') ;
    b2 = t; b1--;
    while((*b1++ = *b2++) != '\0') ;
}

int reject() {  //rtn 1 if files are incompatible;ow, 0
    if(tht != ht || tlht != lht || tmaxw > maxw) return(1);
    else return(0);
}
```

```
onintr() {  //restore environ. reset int trap
    signal(2,onintr);
    if (savetty)    {
        sgtty[1] = savetty;
        savetty = 0;
        stty(1,sgtty);
        savetty = 0;
    }
    reset();
}

int gchardef(fp)
    /* Get the character definition for the current
    character, put it in the char buffer, expand
    blank rows, and display necessary diagnostics */
    int  fp;    {
    register i,j;
    register char *tp;
    if (in) return(1);  //it's already there, rtn 1
    if (find(infont) && include) { //it's on the llist
        if (current->stat == 'd')    {
            printf("deleted ");
            return(0);
        }
        tp = tbuf;
        chardef  = current->def;
        *tp++ = rw = *chardef++; rw =& 0377;
        rw =| (*tp++ = *chardef++) << 8;
        if (rw <= 0)    {
            printf("%o raster width %d ",infont,rw); return(0);
        }
        bytes = (rw%8 == 0) ? rw/8 : rw/8 +1;
        *tp++ = lk = *chardef++; lk =& 0377;
        lk =| (*tp++ = *chardef++) << 8;
        *tp++ = rft = *chardef++; rft =& 0377;
        rft =| (*tp++ = *chardef++) << 8;
        *tp++ = drc = *chardef++; drc =& 0377;
        drc =| (*tp++ = *chardef++) << 8;
        if(drc == 0)    {
            printf("printable ");
            return(0);
        }
        bot = ht - (drc + rft);
        for(i=0; i < rft; i++)
            for(j=0; j < bytes; j++) *tp++ = 0;
        for(i=0; i < drc; i++)
            for(j=0; j < bytes; j++) *tp++ = *chardef++;
        for(i=0; i < bot; i++)
            for(j=0; j < bytes; j++) *tp++ = 0;
        if (wr && dim) pchardim();
        return(1);
    }
    //get it from the file
    if (hdr[infont*2] == 0)    {
        printf("undefined "); return(0);
```

```
    }
    if ((j= (hdr[infont*2] & 0177400) >> 8) != 0)     {
        j =& 0377;
        seek(fp,j,3);
        seek(fp,hdr[infont*2+1],1);
    }
    else seek(fp,hdr[infont*2+1],0);
    read(fp,&rw,2);
    if (rw <= 0)     {
        printf("%o raster width %d ",infont,rw); return(0);
    }
    read(fp,&lk,2);
    read(fp,&rft,2);
    read(fp,&drc,2);
    if (drc == 0 && wr)     {
        printf("printable ");
        return(0);
    }
    bot = ht -(drc + rft);
    bytes = (rw%8 == 0) ? rw/8 : rw/8 + 1;
    tp = tbuf;
    *tp++ = rw & 0377;
    *tp++ = (rw & 0177400) >> 8;
    *tp++ = lk & 0377; *tp++ = (lk & 0177400) >> 8;
    *tp++ = rft & 0377; *tp++ = (rft & 0177400) >> 8;
    *tp++ = drc & 0377; *tp++ = (drc & 0177400) >> 8;
    for(i=0; i < rft; i++)
        for(j=0; j < bytes; j++) *tp++ = 0;
    for(i=0; i < drc; i++)     {
        read(fp,ibuf, bytes);
        for(j=0; j < bytes; j++) *tp++ = ibuf[j];
    }
    for(i=0; i < bot; i++)
        for(j=0; j < bytes; j++) *tp++ = 0;
    if (wr && dim) pchardim();
    return(1);
}

int setse(x)  //set command args s and e
    int x;     {
    peekc = 0;
    s = getnum();
    if (s < 0)     {
        s = 0; e = x-1;
        return(0);
    }
    e = getnum();
    if (e < 0) e = s;
    if (e < s) error;
    if((s >= x || e >= x) && x == 128) error;
    if((s > x || e > x) && x == ht) error;
    return(0);
}

list(fmt,byt)
```

```
//list byte, bit by bit, 0=>'.', 1=>'0'
   char *fmt, byt;     {
   printf(fmt,0200&byt?'0':'.',0100&byt?'0':'.',
              0040&byt?'0':'.',0020&byt?'0':'.',
              0010&byt?'0':'.',0004&byt?'0':'.',
              0002&byt?'0':'.',0001&byt?'0':'.');
}

int find(i)
//if current character is on llist, rtn 1 and
//current points to correct node; ow, rtn 0
   int i;     {
   register struct node *ptr;
   ptr = head;
   while (i > ptr->code )
      ptr = ptr->next;
   if (i == ptr->code)     {
      current = ptr;
      return(1);
   }
   else return(0);
}

getname(file)
//get name ending in '\0' and stick it in file
   char file[];     {
   while((c = getc()) == ' ')     ;
   if(c != '\n')     {
      p = file;
      do   {
         *p++ = c; peekc = 0;
      }   while((c = getc()) != '\n');
      *p = '\0';
   }
}

putdef()     {
//put definition in char buffer on llist
   if (find(infont)) lnode(current,infont);
   else     {
      lnode(insert(avail,infont),infont);
      if (freenode > 128)   {
         printf("overflow"); exit();
      }
      avail = &llist[++freenode];
   }
}

lnode(ptr,k)  //do the work for PUTDEF
   struct node *ptr; int k;     {
   register int i,j;register char *tp;
   int clear;
   ptr->code = k;
   if (cstat == 'd')     {
      ptr->stat = cstat;
```

```
        return;
    }  //count blank rows at top and bottom
    rft = bot = 0;
    i = 0; clear = 1;
    while(i < ht && clear)     {
        for(j=8; j < bytes + 8; j++)
            if (tbuf[i*bytes+j] != 0) clear = '\0';
        if (clear) rft = i+1;
        i++;
    }
    if (i < ht)     {
        i = ht-1; clear = 1;
        while(i > 0 && clear)     {
            for(j=8; j < bytes + 8; j++)
                if (tbuf[i*bytes+j] != 0) clear = '\0';
            if (clear) bot = ht-i;
            i--;
        }
    }
    drc = (drc) ? ht -(rft+bot) : 0;
    if(drc == 0) rft = lk = 0;
    tp  = ptr->def = alloc(bytes*drc+8);
    *tp++ = rw & 0377; *tp++ = (rw & 0177400) >> 8;
    *tp++ = lk & 0377; *tp++ = (lk & 0177400) >> 8;
    *tp++ = rft & 0377; *tp++ = (rft & 0177400) >> 8;
    *tp++ = drc & 0377; *tp++ = (drc & 0177400) >> 8;
    for(i=rft; i < rft+drc;i++)     {
        for(j=8; j < bytes + 8; j++)
            *tp++ = tbuf[i*bytes+j];
    }
    ptr->nsize = 8+drc*bytes;
    ptr->stat = cstat;
}

struct node *insert(a,i)
//rtn a node for PUTDEF to use
    struct node *a; int i;     {
    register struct node *ptr,*temp;
    temp = ptr = head;
    while( i > ptr->code )     {
        temp = ptr;
        ptr = ptr->next;
    }
    if (ptr == head)     {
        a->next = head;
        head = a;
    }
    else     {
        a->next = temp->next;
        temp->next = a;
    }
    a->stat = a->def = a->nsize = 0;
    return(a);
}
```

```
sbase() { //set horizontal starting point for char def
   first = 8; last = bytes;  //normal char, default
   if (bytes > 9) { //too wide, get a starting pt
      printf("\ntoo wide...starting where ?");
      peekc = 0;
      while((last = getnum()) < 0 || last >= rw)     {
         peekc = 0; printf("where ?");     }
      peekc = 0;
      last = (last == 0) ? 1 :  last/8 + 1;
      first = first + last-1;
      last = ((bytes+8-first) > 9) ? 9 : bytes+8-first;
   }
}

getdef() {  //get one byte of a definition
   int mask,i,j;
   peekc = 0;
   while((c = getc()) != '0' && c != '.')     ;
   peekc = c;
   i = j = 0;
   mask = 0400;
   while((j++ < 8) && ((c=getc()) == '0' || c == '.'))   {
      peekc = 0;
      if ((mask = mask>>1) && c == '0')
         i =| mask;
   }
   return(i);
}

pstat(i)  //print char status for edit table
   int i;    {
   if (find(i))    {
      switch(current->stat)     {

         case 'd': printf("  D  ");break;
         case 'i': printf("  I  ");break;
         case 'm': printf("  M  ");break;

      }
   }
   else if (hdr[i*2] == 0) printf("     ");
   else printf("  X  ");
}

pchardim() {  //display char dimensions
   int i;
   if((i = hdr[infont*2] & 0377) == 0)  {
      printf("undefined"); return;
   }
   printf("rw %d  cw %d  ",rw,i);
   if (rw == i) printf("lk %d  rk %d",lk,lk);
   else if (lk)    {
      if (lk+i == rw)printf("lk %d  rk %d",lk,0);
      else printf("lk %d  rk %d",lk,rw -i-lk);
   }
```

```
	else printf("lk %d  rk %d",lk,rw-i);
	printf("  ht %d  lht %d  ",ht,lht);
	printf("rft %d  drc %d\n",rft,drc);
}

getdim()	{
	/* Look for a number and/or name. Take both as
	a request, rejecting invalid requests with a '?'
	Quit on 't' and return to the main command loop */
	int i,j, font; char name[20];
	j = hdr[infont*2]&0377; font = 0;
	while (1)	{
		peekc = 0; printf("\n%3o-> ",infont);
		i = getnum(); getname(name);
		if(cmpr(name,"t")) break;
		if(cmpr(name,"i")) instr();
		else if(cmpr(name,"infont"))	{
			infont = i; i = gchardef(readfp);
		} else if(cmpr(name,"d"))	{
			printf("%s\n",des);
			peekc = 0; getname(des);
		} else if(cmpr(name,"p")) pchardim();
		else if(cmpr(name,"f"))
			printf("ht %o maxw %d lht %d\n",ht,maxw,lht);
		else if(cmpr(name,"ht"))	{
			if(i >= lht){ ht = i; wrflag++; }
			else printf("\n? ");
		} else if(cmpr(name,"lht"))	{
			if(i <= ht){ lht = i; wrflag++; }
			else printf("\n? ");
		} else if(cmpr(name,"maxw"))	{
			if(i < 0 || i > 256) {maxw = i; wrflag++; }
			else printf("\n? ");
		} else if(cmpr(name,"cw"))	{
			if(gchardef(readfp))	{
				if(i <= rw)	{
					hdr[infont*2] =& 0177400;
					hdr[infont*2] =| i & 0377;
					lk = rw-i; font = 1;
				} else printf("\n? ");
			} else printf(" cw now %d\n",(hdr[infont]=i));
		} else if(cmpr(name,"rw"))	{
			if (gchardef(readfp))	{
				if(i <= maxw)	{
					rw = i; font = 1;
					if(rw < j)	{
						hdr[infont*2] =& 0177400;
						hdr[infont*2] =| i & 0377;
						lk  = 0; font = 1;
					}
				} else printf("\n? ");
			} else printf(" rw now %d\n",(rw = i));
		} else if(cmpr(name,"lk"))	{
			if(gchardef(readfp))	{
				if(rw == j)	{
```

```
                if(i == 0) {lk = i; font = 1; }
                else printf("\n? ");
            } else if(i <= rw-j) { lk = i; font = 1; }
            else printf("\n? ");
        } else printf(" lk now %d\n",(lk = i));
    } else if(cmpr(name,"rk"))     {
        if(gchardef(readfp))     {
            if(rw == j)     {
                if (i == 0) ;  else printf("\n? ");
            } else if(i <= rw-j)     {
                if(i+lk == rw-j) ;
                else { lk = rw-i; font = 1; }
            } else printf("\n? ");
        } else printf("\n? ");
    } else printf("\n? ");
  }
  if (font)     {
    wrflag++;cstat = 'm';outdef();in = 0;
  }
}

instr() {  //display instructions for GETDIM
   printf("Modifiable FONT dimensions are:\n");
   printf("height- 'ht'  max character width- 'maxw'");
   printf("  logical height- 'lht'\n\n");
   printf("Modifiable CHARACTER dimensions are:\n");
   printf("raster width- 'rw'  character width- 'cw'");
   printf("  left kern- 'lk'  right kern 'rk'\n\n");
   printf("Type 'i' for instructions, 'p' for ");
   printf("dimensions of character in buffer.\n");
   printf("To move to another character, update ");
   printf("'infont'.\n");
   printf("\nGet font dimensions with 'f'. ");
   printf("Modify font name with 'd'. If you're adding");
   printf("a\n character, make changes in this order only:");
   printf(" 'rw', 'lk' , then 'cw'.\n");
   printf("\nImpossible modifications are rejected....");
   printf("some example inputs might be\n");
   printf(" '22 lht', '063 infont', 'i', or '0 lk'\n\n");
   printf("You'll be prompted with a '->'. ");
   printf("When you are finished, type 't'... \n\n");
}
```

APPENDIX B. VECTOR TO RASTER CONVERSION

A. 'MAKEHF': CREATING A FONT

1. Basic Structure

"Makehf" is designed to convert the vector definitions of Hershey's 14 fonts into a digitized form suitable for use in computer typesetting. The digitized font file created matches the format used in the SAIL files and is compatible with the font editor. This font file format is described in Chapter III; Hershey font files differ slightly because no font description is ever generated by "makehf", so an extra zero byte immediately follows the three words containing the font height, maximum character width, and logical height. This zero-word tells "edf" that no description is available.

All fonts digitized from Hershey vector definitions are variable width fonts. The arguments used to call "makehf" are described in Chapter II and again in this Appendix.

This program can be used in a stand-alone mode, in which case the digitized font file created is normally left on file "/.fonts.01/HFONT" and it can then be copied to any other directory. The digitized font may be written directly to another file, as explained in the next paragraph. When

digitizing a font using "edf", "makehf" is spawned as a child process; the editor waits until the digitization is complete and then opens file "/.fonts.01/HFONT" for reading and continues normally.

An additional option has been added and is normally used for the digitization of fonts larger than 42 point. However, it may be used whenever the user wants the digitized file written to some location other than the default file. Since the file space available on the mountable file is limited, the user may include a full path name as a third argument and indicate a specific output file as the destination for the digitized font. This option should permit the user to avoid system "write" errors that might occur if the digitized file were larger than the space available on "/.fonts.01". A point size must be included as the second argument when using this option, even if the default value is desired.

2. Limitations

Because a decision was made to limit the maximum raster height to 255 pixels, "makehf" will create fonts only up to 91 point in size. The user should also be aware that font files increase in size rapidly as larger point sizes are requested. The next pages, for example, contain the Duplex Roman font digitized to 10, 20, 30, and 40 point sizes. This allows a comparison of the relative sizes of both the characters and the font files themselves. The file

sizes and the times required to create the fonts are listed below:

	SIZE(bytes)	TIME		
		Real	User	System
HDR10	5522	0:45.0	0:22.8	0:19.4
HDR20	15211	1:41.0	1:10.4	0:25.7
HDR30	31757	3:00.0	2:26.4	0:28.8
HDR40	53458	5:18.0	4:11.6	0:49.6

The three times given were obtained using the "time" command discussed in Reference 5. The conversion was made using "edf" with a "w" (write) command waiting for the editor when it returned from the call to "makehf". Normally, the time required to digitize a font will increase noticeably as either the point size desired or the number of lines per character (the complexity) increases. These times were taken early in the evening , and are somewhat faster for the larger fonts than a normal time during the work day would be. If both factors increase, then the time required for digitization becomes noticeably longer. A comparison of the times required to digitize each of the fonts at 20 point is given below:

	REAL	USER	SYSTEM	SIZE(bytes)
HSR	1:01.0	0:35.8	0:20.4	15461
HDR	1:41.0	1:10.3	0:25.3	15211
HCR	1:51.0	1:12.1	0:28.6	18327
HTR	2:19.0	1:35.1	0:30.8	15856
HCI	2:16.0	1:20.9	0:30.8	18936

/.doyle.01/hftemp/HDR10

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
wxyz

/.doyle.01/hftemp/HDR20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKLMNO
PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

/.doyle.01/hftemp/HDR30

!"$&'()*+,-./0123456789:;=?AB
CDEFGHIJKLMNOPQRSTUVWXYZabcd
efghijklmnopqrstuvwxyz

FIGURE B-1. Increasing Point Sizes--HDR

/.doyle.01/hftemp/HDR40

!"$&'()*+,-./012345678
9:;=?ABCDEFGHIJKLMNOP
QRSTUVWXYZabcdefghijkl
mnopqrstuvwxyz

FIGURE B-1. (Continued)

HTI	3:16.0	1:46.6	0:33.5	17063
HSS	1:16.0	0:44.7	0:22.6	13393
HCS	1:46.0	1:02.5	0:24.2	13173
HSG	1:13.0	0:37.9	0:21.5	15302
HCG	2:13.0	1:12.8	0:29.4	18086
HGE	3:44.0	1:47.7	0:35.0	16201
HGG	2:42.0	1:54.7	0:35.0	17560
HGI	2:33.0	1:37.1	0:32.0	16759
HCC	2:06.0	1:28.3	0:29.8	19802

"Makehf" can address up to 200K in memory, which permits the digitization of fonts up to approximately 91 point in size. However, the font output routines can address only 160K, so this limits the size of a digitized font that can be addressed in its entirety to approximately 65 point. Larger fonts can be digitized if only upper case letters, digits, and punctuation are desired or required; some lower case letters may be available (run "prfont" and see what it prints), but some will be unaddressable.

NAME

makehf -- digitize a Hershey font from the vector definition

SYNOPSIS

makehf -HFT [SIZE] [output file]

DESCRIPTION

This command creates a Hershey font in the point size requested by the user. SIZE is an optional parameter; if no SIZE option is used, the font will be created in the default size -- 10 point. The maximum height allowed is 255 pixels (91 point).

A full path name may be used as a third argument to "makehf". This causes the program to write the digitized font to the specified output file rather than to the default file, "/.fonts.01/HFONT". The use of this option is recommended at point sizes larger than 40-42 point. The SIZE must be specified if this option is used, even if the default size is desired.

The font requested by HFT must come from the following list:

HSR -- Simplex Roman
HDR -- Duplex Roman
HCR -- Complex Roman
HTR -- Triplex Roman
HCI -- Complex Italic
HTI -- Triplex Italic
HSS -- Simplex Script
HCS -- Complex Script
HSG -- Simplex Greek
HCG -- Complex Greek
HGE -- Gothic English
HGG -- Gothic German
HGI -- Gothic Italian
HCC -- Complex Cyrillic

FILES

/.fonts.01/hershey/--.v
/.fonts.01/HFONT

SEE ALSO

edf

BUGS

This program will actually convert a Hershey vector definition to a digitization in a stand-alone mode as long as the first, third, and fourth letters in the first argument are correct, i.e., the second character need not be an "H". However, this is the same program called by the font editor to create a Hershey font, and if the argument oassed to "edf" does not begin with a "-H", the editor will not work.

```
            /*                        */

          /*        makehf.c        */

            /*                        */

#define         FFACTOR     .050
#define         SIZE        8192
#define         MODE        0666
#define         HELP        1.75
#define         NOHELP      .40
#define         SOMEHELP    .90
#define         DELV        .60
#define         DELH        .45
#define         DOLLAR      73    // for height
#define         SLASH       95    // also for height
#define         CAPM        155 // capital M location
                            // in directory
#define         SMALLP      225 // small p location in
                             // directory
#define         CAPX        177  // cap x for Greek chars
#define         SMALLX      241  // small x for HCG
#define         CAPP        81  // cap p for HGG
#define         SMALLF      103 // small f for HGG
#define         STDFONT     29.0 // height
#define         R           0
#define         W           1

float xj, yi, xx, yy,
     xl, yl, xr, yr,
     lx, ly, rx, ry,
      b, m, deltax, xwide, yhigh,
     true, test, xconst, yconst, delx;

int i, j, k,                    // counters for "for" loops
   ftht, ftw,                   // font height, width
   rptr, wptr,                  // read and write pointers
   ymin, ymax,                  // min & max height of font
   xmin, xmax,                  // min & max widths of font
   ktr,                         // counter for arrays
   bytes,                       // byte counter
   cptr;

int M[259] ;                    // array for char directory

int hts[4];
long int cpos ;

char rpath[]  {
       "/.fonts.01/hershey/--.v"
       }  ;
char wpath[]  {
       "/.fonts.01/HFONT"
       }  ;
```

```
char lep[2],
X[144], Y[144],
mask,
num1, num2,
    go, flag, ok;

char DM[SIZE] ;

readch()   {                              // read in a char def
   k = seek( rptr, cptr, 0 ) ;
   ktr = 0 ;
   num1 = num2 = 0 ;
   while ( (num1 != 50) || (num2 != 50) )   {
       k = read( rptr, lep, 2 ) ;
       num1 = lep[1] ;
       num2 = lep[0] ;
       if ( num1 > 50 ) num1 = num1 - 100 ;
       if ( num2 > 50 ) num2 = num2 - 100 ;
       X[ktr] = num1 ;
       Y[ktr] = num2 ;
       ktr++ ;
       }
}                                             // end readch() ...

minmax()   {
   ktr = 0 ;
   while ( (X[ktr] != 50) || (Y[ktr] != 50) )   {
       if ( (X[ktr] > xmax) && (X[ktr] != 50) )
           xmax = X[ktr] ;
       else if ( X[ktr] < xmin )  xmin = X[ktr] ;
       if ( (Y[ktr] > ymax) && (Y[ktr] != 50) )
           ymax = Y[ktr] ;
       else if ( Y[ktr] < ymin ) ymin = Y[ktr] ;
       ktr++ ;
       }
}                                    // end minmax() ...

main( argc, argv )
int argc ;
char *argv[] ;   {
int ii, jj,                                  // array counters
   c, cw, rw, ptsize,              // font parameters
   trow, lrow, rows, rowc, maxcw,
                                   // row pointers
   top, bot,                       // top & bottom of char
   high,                           // how high is the char ?
   lht,                            // logical height of char
   maxaddr,                        // highest addr in 16 bits
   rem, block,                     // if font > 65K
   drc, rft;                       // data row count,
                                   // rows from top
int thdr[259] ;                    // temp header for output file

char slope, posit,
    strt,
```

```
      zero,
      big,                              // if font is > 65K
     greek,                             // flag for Greek alphabet
      gothger;                          // flag for Gothic German

if ( argc >= 3 )  {
   if ( (argv[2][0] < '0') || (argv[2][0] > '9') ) {
      printf("incorrect argument--\n");
      printf("point size not given...\n");
      exit();
      }
  else  ptsize = atoi( argv[2] ) ;
  }
else  {
  ptsize = 10 ;
  }
rpath[19] = argv[1][2] ;
rpath[20] = argv[1][3] ;
if ( argv[1][3] == 'G' )  {
   if ( argv[1][2] == 'G' )  {
      gothger = 1;  greek = 0;  }
   else { greek = 1;  gothger = 0;  }
   }
else  { greek = gothger = 0;  }
ftht = ( ptsize * 2.8 ) + 1 ;
delx = FFACTOR ;
deltax = ftht * delx ;
block = rem = 0 ;
maxcw = 0 ;
zero = 0 ;
mask = 01 ;
ymin = ymax = 0 ;
for ( i = 0 ; i < 259 ; i++ )
   thdr[i] = 0 ;
for ( i = 0 ; i < SIZE ; i++ )
   DM[i] = 0 ;

rptr = open( rpath, R ) ;
if ( argc == 4 )  {
    if ( argv[3][0] != '/' )  {
       printf("incorrect output file name--\n");
       printf("full path name required\n");
       exit();
       }
    else  wptr = creat( argv[3], MODE );
    }
else  wptr = creat( wpath, MODE );
k = read( rptr, M, 512 ) ;
if ( k != 512 )  {
  printf("incorrect read from %s", rpath ) ;
  exit();
  }
k = seek( wptr,0,0 ) ;
k = write( wptr,thdr,518 ) ;
k = write( wptr,&zero,1 ) ;
```

```
cpos = 519 ;
if ( ptsize >= 36 )  {
   block = cpos>>9 & 0177777 ;
   rem = cpos & 0777;
   big = 1 ;
   }
else big = 0 ;

   // use a capital M and a small p or x to find
   // the highest and lowest points in the font

hts[2] = DOLLAR ;
hts[3] = SLASH ;
if ( greek )  {
   hts[0] = CAPX ;
   hts[1] = SMALLX ;
   }
else if ( gothger )  {
    hts[0] = CAPP ;
    hts[1] = SMALLF ;
    }
  else  {
   hts[0] = CAPM ;
   hts[1] = SMALLP ;
   }
for ( i = 0 ; i < 4 ; i++ )  {
   cptr = M[ hts[i] ]<<1 ;
   readch() ;
   minmax() ;
   if ( i == 0 ) lht = ymax ;
   }

   // now use the high and low points to find
   // the multiplication constants necessary
   // to make the standard font larger or smaller

yhigh = ymax - ymin ;
yconst = ftht / yhigh ;
xconst = ftht / STDFONT ;
xwide = ( xmax - xmin ) * xconst ;
yy = (ymin * yconst) ;
lht = (lht * yconst) - (ymin * yconst) ;


   // now walk through the directory and convert
   // any non-zero entries (i.e., characters) from
   // vector to raster in the desired pointsize

for ( i = 0 ; i < 128 ; i++ )  {// controlling loop...
  j = i<<1 ;

  if ( M[j] != 0 )  {       // don't bother for nothing...
   cw = M[j] ;
   ftw = rw = cw * xconst ;
   cptr = M[j+1]<<1 ;
```

```
if ( (rw % 8) == 0 )  rows = rw / 8 ;
    else rows = (rw / 8) + 1 ;
ymin = ymax = 0 ;
go = ok = 1 ;
 M[j] = (rw & 0377) | (block<<8 & 0177400) ;
 if ( big )  {
   M[j+1] = rem ;
   }
 else  {
   M[j+1] = cpos ;
   }
lrow = 0 ;
trow = 8 ;
if ( rw > maxcw ) maxcw = rw ;
readch() ;
minmax() ;
top = -( (-ymin * yconst) + yy ) ;
if ( top < 0 ) top = 0 ;
rft = top ;
bot = -( (-ymax * yconst) + yy ) + 1 ;
if ( bot > ftht )  bot = ftht ;
drc = high = ( bot - top ) + 1 ;
ktr = 0 ;
xl = X[ktr] ;
yl = -Y[ktr] ;
++ktr ;
xr = num1 = X[ktr] ;
num2 = Y[ktr] ;
yr = -num2 ;

while ( go )  {      // check each line in the character
  flag = 1 ;
  if ( num1 == 50 )  {// check for "move" or end of char
    if ( num2 == 50 )  go = 0 ;
          // that's all for this char,
          // go on to the next one
    else if ( num2 != 0 )  {
           perror("bad y value for x = 50 ") ;
           break ;
           }
         else  {
           ++ktr ;
           num1 = X[ktr] ;
           num2 = Y[ktr] ;
           if ( num1 > 50 ) num1 = num1 - 100 ;
           if ( num2 > 50 ) num2 = num2 - 100 ;
           xl = num1 ;
           yl = -num2 ;
           flag = 0 ;
           }
    }
  else  {
    if ( yl == yr ) { slope = 0 ; m = 0.0 ; }
    else if ( xr == xl ){slope = -1; m = -1.0;}
      else {slope = 1; m = (yl-yr)/(xl-xr);}
```

```
xx = rw / 2.0 ;
rx = xx + (xr * xconst) ;
ry = yy + (yr * yconst) ;
lx = xx + (xl * xconst) ;
ly = yy + (yl * yconst) ;
ok = 1 ;
rowc = trow ;

if ( slope == 1 )  {          // normal line case
  b = ly - ( m * lx ) ;
  if ( (m >= 3.0) || (m <= -3.0) )
    {
    if ((m >= 7.0) || (m <= -7.0))
        delx = HELP * 1.50 ;
     else delx = HELP;
    }
  else  { if ( (m <= .50) && (m >= -.50) )
        delx = NOHELP ;
       else
        delx = SOMEHELP ;
        }
  if ( m > 0.0 )  { // slope is positive
    for ( ii = top ; ii <= bot+1 ; ii++ ) {
      yi = -ii ;
      if ( ( (yi >= ly) && (yi <= ry) ) ||
            ((yi >= ry) && (yi <= ly) ) ) {
        true = yi ;
        for ( jj = 0; jj < ftw; jj++ ) {
          xj = jj ;
          test = ( m * xj ) + b ;
          if ( (test >= (true - deltax*delx)) &&
               (test <= (true + deltax*delx)) ) {
            c = jj / 8 ;
            posit = jj - ( 8 * c ) + 1 ;
            DM[rowc+c] = DM[rowc+c] |
                                (mask<<(8 - posit));
            }
          }
      }
      rowc = rowc + rows ;
    }
  }
else  {                 // slope is negative
  for ( ii = top ; ii <= bot+1 ; ii++ ) {
    yi = -ii ;
    if ( ( (yi <= ly) && (yi >= ry) ) ||
         ( (yi <= ry) && (yi >= ly) ) )  {
      true = yi ;
      for ( jj = 0; jj < ftw; jj++ ) {
        xj = jj ;
        test = ( m * xj ) + b ;
        if ( (test >= (true - deltax*delx)) &&
             (test <= (true + deltax*delx)) ){
          c = jj / 8 ;
```

```
                posit = jj - ( 8 * c ) + 1 ;
                 DM[rowc+c] = DM[rowc+c] |
                                (mask<< (8-posit)) ;
                }
              }
            }
            rowc = rowc + rows ;
          }
        }
      }
      else if ( slope != 0 )  { // vertical line case
          for (ii = top; ii <= bot+1; ii++ ) {
            yi = -ii ;
            if ( ( (yi < ly) && (yi > ry) )  ||
                 ( (yi < ry) && (yi > ly) )   ) {
              for (jj = 0; jj < ftw; jj++ )  {
                xj = jj ;
                if ( (xj >= (lx - deltax*DELV )) &&
                     (xj <= (lx + deltax*DELV ))) {
                  c = jj / 8 ;
                  posit = jj - (8 * c) + 1 ;
                  DM[rowc + c] = DM[rowc + c] |
                                  (mask<<(8 - posit)) ;
                  }
               }
              }
              rowc = rowc + rows ;
          }
              }
          else  {                       // horizontal line case
              for (ii = top; ii <= bot+1; ii++ )  {
                yi = -ii ;
                if ( (yi <= (ly + deltax*DELH )) &&
                     (yi >= (ly - deltax*DELH ))   ) {
                  for (jj = 0; jj < ftw; jj++ )   {
                    xj = jj ;
                    if ( ( (xj >= lx) && (xj <= rx) ) ||
                          ((xj >= rx) && (xj <= lx) ) ) {
                      c = jj / 8 ;
                      posit = jj - (8 * c) + 1 ;
                      DM[rowc + c] = DM[rowc + c]  |
                                      (mask<<(8 - posit)) ;
                      }
                  }
                  }
                  rowc = rowc + rows ;
              }
              }
}
if ( flag )  {
    xl = xr ;
    yl = yr ;
    }
if ( go )  {
    ++ktr ;
```

```
        num1 = X[ktr] ;
        num2 = Y[ktr] ;
        if ( num1 > 50 ) num1 = num1 - 100 ;
        if ( num2 > 50 ) num2 = num2 - 100 ;
        xr = num1 ;
        yr = -num2 ;
        }
    }
    DM[lrow+1] = ( rw & 0177400 )>>8 ;
    DM[lrow] = rw & 0377 ;
    DM[lrow+2] = DM[lrow+3] = 0 ;
    DM[lrow+5] = ( rft & 0177400 )>>8 ;
    DM[lrow+4] = rft & 0377 ;
    DM[lrow+7] = ( drc & 0177400 )>>8 ;
    DM[lrow+6] = drc & 0377 ;
    lrow = trow + ( high * rows ) ;
    cpos = cpos + lrow ;

    // insert code to handle large fonts, i.e.,
    // fonts that have more than 65535 bytes
    // in the character definitions.  This
    // should happen around the 40-42 point size.

     if ( big )  {
         block = cpos>>9 & 0177777 ;
         rem = cpos & 0777 ;
         if ( block >= 253 ) {
            big = 0 ;
            cpos = rem ;
            }
        }

    k = write( wptr,DM,8 ) ;
    j = trow ;
    if ( high < 25 ) bytes = rows ;
    else bytes = rows * 25 ;
    ktr = bytes ;
    while ( ktr >= bytes )  {
        k = write( wptr,&DM[j],bytes ) ;
        j = j + bytes ;
        ktr = lrow - j ;
        }
    if ( ktr > 0 )
        k = write( wptr,&DM[j],ktr ) ;
    for ( j = 0 ; j < (lrow+ 2*rows) ; j++ )
        DM[j] = 0 ;

   }
}

    //  wrap things up-- put the data for a blank in
    // the SPACE location so that it will plot on
    // the Versatec later...  Then write out the
    // directory and finish up
```

```
for ( k = 1; k < 4; k++)
   thdr[k] = 0;
M[64] = thdr[0] = M[146] & 0377 ;
M[64] = M[64] | (block<<8 & 0177400) ;
if ( big )  {
   M[65] = rem ;
   }
else  M[65] = cpos;
k = write(wptr,&thdr,8) ;
M[257] = maxcw ;
M[256] = ftht ;
M[258] = lht ;
k = seek( wptr,0,0 ) ;
k = write( wptr, &M, 518 ) ;
}
```

APPENDIX C. THE DATA BASE

A. OBTAINING THE DATA BASE

Dr. Allen V. Hershey's complete set of 1377 occidental characters is available in Reference 16, where Appendix A contains the complete vector representation of each character and Appendix B contains a drawing of each character. The vector representation data was also available locally on a tape at the Naval Postgraduate School's W.R. Church Computer Center; that tape, labeled NPS451, provided the data base for this thesis. The data was read from NPS451 onto another tape so that it could be used in the PDP-11/50 environment available in the Computer Science Department's computer laboratory.

Information regarding the IBM system utility program IEBGENER used to read from NPS451 and to write to the transport tape is contained in Reference 10. The information on either tape can be printed out for verification or other purposes using the TAPEOUT utility described in Reference 11.

B. CHANGING THE ENVIRONMENT

Once Hershey's data was initially available on the PDP-11, it was still not ready for manipulation. It was necessary to convert the EBCDIC characters that were used on the IBM-360/67 to ASCII characters that could be used on the PDP-11; fortunately, the "dd" shell command described in Reference 15 made converting the tape a fairly simple process. By using "dd" as follows,

```
dd if=/dev/rmt5 of=digit bs=80 cbs=80 skip=N count=M
   conv=ascii
```

where N is the number of records to skip before starting to copy and M is the number of records to be copied, the UNIX shell would read the EBCDIC tape in logical records, i.e., card images. The EBCDIC characters were then converted to ASCII, trailing blanks were omitted and '\n' (newline) was appended to the line before it was sent to the output. The resulting file contained a series of groups of ASCII characters; each group was no larger than 77 characters and no smaller than 28, and each group was ended by newline.

The first task was to strip the groups, or records, of unnecessary characters; each record began with "2524" and could contain up to 24 additional characters that were not coordinate pairs needed for the vector generation, but that were padding characters. The following program, "cnvrt.c", took those records and output logical records of the form:

```
CCCCSSSX1Y1X2Y2X3Y3...XiYi...XkYk\n
```

where

CCCC = Hershey character number (describes font, etc.)

SSS = card sequence number (one card was not usually enough)

XiYi = one endpoint of a vector

and k must be less than or equal to twelve, since there was room for a maximum of twelve coordinate pairs on each original EBCDIC input card image.

NAME

cnvrt -- convert a tape file for initial use

SYNOPSIS

< tapefile > > cnvrt > < pre-vector Hershey >

DESCRIPTION

After the required number of records have been read from the tape containing the vector representations, this program is used to strip away extraneous characters from each card image so that only character identification numbers, card sequence numbers, and the x,y coordinate pairs are left. This program is the first step in the adaptation of the Hershey fonts.

FILES

SEE ALSO

BUGS

The input and output files from this file must be redirected at the terminal. The input files are available on tape, as is the entire original EBCDIC tape of the vector definitions.

```
                /*                          */

                /*        cnvrt.c           */

                /*                          */

int     oid,            // old card id nr
        nid,            // new card id nr
        num1,           //
        num2,           //
        maxnr,          // number of last character + 1
        endf,           // nid of last character in this font
        *ptr;           // pointer for array ops
char    temp[4],        // temporary holder for characters
        hold[7],        // another temporary array...
        card[80],       // array to hold card images
        ncard[80],      // array for "stripped" card images
        strip,          // flag--to strip or not to strip a card
        flag,           // flag for outputting characters
        go,             // flag to stop recursion
        i,              // counter for "for" loops
        n,              // card column counter...
        csn,            // card sequence number
        cha,            //
        chb,            //
        chc,            // char variables for various ops
        chd,            //
        che;            //

        /*                          */
        /*      PROCEDURES          */
        /*                          */

getcard( val )                  // gets one card image (a logical
int val ;  {                    // record) at a time
   char j, t ;
   i = 0 ;
   che = 'A' ;
   strip = 1 ;
   while ( ( card[i] = getchar() ) != '\n' )
        i++ ;
   t = 0 ;
   for ( j = 4 ; j <= i ; j++ )  {
        if ( (j == 11) && ( (card[j] == '}') ||
                            (card[j] == '0') ) )  {
            if ( card[j] == '}' ) save() ;
            if ( card[j] == '0' )
                strip = compar( &card[j+1],hold ) ;
            if ( strip )  {
                for ( j = 11 ; j <= 25 ; j++ )
                    cha = card[j] ;
                che = card[j] ;
                }
            else
                {
```

```
                    ncard[t] = card[j] ;
                    t++ ;
                    }
                }
            else
                {
                ncard[t] = card[j] ;
                t++ ;
                }
            }
        val = conv( ncard,4 ) ;
        return( val ) ;
        }

save( )  {                        // holds 7 characters temporarily
char k ;
    for ( k = 0 ; k < 7 ; k++ )
        hold[k] = card[k+12] ;
}

compar( arrc,arrh )                   // compare 2 strings;
char arrc[ ], arrh[ ] ;  {  // return 1 if =, 0 if !=
char k ;
    k = 0 ;
    while ( arrc[k] == arrh[k] )  {
        k++ ;
        if ( k == 7 ) break ;
        }
    if ( k == 7 )
        return( 1 ) ;
    else
        return( 0 ) ;
}

conv( arr,nr )                    // converts a string of nrs of
char arr[ ], nr ;  {              // length nr to a decimal value
    int j, sum ;
    char  ch ;
    sum = 0 ;
    for ( j = 0 ; j < nr ; j++ )  {
        ch = arr[j] ;
        sum = sum + ( (ch - '0')*ten( nr - (j+1) ) ) ;
        }
    return( sum ) ;
}

ten( nr )                         // returns 10**nr
int nr ;  {
    int j, sum ;
    sum = 1 ;
    for ( j = 0 ; j < nr ; j++ )
        sum = sum  * 10 ;
    return( sum );
}
```

```
endch( ch )                          // got all end-points ?
char ch ;   {
    int ok ;
    switch( ch )     {
        case 'J':  case 'K':  case 'L':  case 'M':
        case 'N':  case 'O':  case 'P':  case 'Q':
        case 'R':  case '}':
            ok = 0 ;
            break ;
        default:
            ok = 1 ;
        }
    return( ok ) ;
}

finch( nr )                                 // removes filler chars
int nr ;   {
    int t ;
    while ( (ncard[nr] != '\n') && ( go ) )   {
        go = endch( ncard[nr] ) ;
        nr++ ;
        }
    if ( go )   {
        t = getcard( 0 ) ;
        if ( t == nid )  go = endch( che ) ;
        finch( 0 ) ;
        }
    }

        /*                           */
        /*       MAIN PROGRAM        */
        /*                           */

main( )   {
nid = 0 ;
maxnr = 3927 ;
endf = 3729 ;
while ( nid < maxnr )   {
    nid = getcard( 0 ) ;
    ptr = &ncard[4] ;
    csn = conv( ptr,3 ) ;
    flag = 1 ;
    go = 1 ;
    n = 7 ;
    while ( ( temp[0] = ncard[n] ) != '\n' )   {
        for ( i = 1 ; i < 4 ; i++ )
            temp[i] = ncard[n+i] ;
        n = n + 4 ;
        ptr = &temp[2] ;
        num1 = conv( temp,2 ) ;
        num2 = conv( ptr,2 ) ;
        if ( (num1 == num2) && (num1 == 50) )   {
            for ( i = 0 ; i < n ; i++ )
                putchar( ncard[i] ) ;
            putchar( '\n' ) ;
```

```
            finch( n ) ;
            flag = 0 ;
            if ( nid >= endf )  nid = 4000 ;
            }
        }
        if ( flag )
            for ( i = 0 ; i <= n ; i++ )
                putchar( ncard[i] ) ;
    }
}
```

C. VECTOR GENERATION

Once the Hershey data had been converted to ASCII and stripped of extraneous digits and characters, it was converted to vector form using the "mkvec.c" program listed below. The output files from "cnvrt.c" provided the input to "mkvec.c"; these files had been arranged by font with upper case letters, lower case letters, and digits and special characters grouped in that order. A large "if ... else ..." statement was used to determine whether each character being processed was a letter or a digit or special character; letters were placed into the proper position by an array counter, and digits and special characters were run through a large case statement to determine their position in the array. Two additional large case statements were necessary to transliterate the 24 Greek and the 32 Cyrillic letters into their English equivalents.

The transliterations used were taken from Reference 4 and are reproduced on the following page. The Gothic German was transliterated into English on a one-for-one basis, with the three extra lower case letters going into the octal

A→A B→B G→Γ D→Δ E→E Z→Z H→H Q→Θ I→I K→K L→Λ M→M N→N X→Ξ O→O P→Π R→P S→Σ T→T U→Υ F→Φ C→X Y→Ψ W→Ω a→α b→β g→γ d→δ e→ε z→ζ h→η q→θ i→ι k→κ l→λ m→μ n→ν x→ξ o→o p→π r→ρ s→σ t→τ u→υ f→φ c→χ y→ψ w→ω

FIGURE C-1. Greek Transliteration

АБВГД ЕЁЖЗИ ЙКЛМН ОПРСТ УФХЦЧ ШЩЪЫЬ ЭЮЯ абвгд еёжзи йклмн опрст уфхцч шщъыь эюя
Transliteration: A→А B→Б V→В G→Г D→Д E→Е %→Ё T→Ж Z→З I→И Y→Й K→К L→Л M→М N→Н O→О P→П R→Р S→С T→Т U→У F→Ф H→Х ←→Ц *→Ч @→Ш &→Щ '→Ъ #→Ы ∩→Ь U→Э ⊂→Ю ⊃→Я a→а b→б v→в g→г d→д e→е $→ё t→ж z→з i→и y→й k→к l→л m→м n→н o→о p→п r→р s→с t→т u→у f→ф h→х →→ц "→ч ⊕→ш +→щ '→ъ =→ы ∧→ь V→э <→ю >→я

FIGURE C-2. Cyrillic Transliteration

codes 0173, 0174, and 0175, immediately following the "z". Gothic Italian was transliterated completely on a one-to-one basis.

This program put the digitization into the form that was used for the TEKTRONIX program described in Appendix D, and as the data base from which the dot matrix character representations were made.

NAME

mkvec -- put a file into vector format

SYNOPSIS

mkvec < Hershey font >

DESCRIPTION

The output from "cnvrt" provides the input for this program; this program takes each character identification number, goes through each card with that character id, and puts each x,y coordinate pair into a 16-bit word. The left byte receives the x coordinate and the right byte receives the y coordinate.

The program also sets up a 256 word directory for each font. The even numbered words from 0 to 254 correspond to the ASCII codes from 0 to 0177 and contain the widths of the appropriate character. The odd numbered words contain pointers to the byte at which the vector definition of the character begins. This program's output files provided the data base from which "drawhf" and "makehf" obtain their vector definitions for either display or digitization.

FILES

SEE ALSO

makehf, cnvrt

BUGS

"mkvec" automatically writes the vector files to directory "/usr/doyle/hfonts.f", which might not be in existance at this point in time. If this program must be used, then a simple modification to the source code will write the output to any file desired.

```
                /*                          */

                /*          mkvec.c         */

                /*                          */

#define         SR      0       // Simplex Roman
#define         DR      1       // Duplex Roman
#define         CR      2       // Complex Roman
#define         TR      3       // Triplex Roman
#define         CI      4       // Complex Italic
#define         TI      5       // Triplex Italic
#define         SS      6       // Simplex Script
#define         CS      7       // Complex Script
#define         SG      8       // Simplex Greek
#define         CG      9       // Complex Greek
#define         GE      10      // Gothic English
#define         GG      11      // Gothic German
#define         GI      12      // Gothic Italian
#define         CC      13      // Complex Cyrillic
#define         NOTNUM      -1
#define         EOC         031062  // octal for "5050"
#define         EOF         9999    // end of font for M[]
#define         ENDFONT     9000    // end of font n
#define         R       0       // open a file for reading
#define         W       1       // open a file for writing
#define         MODE    0644    // access to files created

int M[10192] ;
int posit,
   apos,
   num3,
   idh,
   ktr,
   cw,
   xi,
   *ptr;

char   card[80],
       xyval[144] [2],
       temp[4],
       flag,
       cktr,
       num1,
       num2,
       csn;

char   infile[ ]  {
           "/.doyle.01/fonts/--.out"   } ;
char   outfile[ ] {
           "/usr/doyle/hfonts.f/--.f"  } ;
char   rptr, wptr ;

setcar( x, y )
int x, y ; {
```

```
    M[x] = ( M[x] & 0377 ) | ( y<<8 ) ;
}

setcdr( x, y )
int x, y ; {
    M[x] = ( M[x] & 0177400 ) | y ;
}

getcard( ) {
    int *p, i ;
    char buf[], t ;
    i = 0 ;
    t = read( rptr, buf, 1 ) ;
    while ( (card[i] = buf[0]) != '\n' )
        {
        t = read( rptr, buf, 1 ) ;
        i++ ;
        }
    idh = conv( card,4 ) ;
    p = &card[4] ;
    csn = conv( p,3 ) ;
}                               // end getcard()

start( )    {
    int j ;
    for ( j = 0 ; j < 256 ; j++ )
        M[j] = 0 ;
    rptr = open( infile, R ) ;
    wptr = creat( outfile, MODE ) ;
    idh = 0 ;
    ktr = 64 ;
    cktr = 0 ;
    flag = 1 ;
    apos = 256 ;
}                               // end start

reset( )    {
    int i ;
    M[0] = M[1] = 0 ;
    M[apos] = EOF ;
    for ( i = 0 ; i <= apos ; i++ )
        write( wptr, &M[i], 2 ) ;
}                               // end reset

notnum( val )
int val ;   {           // is this a number or char?
    int ok ;
    if ( (val >= 700) && (val <= 734) )
        {
        if ( val <= 715 )  ok = val - 700 ;
        else  ok = val ;
        }       // SR, SG, SS number
    else
      if ( (val >= 2200) && (val <= 2275) )
        {
```

```
        if ( val <= 2215 )  ok = val - 2200 ;
        else  ok = val ;
        }       // CR, CG, CI number
      else
        if ( (val >= 2700) && (val <= 2728) )
          ok = val - 2700 ;
            // DR number
        else
          if ( (val >= 2750) && (val <= 2778) )
            ok = val - 2750 ;
              // CS number
          else
            if ( (val >= 3200) && (val <= 3228) )
              ok = val - 3200 ;
                // TR number
            else
              if ( (val >= 3250) && (val <= 3278) )
                ok = val - 3250 ;
                  // TI number
              else
                if ( (val >= 3700) && (val <= 3728) )
                  ok = val - 3700 ;
                    // GE, GG, GI number
                else
                    ok = NOTNUM ;
                      // it's a character...
    return( ok ) ;
}                               // end notnum

cyr( val )
int val;  {
int ok;

    switch( val )  {
        case 2801:        // A
          ok = 65;  break;

        case 2802:        // B
          ok = 66;  break;

        case 2803:        // V
          ok = 86;  break;

        case 2804:        // G
          ok = 71;  break;

        case 2805:        // D
          ok = 68;  break;

        case 2806:        // E
          ok = 69;  break;

        case 2807:        // ↑
          ok = 94;  break;
```

```
case 2808:          // Z
  ok = 90;   break;

case 2809:          // I
  ok = 73;   break;

case 2810:          // Y
  ok = 89;   break;

case 2811:          // K
  ok = 75;   break;

case 2812:          // L
  ok = 76;   break;

case 2813:          // M
  ok = 77;   break;

case 2814:          // N
  ok = 78;   break;

case 2815:          // O
  ok = 79;   break;

case 2816:          // P
  ok = 80;   break;

case 2817:          // R
  ok = 82;   break;

case 2818:          // S
  ok = 83;   break;

case 2819:          // T
  ok = 84;   break;

case 2820:          // U
  ok = 85;   break;

case 2821:          // F
  ok = 70;   break;

case 2822:          // H
  ok = 72;   break;

case 2823:          // ←
  ok = 95;   break;

case 2824:          // #
  ok = 35;   break;

case 2825:          // @
  ok = 64;   break;

case 2826:          // &
```

```
  ok = 38;  break;

case 2827:        // '
  ok = 39;  break;

case 2828:        // 036
  ok = 30;  break;

case 2829:        // 022
  ok = 18;  break;

case 2830:        // 023
  ok = 19;  break;

case 2831:        // 020
  ok = 16;  break;

case 2832:        // 021
  ok = 17;  break;

case 2901:        // a
  ok = 97;  break;

case 2902:        // b
  ok = 98;  break;

case 2903:        // v
  ok = 118;  break;

case 2904:        // g
  ok = 103;  break;

case 2905:        // d
  ok = 100;  break;


case 2906:        // e
  ok = 101;  break;

case 2907:        //
  ok = 1;  break;

case 2908:        // z
  ok = 122;  break;

case 2909:        // i
  ok = 105;  break;

case 2910:        // y
  ok = 121;  break;

case 2911:        // k
  ok = 107;  break;

case 2912:        // l
```

```
    ok = 108;  break;

  case 2913:        // m
    ok = 109;  break;

  case 2914:        // n
    ok = 110;  break;

  case 2915:        // o
    ok = 111;  break;

  case 2916:        // p
    ok = 112;  break;

  case 2917:        // r
    ok = 114;  break;

  case 2918:        // s
    ok = 115;  break;

  case 2919:        // t
    ok = 116;  break;

  case 2920:        // u
    ok = 117;  break;

  case 2921:        // f
    ok = 102;  break;

  case 2922:        // h
    ok = 104;  break;

  case 2923:        //
    ok = 25;  break;

  case 2924:        // "
    ok = 34;  break;
  case 2925:        // 026
    ok = 22;  break;

  case 2926:        // +
    ok = 43;  break;

  case 2927:        // 0140
    ok = 96;  break;

  case 2928:        // =
    ok = 61;  break;

  case 2929:        // 004
    ok = 4;  break;

  case 2930:        // 037
    ok = 31;  break;
```

```
        case 2931:       // <
          ok = 60;   break;

        case 2932:       // >
         ok = 62;   break;

    }
return( ok ) ;
}                          // end cyr


grkch( val )
int val ;  {               // which Greek character?
int ok ;

switch( val )  {

   case 529: case 2029:  // G
       ok = 71;  break ;

   case 532: case 2032:  // Z
       ok = 90;  break;

   case 533: case 2033:  // H
       ok = 72;  break;

   case 534: case 2034:  // Q
       ok = 81;  break;

   case 536: case 2036:  // K
       ok = 75;  break;

   case 537: case 2037:  // L
       ok = 76;  break;

   case 538: case 2038:  // M
       ok = 77;  break;

   case 539: case 2039:  // N
       ok = 78;  break;

   case 540: case 2040:  // X
       ok = 88;  break;

   case 543: case 2043:  // R
       ok = 82;  break;

   case 544: case 2044:  // S
       ok = 83;  break;

   case 545: case 2045:  // T
       ok = 84;  break;

   case 546: case 2046:  // U
       ok = 85;  break;
```

```
case 547: case 2047:  // F
    ok = 70;  break;

case 548: case 2048:  // C
    ok = 67;  break;

case 549: case 2049:  // Y
    ok = 89;  break;

case 550: case 2050:  // H
    ok = 87;  break;

case 629: case 2129:  // g
    ok = 103;  break;

case 632: case 2132:  // z
    ok = 122;  break;

case 633: case 2133:  // h
    ok = 104;  break;

case 634: case 2134:  // q
    ok = 113;  break;

case 636: case 2136:  // k
    ok = 107;  break;

case 637: case 2137:  // l
    ok = 108;  break;

case 638: case 2138:  // m
    ok = 109;  break;

case 639: case 2139:  // n
    ok = 110;  break;

case 640: case 2140:  // x
    ok = 120;  break;

case 643: case 2143:  // r
    ok = 114;  break;

case 644: case 2144:  // s
    ok = 115;  break;

case 645: case 2145:  // t
    ok = 116;  break;

case 646: case 2146:  // u
    ok = 117;  break;

case 647: case 2147:  // f
    ok = 102;  break;

case 648: case 2148:  // c
```

```
        ok = 99;  break;

    case 649: case 2149:  // y
        ok = 121;  break;

    case 650: case 2150:  // w
        ok = 119;  break;

    default:                // A,B,D,E,I,O,P (u & 1)
        ok = ktr;  break;

    }
return( ok ) ;
}

which( val )
int val ;  {
int ok ;
    if ( val < 10 )  ok = val + 48 ;
    else
        switch( val )   {

        case 10:                //  .
            ok = 46 ;  break ;

        case 11:                //  ,
            ok = 44 ;  break ;

        case 12:                //  :
            ok = 58 ;  break ;

        case 13:                //  ;
            ok = 59 ;  break ;

        case 14:                //  !
            ok = 33 ;  break ;

        case 15:                //  ?
            ok = 63 ;  break ;

        case 716: case 2216: //  '
        case 27:
            ok = 39 ;  break ;

        case 717: case 2217: //  "
        case 28:
            ok = 34;  break ;

        case 18:  case 734: //  &
        case 2272:
            ok = 38 ;  break ;

        case 719: case 2274: // $
        case 19:
            ok = 36 ;  break ;
```

```
case 720: case 2220: //  /
case 20:
    ok = 47 ;  break ;

case 721: case 2221: //  (
case 21:
    ok = 40 ;  break ;

case 722: case 2222: //  )
case 22:
    ok = 41 ;  break ;

case 728: case 2219: //  *
case 23:
    ok = 42 ;  break ;

case 724: case 2231: //  -
case 24:
    ok = 45 ;  break ;

case 725: case 2232: //  +
case 25:
    ok = 43 ;  break ;

case 726: case 2238: //  =
case 26:
    ok = 61 ;  break ;

case 723: case 2229: //  ¦
    ok = 124 ;  break ;

case 733: case 2275: //  #
    ok = 35 ;  break ;

case 2223:           //  [
    ok = 91 ;  break ;

case 2224:           //  ]
    ok = 93 ;  break ;

case 2225:           //  {
    ok = 123 ;  break ;

case 2226:           //  }
    ok = 125 ;  break ;

case 2241:           //  <
    ok = 60 ;  break ;

case 2242:           //  >
    ok = 62 ;  break ;

case 2262:           //  ↑
    ok = 94 ;  break ;
```

```
        case 2263:              //  ←
             ok = 95 ;  break ;

        case 2271:              //  %
             ok = 37 ;  break ;

        case 2273:              //  @
             ok = 64 ;  break ;

        default:
             ok = 0 ;  break ;

        }
    return( ok ) ;
}                               //  end  of which


buildch( )  {
int p, test ;
int k ;
    p = apos ;
    for ( k = 0 ; k < xi ; k++ )  {
        setcar( apos,xyval[k][0] ) ;
        setcdr( apos,xyval[k][1] ) ;
        apos++ ;
        }
    test = M[apos - 1] ;
    if ( test != EOC )
        perror("stopped before EOC") ;
    return( p ) ;
}

conv( arr,nr )                  // converts a string of numbers of
char arr[ ], nr ;  {            // length nr to a decimal value
    int j, sum ;
    char  ch ;
    sum = 0 ;
    for ( j = 0 ; j < nr ; j++ )  {
        ch = arr[j] ;
        sum = sum + ( (ch - '0')*ten( nr - (j+1) ) ) ;
        }
    return( sum ) ;
}

ten( nr )                       // returns 10**nr
int nr ;  {
    int j, sum ;
    sum = 1 ;
    for ( j = 0 ; j < nr ; j++ )
        sum = sum  * 10 ;
    return( sum );
}

main( argc, argv )
int argc ;
```

```
char *argv[] ;  {
int n, *p, last , bigch ;
int k ;
char greek, cyrillic;

infile[17] = outfile[20] = argv[1][0] ;
infile[18] = outfile[21] = argv[1][1] ;
bigch = 26 ;
greek = 0 ;
cyrillic = 0 ;
if ( infile[18] == 'G' )  {
   if ( infile[17] != 'G' )  { bigch = 24; greek = 1 ; }
   }
if ( infile[18] == 'C' ) cyrillic = 1 ;

start( ) ;
getcard() ;

   while ( idh != ENDFONT )  {
        xi = 0 ;
        last = idh ;
        ktr = ktr + 1 ;
        cktr = cktr + 1 ;

        while ( idh == last )  {
             n = 7 ;

             while ( (temp[0] = card[n]) != '\n' ) {
                  for ( k = 1; k < 4; k++ )
                       temp[k] = card[n+k] ;
                  n = n + 4 ;
                  p = &temp[2] ;
                  num1 = conv( temp,2 ) ;
                  num2 = conv( p,2 ) ;
                  if ( (csn == 1) && (n == 11) )  {
                       if ( num1 > 50 ) num1 = 100 - num1 ;
                       cw = num1 + num2 ;
                       }
                  else
                       {
                       xyval[xi][0] = num1 ;
                       xyval[xi][1] = num2 ;
                       xi++ ;
                       }
                  }

             getcard( ) ;
             }

        if ( ( cktr > bigch ) && ( flag ) )  {
             ktr = 97 ;
             cktr = 0 ;
             flag = 0 ;
             }
        if ( (num3 = notnum(last)) == NOTNUM )
```

```
            {
            if ( greek )  posit = grkch( last ) * 2 ;
             else if ( cyrillic ) posit = cyr( last ) * 2 ;
            else  posit = ktr * 2 ;
            }
        else
            {
            posit = which( num3 ) * 2 ;
            ktr = 0 ;
            }
        ptr = buildch( ) ;
        M[ posit ] = cw ;
        M[ posit + 1 ] = ptr ;

        }
    reset( ) ;

}                              // end of main...
```

APPENDIX D. FONT OUTPUT ROUTINES

A. VERIFYING THE VECTOR DATA

After all of the fonts had been converted into vector form, "drawhf.c" was written to allow visual inspection of each character in each font. This inspection ensured that all of the data had been transformed correctly and was available for further use in the vector to dot matrix conversion. It also revealed several minor omissions that had allowed special cases to slip through the vector generation program described in paragraph C of Appendix C.

This program is available as source code on directory "/.fonts.02/hershey" and is listed as "drawhf.c". The object code program used to display the characters on the TEKTRONIX 4014 is available on directory "/.fonts.01/hftools". Any character from any of the fonts currently available can be drawn by changing to the directory above and typing "drawhf FONT", where FONT is a three-character code specifying the font desired. Fonts available are:

HSR.....Simplex Roman

HDR.....Duplex Roman

HCR.....Complex Roman

HTR.....Triplex Roman

HCI.....Complex Italic

HTI.....Triplex Italic

HSS.....Simplex Script

HCS.....Complex Script

HSG.....Simplex Greek

HCG.....Complex Greek

HGE.....Gothic English

HGG.....Gothic German

HGI.....Gothic Italian

HCC.....Complex Cyrillic

Many special characters are available only in the complex fonts; however, the user is notified if the character desired is not available in the font currently being displayed. The size of the character drawn on the CRT can be changed by adding a size parameter to the program call, i.e., "drawhf FONT SIZE". If no "SIZE" parameter is given, the program defaults to a value of eight; this size was chosen because it made all of the vectors visible, and because it minimized the distortion noticeable on the short vectors used to approximate curves. Parameters larger than 20 and less than one will default to those values.

NAME

drawhf -- draw a Hershey font on the TEKTRONIX 4014

SYNOPSIS

drawhf < Hershey font > [size]

DESCRIPTION

This program draws characters from the selected Hershey font on the TEKTRONIX 4014. The fonts must be selected from the list given on the preceding pages.

The size of the character display on the CRT can be changed by executing the program with an optional size parameter. This should be an integer between 1 and 20. The default value is 8.

FILES

SEE ALSO

makehf

BUGS

Only one character can be drawn at a time. It is also necessary to terminate the program and re-execute it to look at another font.

```
                    /*                              */

                  /*            drawhf.c              */

                    /*                              */

#define         EOF      9999
#define         EOC      031062
#define         RES      1024
#define         X        512
#define         Y        512
#define         R        0
#define         W        1

int M[256],
    fptr,
    ptr;

int     ch[120];

char    path[]   {
             "/.fonts.01/hershey/--.v"
             }  ;
char    flag ;

main( argc, argv )
int argc ;
char *argv[] ;   {
int x, y, ktr , xx, yy ;
int i, p, num1, num2, size ;
char ibuf[20], ip, k , stop , times ;
char doit ;

initt(960) ;
term(3, RES) ;
path[19] = argv[1][1] ;
path[20] = argv[1][2] ;

if ( argc == 3 )  {
    size = atoi( argv[2] ) ;
    if ( size > 20 ) size = 20 ;
    else if ( size <= 1 ) size = 1 ;
    }
else  size = 8 ;

fptr = open( path, R ) ;
k = read( fptr, M, 512 ) ;

ibuf[0] = '0' ;
while ( ibuf[0] != ']' )  {
    i = ip = 0 ;
    flag = 1 ;
    erase() ;
    movabs(100,800) ;
    anmode() ;
```

```
printf("input the desired character followed by c/r:") ;
while ( (ibuf[ip] = getchar()) != '\n' )
    ip++ ;
xx = X ;
yy = Y ;
movabs(xx,yy) ;
p = ( num1 = ibuf[0] )<<1 ;

while ( flag )  {
    stop = times = 1 ;
    ptr = M[p+1]<<1 ;
    if ( ptr == 0 )  {
        movabs(100,200) ;
        anmode() ;
        printf("SORRY:'%c' is not available in this font...",
                                            ibuf[0]) ;
        stop = 0 ;
        }
    else k = seek( fptr, ptr, 0 ) ;

    while ( stop )  {
        doit = 1 ;
        k = read( fptr, ibuf, 2 ) ;
        x = ibuf[1] ;
        y = ibuf[0] ;
        if ( x > 50 ) x = x - 100 ;
        if ( y > 50 ) y = y - 100 ;
        if ( times )  {
            x = xx + (x * size) ;
            y = yy - (y * size) ;
            movabs(x,y) ;
            doit = 0 ;
            times = 0 ;
            }
        if ( x == 50 )  {
            if ( y != 0 )  {
                if ( y == 50 )  stop = 0 ;
                else  perror("bad y value...") ;
                }
            else
                {
                k = read( fptr, ibuf, 2 ) ;
                if ( (x = ibuf[1]) > 50 )  x = x - 100 ;
                if ( (y = ibuf[0]) > 50 )  y = y - 100 ;
                x = xx + (x * size) ;
                y = yy - (y * size) ;
                movabs(x,y) ;
                }
            }
        else  {
            if ( doit )
            {
            x = xx + (x * size) ;
            y = yy - (y * size) ;
            drwabs(x,y) ;
```

```
                }
            }
        }

    flag = 0 ;
    tsend() ;
    }

movabs(100,150) ;
anmode() ;
printf("enter c/r to continue, 1 c/r to exit:") ;
ip = 0 ;
while ( (ibuf[ip] = getchar()) != '\n' )
    ip++ ;
}

erase() ;
finitt(0,750) ;
}
```

A sample of the CRT display from "drawhf" is located on the following page.

B. VERIFYING DIGITIZED FONTS

"Prfont" is a font manipulation program designed to display an entire digitized font file by walking through the header table and plotting all of the characters that are defined in the font. It will print out one font at a time or as many as are needed, depending upon the arguments. The program accepts full path names as input; these arguments can be used in several ways, as is demonstrated by the examples given below. All of these are valid calls to "prfont":

1) prfont HCC12 BDJ8

The user wants to display fonts HCC12 and BDJ8, both of which must exist on directory "/.fonts.01/font".

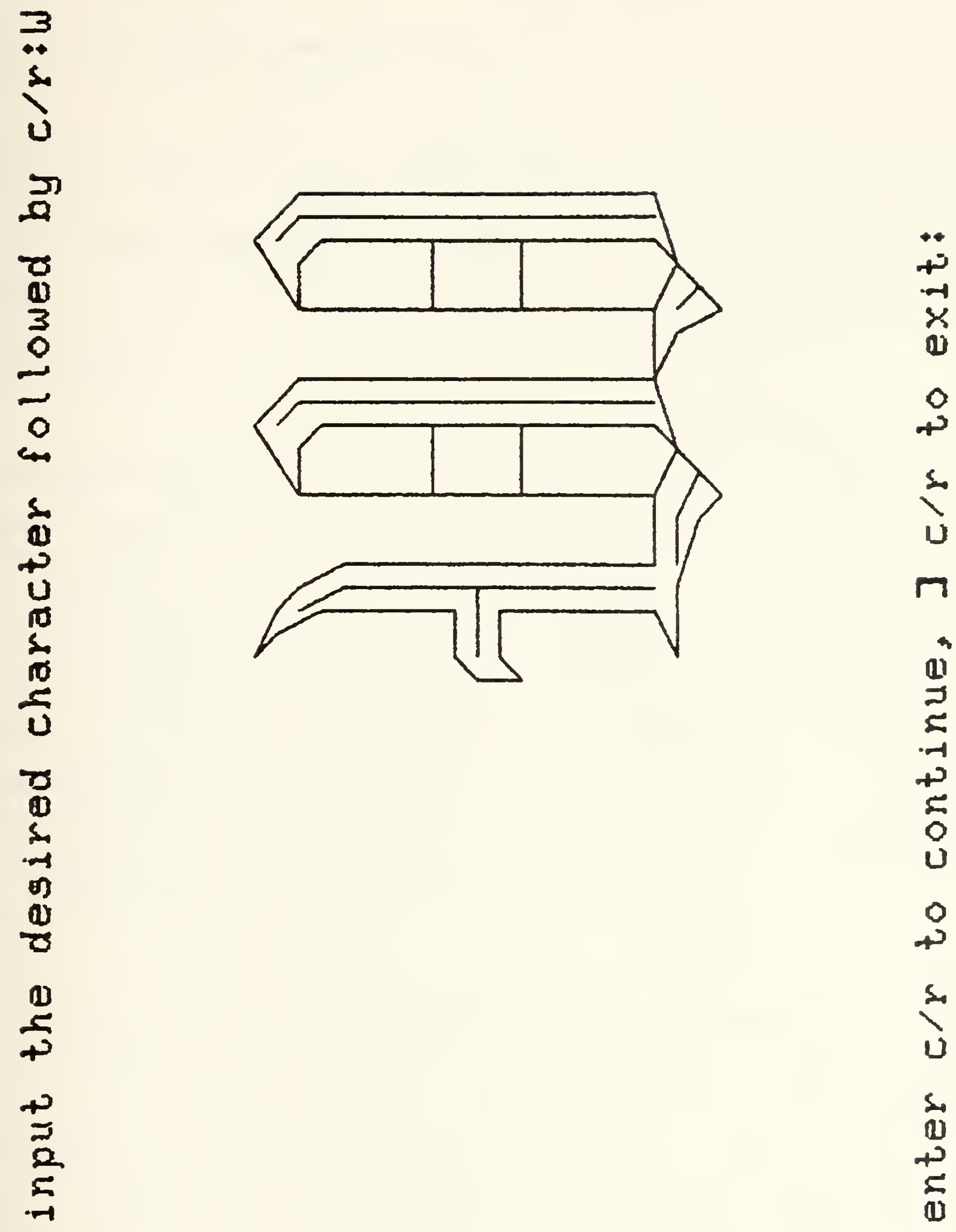

FIGURE D-1. "Drawhf" CRT Display

2) prfont /usr/doyle/fonts/H*

The user desires to display all of the Hershey fonts located on a specified directory.

3) prfont SIGN41 HGG16 BDR25 HCG10 /usr/mccord/temp

The user wants to display four files from directory "/.fonts.01/font" and a file called "temp" on directory "/usr/mccord".

Commands of the type "prfont *", "prfont H*", and "prfont BD*", will not display all of the fonts on the main font directory, nor will they display any combination of them. To display the entire collection of fonts or selected groups of them, the user must change directories to "/.fonts.01/font" and type "../prfont < ARG >", where ARG is some font name combination of the form *, H*, MATH*, and so on.

"Prfont" will handle spacing and pagebreaks, and will print the font file name with each font. With fonts above 40-42 point, the program may tell you that it is out of memory and exit. It will suggest that you try a smaller pagewidth, which will cause your fonts to be plotted with fewer characters per line.

NAME

prfont -- display a digitized font on the VERSATEC plotter/printer

SYNOPSIS

prfont [-pagewidth] < SAIL font > ¦ < Hershey font > ¦ < complete path name >

DESCRIPTION

This program allows the user to display a complete digitized font file on the VERSATEC plotter/printer, so that he can see how it will actually look. It is especially useful in seeing whether or not a Hershey font will be acceptable after digitization.

On fonts larger than 40-42 point, it may be necessary to decrease the pagewidth used by "prfont" to determine the size of its plot buffer. If this becomes necessary, the program will exit and suggest that you try a smaller pagewidth. Pagewidth is initialized to 216 bytes.

FILES

/dev/rvp
/dev/spp

SEE ALSO

makehf, edf

BUGS

"prfont" occasionally prints some extra dots and lines after completing the last character in the font directory.

```
/*                              */

/*          orfont.c          */

/*                              */

#define SPACE 1             // one 1/4 inch vertical space
#define TOP 230             // top margin
#define PAGEHT 14*100
int roww, rows;
int linecount PAGEHT;
int pagewth;
int prdev, pldev, infont;
int ht, maxw, lht, fp;
int head, tail, nodeptr;
int zero[1], hdr[256];
char *lp, *p;
char ff 014; char nl 012;
char header[40] {"/.fonts.01/font/"} ;
char prbuf[132], plbuf[264];
struct cnode   {
   int cc;                  //char code
   char *optr;              //->1st raster line
   char *lptr;              //-> next raster line
   int rw;                  //raster line width
   int bytes;               //bytes per raster line
   int lk;                  //left kern
   int rft;                 //rows from top
   int drc;                 //data row count
} clist[128];
struct cnode *a;
struct cnode *fset[128];

main(argc, argv)
   int argc; char **argv;    {
   register int i, argptr;
   char go;
   argptr = 1;
   if((prdev=open("/dev/spp",1)) < 0)     {
      printf("cannot open printer");exit();}
   if((pldev=open("/dev/rvp",1)) < 0)     {
      printf("cannot open plotter");exit();}
   if (argv[1][0] == '-') {//reset pagewth
      pagewth = atoi( &argv[1][1] ); go = 1;   }
   else  { pagewth = 216; go = 0; }
   init();
   while(--argc != go) {//process all files
      p = argv[argptr+go];
      if ( *p == '/' ) { //full pathname
         if ( (fp=open(argv[argptr+go],0)) < 0) {
            printf("cannot open %s",argv[argptr+go]);
            exit();   }
         printf("%s opened....",argv[argptr+go]);
         }
      else { //prepend /fonts.01/font
```

```
            for(i=16;(header[i]= *p++) != '\0';i++) ;
            if((fp=open(header,0)) < 0)     {
              printf("cannot open %s",header);exit();}
            printf("%s opened.....",header);
            }
         infont = head = tail = nodeptr = roww = 0;
         read(fp,hdr,512); read(fp,&ht,2);
         read(fp,&maxw,2); read(fp,&lht,2);
         check(); //check for bad font file
        if ( ht <= 82 ) {
         //set vert spacing
          if (ht <= 40) rows = 2 ;
          else rows = 3 ;
          }
        else rows = 4 ;
         //pgbk if font display won't fit
         if(nroom(rows*ht + 40)) pagebreak();
         p = prbuf; for(i=0;i<60;i++) *p++ = ' ';
         for(i=0;(*p++ = argv[argptr+gol[i]) != '\0';i++);
         *p = nl;
         //center, write font name
         write(prdev,prbuf,i+62);
         for(i=0;i<25;i++) write(pldev,zero,2);
         linecount =+ 25;
         while (1)     {
            getrow();
            putrow();
            if(infont > 127) break;
         }
         close(fp); printf("closed\n"); argptr++;
         //if need be, pgbk
         if(nroom(SPACE*2)) pagebreak();
         else space(SPACE*2);
   }
   exit();
}

init()     {
   register int i;
   for(i=0;i<128;i++) fset[i] = &clist[i];
}

pagebreak() { //page eject
   int i;
   char err;
   err = cvers(pldev,020);
   if ( err == -1 ) {
   printf("invalid filedes in pagebreak\n");
   exit();
   }
   for (i=0;i<TOP;i++) write(pldev,zero,2);
   linecount = TOP;
}

getrow() { //get a row of chars to plot
```

```
    if(tail)    {
        roww = fset[++tail]->bytes;
        head = tail++;
    }
    while (1)    {
        if(getdef())    {
            if(roww + fset[tail]->bytes <= pagewth)
                roww =+ fset[tail]->bytes;
            else {tail--; ++infont; break;}
            if (++infont > 127) break;
            tail++;
        }
        else if(++infont > 127) break;
    }
}

putrow()  { //plot the row of characters
    register int h,i,l; int t;
    struct cnode *ptr;
    for(h=0; h < ht; h++)    {
        p = &plbuf[24];
        ptr = fset[(t = head)];
        for(l=head;l<=tail;l++)    {
            if(ptr->drc)    {
                if(h >= ptr->rft && h < ptr->rft+ptr->drc)    {
                    //lp-> next raster line
                    lp = ptr->lptr;
                    //do it by bytes
                    for(i=0;i<ptr->bytes;i++)
                        *p++ = *lp++;
                    //update lptr for next pass
                    ptr->lptr =+ ptr->bytes;
                }
                //blank line
                else for(i=0;i<ptr->bytes;i++) *p++ = 0;
            }
            //blank character
            else for(i=0;i<otr->bytes;i++) *p++ = 0;
            ptr = fset[++t];
        }
        //plot 1 raster line of row of characters
        write(pldev,plbuf,roof(roww+24));
    }
    //row plotted, plot some white space
    for(h=0;h<5;h++) write(pldev,zero,2);
    linecount =+ ht+5;
    //free bytes in reverse order
    for(i=tail;i>=head;i--)
        if(fset[i]->optr)
            free(fset[i]->optr);
}

int getdef()    {
    int blkc,bytc; register i;
    if(hdr[infont*2])    {
```

```
        blkc = (hdr[infont*2]&0177400) >> 8;
        blkc =& 0377;
        bytc = hdr[infont*2+1];
        if(blkc) { //ptr is in blks and bytes
           seek(fp,blkc,3); seek(fp,bytc,1); }
        else seek(fp,bytc,0);
        getnode();
        a->cc = infont; read(fp,&a->rw,2);
        read(fp,&a->lk,2); read(fp,&a->rft,2);
        read(fp,&a->drc,2);
        a->bytes = (a->rw%8 == 0) ? a->rw/8 : a->rw/8+1;
        if(fcheck()) { //check for bad char dimensions
           if(a->drc) { //need bytes?, call alloc
              if((i=a->optr=a->lptr=alloc(a->drc*a->bytes))<0){
                 printf("\nout of memory...");
                 printf("use a smaller pagewidth\n");
                 exit();  }
              read(fp,a->lptr,a->drc*a->bytes);
        }
        return(1);
        }
    }
    return(0);
}

getnode()    {
    if(nodeptr > 127)    {
       printf("overflow"); exit();}
    a = fset[nodeptr++];
    a->optr = a->lptr = 0;
}

int roof(x)
    int x; { //send plotter even # bytes only
    if(x%2 == 0) return(x);
    //for some reason 264 bytes crashes program
    if(x == 263) return(262);
    *p = 0; return(++x);
}

space(x)
    int x; { //plot x 1/4 inches space
    int i;
    for(i=0;i<x*50;i++) write(pldev,zero,2);
    linecount =+ x*50;
}

check() { //print then exit on bad file
    if(ht < 0 || maxw < 0 || lht < 0 ||
       ht > 256 || maxw > 256 || lht > ht)    {
          printf("bad file"); exit();
    }
}

int nroom(x)
```

```
    int x; { //rtn 1 there are not x plot lines
            //left before bottom; otherwise, 0
    if(linecount + x > PAGEHT) return(1);
    else return(0);
}

fcheck() { //if bad chardef, rtn 0 to skip it
            //otherwise; rtn 1.
    if ( (a->rw<0 || a->rw>255) || (a->rft<0 || a->rft>255)
      || (a->lk<0 || a->lk>255) || (a->drc<0 || a->drc>255)
          ) {
       printf("\ninvalid value for character '%c'\n",infont);
       printf("rw %d\trft %d\tlk %d\tdrc %d\n",a->rw,
              a->rft,a->lk,a->drc);
       return(0);
       }
    else return(1);
}
```

C. USING THE DIGITIZED FONTS

"Signmkr" is a program with limited text processing capabilities designed to fill an interim gap in the computerized typesetting system at NPS. It was designed to give the user a limited means of outputting small files that required the use of the fonts from the data base; when a virtual typesetter that will accept fonts with variable dimensions is developed, the "signmkr" can be used as a novelty program to generate signs and other small files that use exotic fonts. "Signmkr" can center lines of text, leave blank lines, cause pagebreaks and paragraphing, and can change font styles from line to line. The user may also insert literal codes to have a certain special character used in his output. The use of these commands is explained in the next paragraph; unless otherwise indicated, blanks are optional after commands but are recommended in most cases to

improve readability.

Commands accepted by "signmkr" are listed below. The letters "ESC" preceding each command represent the ASCII escape character at octal code 033, and "\n" is the standard "newline" character (octal 012) used to represent carriage return. Each of the following commands must begin at the beginning of a line and some must be on lines by themselves.

a) ESCc < one line of text >

The "center" command centers one and only one line of text, and that line is the line immediately following the command. This requires the user to use this command in each line to be centered. If a line is too long to be centered, then "signmkr" will inform the user of this and ignore the line.

b) ESCf< SAIL font > ¦ < Hershey font > ¦
< complete path name >

The "change font" command allows the user to change the font being used for typesetting; it must be used only at the head of a line or on a line by itself. A blank must not be left between the command and the new font name.

c) ESCpg\n

This is the "pagebreak" command and is similar to the ".bp" command used in NROFF. It sends a form-feed signal to

the VERSATEC and re-positions the text to the top of a new page. The command should be used on a line by itself.

d) ESCpp\n

The "begin paragraph" command indents the text line for paragraphing. The size of the indent is determined by the size of the current font. Like the "pagebreak" command, it should be on a line by itself.

e) ESCs< decimal number > ¦ < octal number >

The "space" command inserts blank lines within the text. The height of the blank line is equal to the font height. A blank must not be left between the command and the number.

The following command may be used at any place within an input line :

f) ESCo< octal nr > ¦ < decimal nr >

The "literal" command can be used to request a certain octal or decimal code that will be used to access a character within the current font. The command may be used at any point within a line, but it must not be followed by a blank. This command is useful to access either characters that the user may have inserted within a font file during an edit session, or to access characters from a SAIL font whose character codes correspond to control characters in ASCII.

Octal numbers begin with the character '0' and do not contain the numbers 8 or 9, e.g., 0176 and 0103.

Users with previous exoerience with text processing programs should have no trouble in adapting to "signmkr". However, caution should be exercised when using the "ESCpp" (paragraoh) and "ESCf" (change fonts) commands at the same point in the inout file. The two sequences of inout lines

```
(a) ESCf BDR8                (b) ESCf BDR8
    ESCpp\n                      ESCf HTR30\n
    ESCf HTR30\n                 ESCpp\n
    < input text >               < input text >
```

are not identical. Sequence (a) will set up the indentation for the next paragraph assuming a font height of 8 point, but the text will actually be set in 30 point type, so the indentation will not be obvious. Sequence (b) changes the font height to 30 point and then indents based on that height instead of 8 point.

NAME

signmkr -- a program with limited text processing ability; useful with small projects that require exotic fonts, or for making cute signs

SYNOPSIS

signmkr < source file >

DESCRIPTION

This program is capable of limited typesetting functions using commands described more fully above. It reads the input text and commands from a file located on the same directory as the program, in most cases "/.fonts.01".

When designing input files for the signmaker, the user should try to do as much of the formatting for the output file as is possible. The signmaker will, in general, give you back what you put in; it is very good at truncating lines that are too long and at not filling lines that may be too short.

Command Summary:

ESCc	Center one line of text
ESCf	Change the current font
ESCpg	Cause a pagebreak
ESCpp	Begin a paragraph
ESCs	Space down n lines
ESCo	Interpret the following number as a literal character code

FILES

SEE ALSO

BUGS

```
                /*                              */

              /*            signmkr.c         */

                /*                              */

#define TOP 230             // top margin
#define PAGEHT 14*100
int roww;
int sl 0;
int pagewth 216;
int linecount PAGEHT;
int pldev, infont, in, base;
int ht, maxw, lht, fp, ip, r;
int nodeptr, openbits;
int zero[32], hdr[256];
char *lp, *p, *t, *n, *pl;
char esc 033; char blank 040; int c;
char header[40] {"/.fonts.01/font/"} ;
char pbuf[90], tbuf[90], plbuf[264];
char fmark[128];
char fontname[20], ochar[10];
struct cnode    {
   int cc;                  //character code
   char *optr;              //->1st raster line
   int rw;                  //raster line width
   int bytes;               //bytes per raster line
   int lk;                  //left kern
   int rft;                 //rows from top
   int drc;                 //data row count
} clist[128];
struct cnode *a, *ptr;
struct cnode *fchar[128];



main(argc, argv)
   int argc; char **argv;    {
   if (argc < 2) exit();
   else if ((ip=open(argv[1],0)) < 0)    {
      printf("cannot open %s",argv[1]); exit();
   }
   init();
   while (getln()) putln();
   printf("closed\n"); exit();
}

init()    {
   register int i;
   if((pldev=open("/dev/rvp",1)) < 0)    {
      printf("cannot open plotter"); exit();
   }
   for(i=0;i<128;i++) fchar[i] = 0;
   n = fmark; for(i=0;i<128;i++) *n++ = -1;
   fp = 0; cfont("SAIL10"); //default font
```

```
}

int getln() { //rtn 1 if there's a line to
               //be plotted;otherwise, 0
   char k;
   t = tbuf;
   k = 0;
   while ( ((*t = getch()) != '\n') &&
             (*t != '\0') )   {
     if ( k++ == 89 ) { *t = '\n'; break; }
     t++;
     }
   if ( *t == '\0' ) return(0);
   else return(1);
}

putln()  { //plot as much as can fit in PAGEWTH
   register int h,i;
   roww = 0; pagewth = 216;
   if ( sl == 0) sl = 24;
   t = tbuf; p = pbuf;
   while (*t != '\n')    {
      if (*t == esc) { if (eschar()) break; }
      if (filchar()) break;
   }
   *p = '\n';
   if (t == tbuf) return; //null line in input file
   //check for room
   if (nroom(ht+(ht/10+1))) pagebreak();
   for(h=0;h<ht;h++)     {
      pl = &plbuf[sl]; *pl = 0; openbits = 8;
      ptr = fchar[*(p = pbuf)];
      while (*p != '\n')     {
         r = ptr->rw;
         if (ptr->drc)    {
            if(h >= ptr->rft && h < ptr->rft+ptr->drc)     {
               i = h - ptr->rft;
               lp = ptr->optr + i*ptr->bytes;
               while(r > 0)     {
                  shift(); r =- 8; }
            } else     {
               lp = zero;
               while(r > 0)     {
                  shift(); r =- 8;}
            }
         } else     {
            lp = zero;
            while(r > 0)     {
               shift(); r =- 8;}
         }
       ptr = fchar[*++p];
      }
      //plot one row raster line
      write(pldev,plbuf,roof(roww+sl*8));
   }
```

```
    //plot some white space
    for(h=0;h < ht/10+1;h++)
        write(oldev,zero,2);
    linecount =+ ht+(ht/10+1);
    sl = 0;
}

eschar()  { //esc- special characters
    int i, hi, space;
    char tt, *tb, *te;
    if (t == tbuf)    {
        if ((c = *++t) == 'f') { //font change
            n = fontname; t++;
            while ( (*n = *t++) != ' ' && *n != '\n' )
              n++;
            tt = *n;
            *n = '\0'; cfont(fontname);
            if ( (tt == '\n') || (*t == '\n'))   {
                t = tbuf; return(1);  }
        } else if (c == 's') { //need space
            n = ochar; t++;
            base = (*t == '0') ? 8 : 10 ;
            while (num(*n = *t)) {
              n++; t++;  }
            *n = '\0';
            hi = oct(ochar) * ht ;
            if (nroom(hi))  {
                pagebreak(); t = tbuf; return(1); }
            for (i=0;i<hi;i++)
                write(oldev,zero,2);
            linecount =+ hi ;
            t = tbuf; return(1);
        } else if (c == 'o'){ //no ascii equivalent
            n = ochar; t++;
            base = (*t == '0') ? 8 : 10;
            while (num((*n = *t)) )   {
              n++; t++;  }
            *n = '\0'; t--;
            *t = ((i = oct(ochar)) > -1 && i < 128 ) ? i
                                                : blank;
        } else if (c == 'c') { //center this line
            while (*++t == ' ') ;
            tb = t ;
            while (*++t != '\n') ;
            while (*--t == ' ') ;
            te = t; space = 0;
            for(t=tb; t<=te; t++)     {
              if (hdr[*t*2])
                  space =+ hdr[*t*2] & 0377;
              else if (hdr[040*2])     {
                  space =+ hdr[040*2] & 0377;
                  *t = 040;
              } else    {
                  printf("input error-- ");
                  printf("\tundefined character...%c\n",*t);
```

```
                flushh();
            }
        }
        space = (space%8 == 0) ? space/8 : space/8+1;
        sl = 132 - space/2;
        if (sl < 24)    {
          printf("input error-- ");
          printf("\ttoo many characters to center\n");
          flushh();
       }
       for(i=0;i<sl;i++) plbuf[i] = 0;
       t = tb;
     } else if (c == 'p')  {
         if ( (c = *++t) == 'g')  {//pgbreak
            pagebreak(); t = tbuf; return(1);  }
         else if (c == 'p')  {//paragraph
                for(i=0;i<ht;i++)
                   write(pldev,zero,2);
                sl = 24 + (24 * ht/120);
                pagewth = pagewth - (24 * ht/120);
                t = tbuf; return(1);
                }
              else  {
                printf("invalid character folowing ");
                printf("'ESCp'..");
                exit();
                }
     } else      {
         printf("input error- ");
         printf("\tinvalid escape character... %c",c);
         flushh();
     }
  } else if ((c = *++t) == 'o') { //no ascii equiv
     n = ochar; t++;
     base = (*t == '0') ? 8 : 10 ;
     while (num((*n = *t)) )  {
        n++; t++; }
     *n = '\0'; t--;
     *t = ((i=oct(ochar)) > -1 && i < 128) ? i
                                         : blank;
  } else if (c == 'f') {//no font chg allowed here
     printf("change fonts at line head only ");
     flushh();
  } else     {
     printf("input error- ");
     printf("\tinvalid escape character ( %c )\n",c);
     printf("\tembedded within text...\n");
     flushh();
  }
  return(0);
}

int filchar()    { //move chars from tbuf to pbuf until
                   //PAGEWTH exceeded, replace nonexistent
                   //chars with blank; ow, exit
```

```
	register int i;
	infont = *t;
	if (hdr[infont*2])	{
		if (fchar[infont] == 0)	{
			getdef();
			if (roww+a->rw <= pagewth*8)
				roww =+ a->rw;
			else {*p = '\n'; return(1);}
		} else if (roww+fchar[infont]->rw <= pagewth*8)
				roww =+ fchar[infont]->rw;
		else {*p = '\n'; return(1);}
	} else if (hdr[(infont=blank)*2])	{
		*t = blank;
		if (fchar[infont] == 0)	{
			getdef();
			if (roww+a->rw <= pagewth*8)
				roww =+ a->rw;
			else {*o = '\n'; return(1);}
		} else if (roww+fchar[infont]->rw <= pagewth*8)
				roww =+ fchar[infont]->rw;
		else {*p = '\n'; return(1);}
	} else	{
		printf("character '%3o' not defined in %s",*t,
			header);
		flushh();
	}
	*o++ = *t++;
	return(0);
}

cfont(q)
	char *q; { //q points to new font name
	register int i;
	if (fp)	{
		printf("closed\n"); close(fp);
	}
	for(i=16;(header[i] = *q++) != '\0';i++)	;
	if((fp=open(header,0)) < 0)	{
		printf("cannot open %s",header); exit();
	}
	printf("%s opened....",header);
	dealloc(nodeptr); nodeptr = 0;
	for(i=0;i<128;i++) fchar[i] = 0;
	read(fp,hdr,512); read(fp,&ht,2);
	read(fp,&maxw,2); read(fp,&lht,2);
	if(check())	{
		printf("%s bad font file",header);
		exit();
	}
}

dealloc(x)
	int x;	{ //free in reverse order
			// of allocation
	while (x)
```

```
        if(fchar[fmark[--x]]->optr)
            free(fchar[fmark[x]]->optr);
}

pagebreak() { //page eject
    int i;
    char err;
    err = cvers(pldev,020);
    if ( err == -1 ) {
    printf("invalid filedes in pagebreak\n");
    exit();
    }
    for (i=0;i<TOP;i++) write(pldev,zero,2);
    linecount = TOP;
}

getdef()     {
    int blkc,bytc; register i;
    blkc = (hdr[infont*2]&0177400) >> 8;
    blkc =& 0377;
    bytc = hdr[infont*2+1];
    if(blkc)     {
        seek(fp,blkc,3); seek(fp,bytc,1); }
    else seek(fp,bytc,0);
    getnode();
    a->cc = infont;
    read(fp,&a->rw,2);
    read(fp,&a->lk,2); read(fp,&a->rft,2);
    read(fp,&a->drc,2);
    a->bytes = (a->rw%8 == 0) ? a->rw/8 : a->rw/8+1;
    if(a->drc)     {
        if((i=a->optr=alloc(a->drc*a->bytes)) < 0) {
            dealloc(nodeptr-1);
             getdef(); return;
        }
        read(fp,a->optr,a->drc*a->bytes);
    }
    in = 0;
    for(i=0;i<nodeptr;i++)     {
        if(fmark[i] == infont) in++;
    }
    if(in == 0) fmark[nodeptr-1] = infont;
}

getnode()     {
    if(nodeptr > 127)     {
        printf("overflow"); exit();}
    a = fchar[infont] = &clist[nodeptr++];
    a->optr = 0;
}

int roof(x)
    int x;     {
    x = (x%8 == 0) ? x/8 : x/8 + 1;
    if(x%2 == 0) return(x);
```

```
    if(x == 263) return(262);
    *++pl = 0; return(++x);
}

int check()     {
    if(ht < 0 || maxw < 0 || lht < 0 ||
       ht > 120 || maxw > 256 || lht > ht) return(1);
    else return(0);
}

int nroom(x)
    int x;     {
    if(linecount + x > PAGEHT) return(1);
    else return(0);
}

shift()     {
    int tb;
    tb = *lp; tb =& 0377; tb =<< openbits;
    if(r > 7)     {
       *pl++ =| (tb & 0177400) >> 8;
       *pl =& 0; *pl =| tb & 0377;
    } else     {
       if(r <= openbits)     {
          *pl =| (tb & 0177400) >> 8;
          openbits =- r;
       } else     {
          *pl++ =| (tb & 0177400) >> 8;
          *pl =& 0; *pl =| tb & 0377;
          openbits = 8-(r-openbits);
       }
    }
    lp++;
}

int oct(cp)
    char *cp;     {
    int i; i = 0;
    base = (*cp == '0') ? 8 : 10;
    while (num(*cp) && *cp != '\0')
       i = i*base + *cp++ - '0';
    return(i);
}

int num(cp)
    char cp;     {
    if(base == 10 && (cp >= '0' && cp <= '9')) return(1);
    if(base == 8 && (cp >= '0' && cp <= '7')) return(1);
    if (cp == '8' || cp == '9')   {
      printf("input error-- ");
      printf("\timproper octal number...%d",cp);
      while (*t != '\n') outchar(*t++);
      exit();
      }
    else return(0);
```

```
}

getch() {
   char tt,s;
   s = read(ip,&tt,1);
   if ( s == 0 ) return('\0');
   else return(tt);
}

flushh() { //print bad input line and exit
   while (*t != '\n') putchar(*t++);
   exit();
   }
```

APPENDIX E. HERSHEY FONTS AVAILABLE

The fonts listed below are currently available on "/.fonts.01/hershey" in vector form. They are used in this form by "drawhf", and they are converted to dot matrix form from vectors by "makehf". The 14 fonts available are :

HSR -- Simplex Roman

HSG -- " Greek

HSS -- " Script

HDR -- Duplex Roman

HCR -- Complex Roman

HCG -- " Greek

HCS -- " Script

HCI -- " Italic

HCC -- " Cyrillic

HTR -- Triplex Roman

HTI -- " Italic

HGE -- Gothic English

HGG -- " German

HGI -- " Italian

The following pages provide a display of each font and its character set. The last page of this appendix contains a quotation written in each font for comparison and contrast of the fonts.

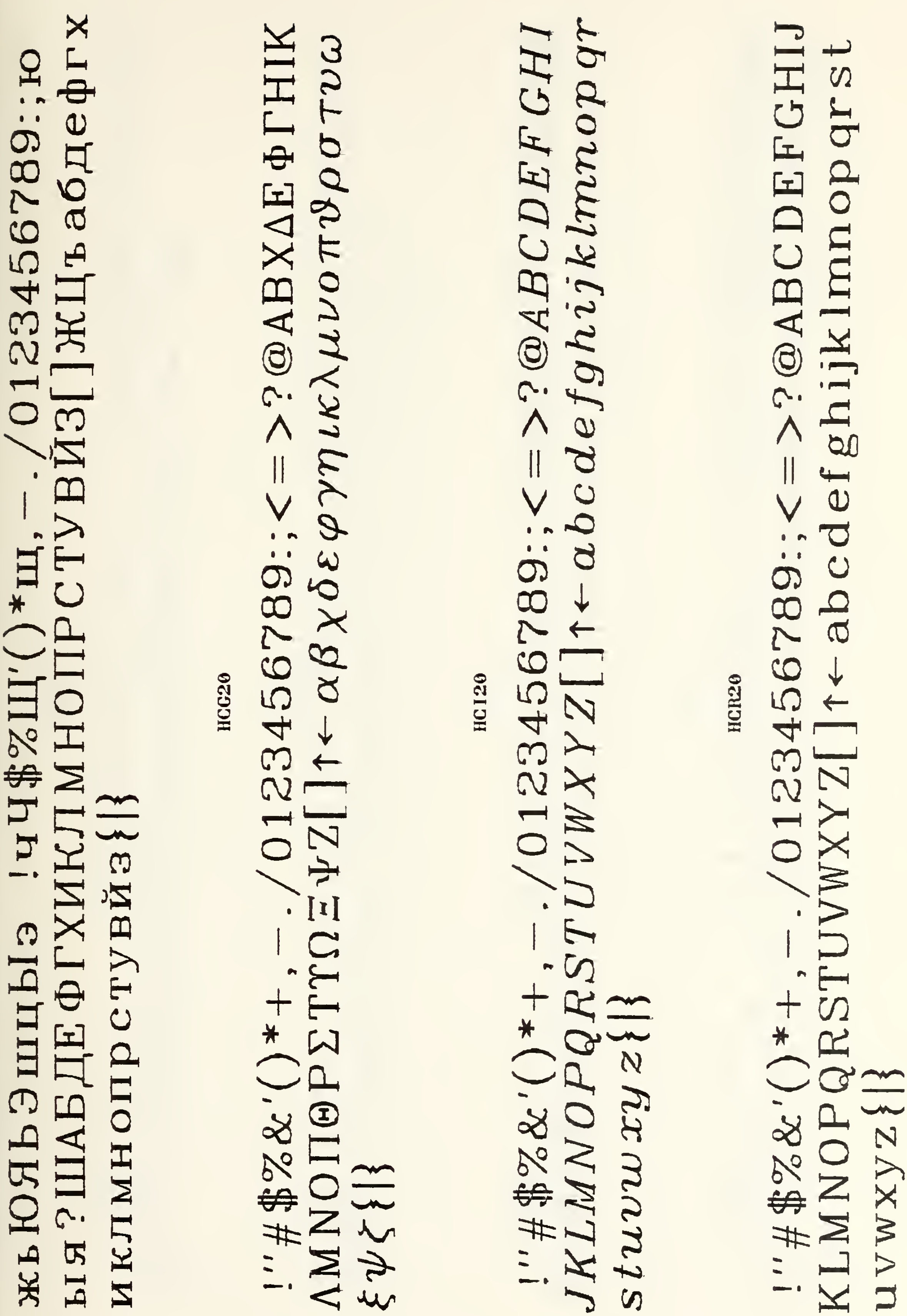

FIGURE E-1. Hershey Font Examples

HCS20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

HDR20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKLMNO
PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz.

HGE20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKLM
NOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

HGG20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrſtuvwxyz
sßtz

FIGURE E-1. (Continued)

HGI20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJK
LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx
yz

HSG20

!"#$&'()*+,-./0123456789:;=?ΑΒΧΔΕΦΓΗΙΚΛΜΝΟΠ
ΘΡΣΤΥΩΞΨΖαβχδεφγηικλμνοπϑρστυωξψζ|

HSR20

!"#$&'()*+,-./0123456789:;=?ABCDEFGHIJKLMNO
PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz|

HSS20

!"#$&'()*+,-./0123456789:;=?ABCDEFGHIJKLMN
OPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz|

FIGURE E-1. (Continued)

HTI20

!"$&'()+,-./0123456789:;=?ABCDEFGHIJKL*
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
xyz

HTR20

!"$&'()*+,-./0123456789:;=?ABCDEFGHIJKLMNO
PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

FIGURE E-1. (Continued)

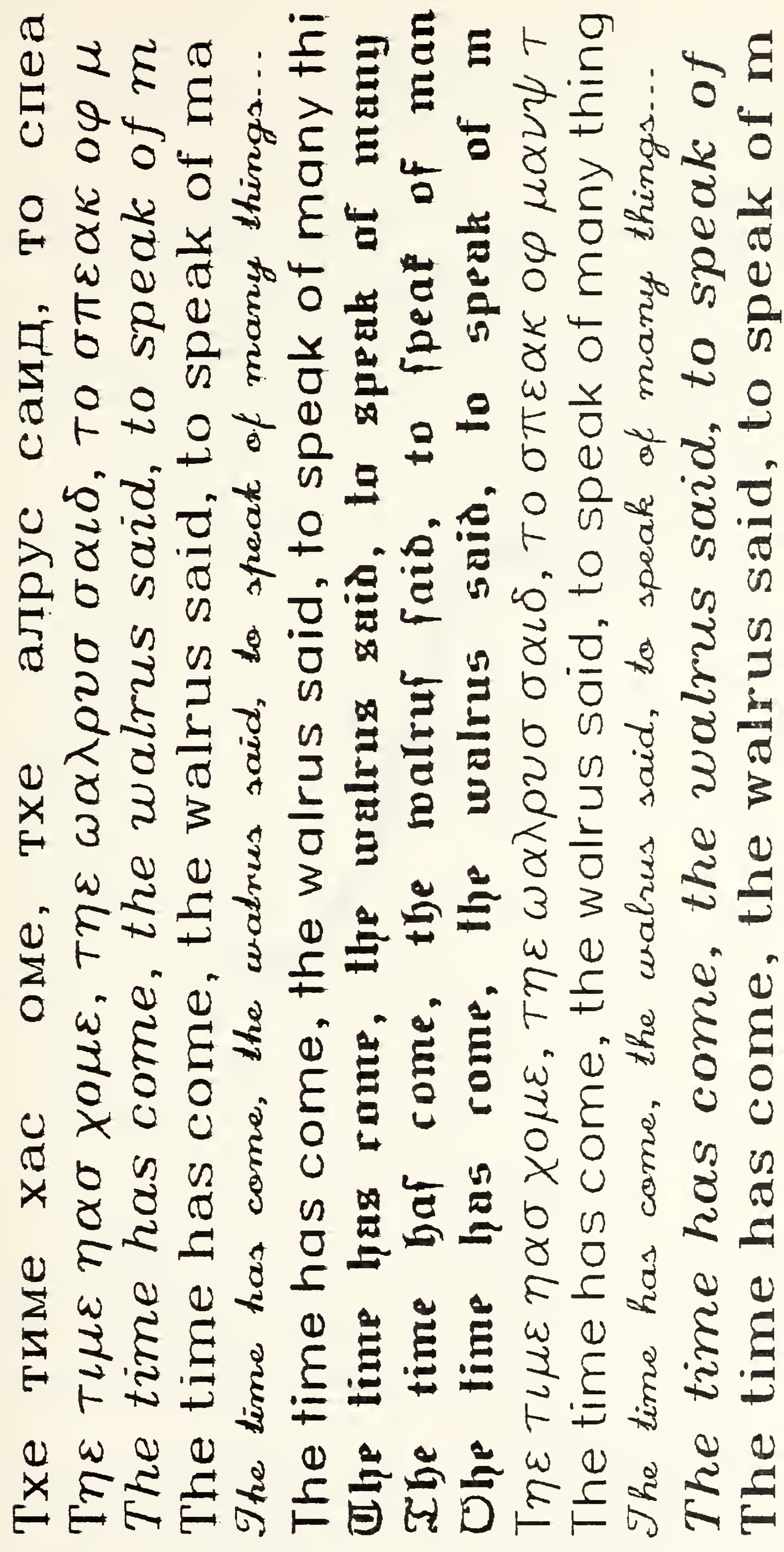

FIGURE E-2. Hershey Font Comparisons

APPENDIX F. FONT/CHARACTER DIMENSIONS

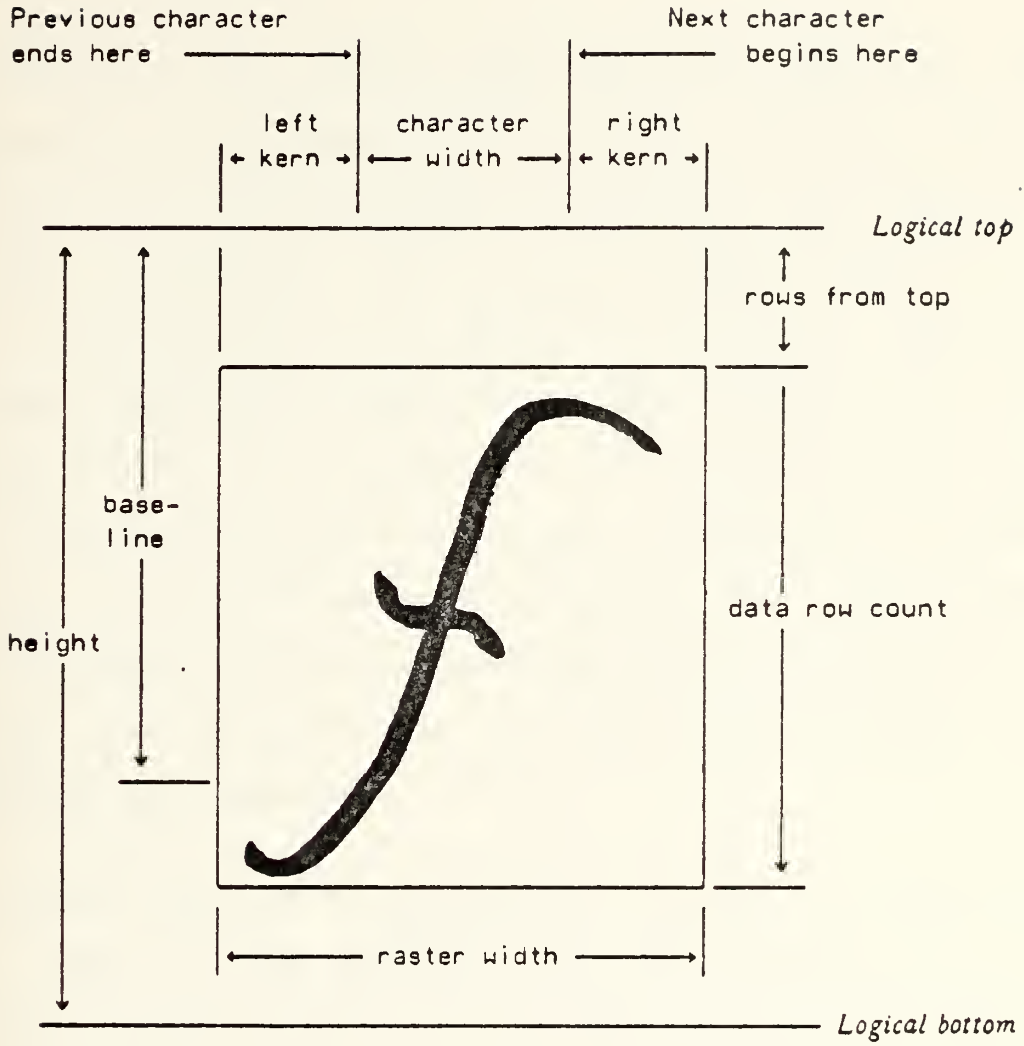

FIGURE F-1. Font dimensions

This figure, taken from Reference 4, displays the dimensions of fonts and characters that must be taken into account when setting type by computer.

The most important characteristics of a font are its height, the width of its widest character, and its logical height. The values for height and logical height remain con-

stant throughout a font and are the real measure of compatibility among fonts, i.e., in creating a new font, characters from fonts of differing heights or logical heights cannot be mixed.

Character width, raster width, and left kern are the characteristic dimensions of characters. The right kern is not listed, but may be computed if desired. There are two additional dimensions which play an important part in the stored representation of the digitized character. These are rows-from-top(rft), a count of the blank raster lines from the logical top of the character to the first non-blank row, and the data-row-count(drc), a count of the number of raster lines that contain character information. The font height minus the sum of rft plus drc provides the number of blank lines that must be added after the last nonblank raster line to complete the character.

Another important characteristic of a font is the baseline. This is the distance from the logical top of the character to the imaginary line on which the row of characters rests, although some characters may extend below this line. All characters in a given font file have the same height and baseline.

"Kerning" is a characteristic which occurs only when a font has a non-zero left or right kern, so that the character width is smaller than the raster width. Kerning allows the computer to set some characters closer to others to

avoid leaving what appears to be too much white space between characters; of course, the computer must first make some checks to ensure that no character overlays occur. When setting a kerned font, the typesetting program will space ahead according to the character width and not the raster width. Kerning occurs in only two of the SAIL fonts. Neither the current version of the virtual typesetter nor the typesetting program described in this guide deal with kerning, but font files and programs provide a place for the left kern so that the concept may be fully implemented later without reorganizing font file structure.

APPENDIX G. THE 'SAIL' FONTS

A. 'SAIL' FONTS AVAILABLE

All of the digitized fonts currently available are listed below by typeface and style. Each of these is located on directory "/.fonts.01/font" :

BDJ8	--	8	point	Bodoni	Mathematical
BDR10	--	10	"	"	Roman
BDI10	--	10	"	"	Italic
BDJ10	--	10	"	"	Mathematical
BDR10X	--	10	"	"	Bold
BDR12	--	12	"	"	Roman
BDI12	--	12	"	"	Italic
BDB12	--	12	"	"	Bold
BDR15	--	15	"	"	Roman
BDI15	--	15	"	"	Italic
BDR25	--	25	"	"	Roman
NONS	--	10	"	Nonie	Roman
NONSI	--	10	"	"	Italic
NONSB	--	10	"	"	Bold
NONSBI	--	10	"	"	Bold Italic
NONM	--	12	"	"	Roman
NONMI	--	12	"	"	Italic
NONMB	--	12	"	"	Bold

NONMBI	--	12 point	Nonie	Bold Italic
NONL	--	14 "	"	Roman
NONLI	--	14 "	"	Italic
NONLB	--	14 "	"	Bold
NONLBI	--	14 "	"	Bold Italic
SAIL10	--	10 "	Delegate (similar to IBM Selectric)	
SHD15	--	15 "	Shadow	
SIGN22	--	22 "	Sign	
SIGN41	--	41 "	Sign	
GRFX10	--	10 "	Graphics	
GRFX14	--	14 "	Graphics	
MATH10	--	10 "	Math	
MATH13	--	13 "	Math	
MATH15	--	15 "	Math	
MATH20	--	20 "	Math	
MATH21	--	21 "	Math	

B. 'SAIL' CHARACTER CODES

The SAIL character set and corresponding octal codes are found on the next page, with the ASCII character set. A blank indicates that no character exists for that code.

	0	1	2	3	4	5	6	7
000	NUL	↓	α	β	∧	¬	ϵ	π
010	λ	HT	LF	VT	FF	CR	∞	∂
020	⊂	⊃	∩	∪	∀	∃	⊗	↔
030	_	→	~	≠	≤	≥	≡	∨
040	SP	!	"	#	$	%	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	↑	←
140	'	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	ESC	}	BS

FIGURE G-1. SAIL Character Set

	0	1	2	3	4	5	6	7
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
010	BS	HT	NL	VT	NP	CR	SO	SI
020	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
030	CAN	EM	SUB	ESC	FS	GS	RS	US
040	SP	!	"	#	$	$	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
120	X	Y	Z	[	\	]	↑	←
130	'	a	b	c	d	e	f	g
140	h	i	j	k	l	m	n	o
150	p	q	r	s	t	u	v	w
160	x	y	z	{	|	`	}	DEL

FIGURE G-2. ASCII Character Set

APPENDIX H. FINDING A FONT.

A. FONT LOCATION

All of the fonts and font manipulation routines are located on a mountable file called "fonts.01". To access this file, the following procedure is necessary after logging in:

```
% fsmount fonts.01
    /dev/fonts.01
    spcl  aaaa
    files bbbb
    large cccc
    direc dddd
    indir eeee
    used  ffff
    free  gggg
    /dev/fonts.01
    /dev/fonts.01 mounted to directory /.fonts.01
%
```

A complete description of the directory configuration is given on the next page. Detailed explanations of the font editor "edf" and the Hershey conversion program "makehf" are given in Appendix A and in Chapter III respectively; brief descriptions of these programs are also located with the program listings in Appendixes A and B respectively. The source programs, a copy of "A User's Guide For Font Manipulation at the Naval Postgraduate School", and instructions for acquiring both are contained on "fonts.02", another mountable file which is mounted and accessed in the same manner as "fonts.01".

B. MOUNTABLE FILE DESCRIPTION

The following diagrams describe the directory configurations of "fonts.01" and "fonts.02". A "d" in a branch of the tree indicates that the next name is a directory.

1. Fonts.01

```
                        /.fonts.01
                             |
                             |
      ---------------------------------------------
      |         |          |          |           |
      |         d          d          d           d
     edf        |          |          |           |
   HFONT       font     hershey    hftools       sail
   makehf       |          |          |           |
   prfont      SAIL     Hershey    drawhf      transfile
   signmkr     fonts     fonts     cnvrt       listfont
                                   mkvec
```

The file called "HFONT" normally contains the most recently created Hershey font, unless it was specifically written to another directory. This process is explained in Appendix B. All other program names that pertain to Hershey fonts are explained elsewhere in this report. "Transfile" and "listfont" pertain to the conversion of SAIL fonts for NPS use and are discussed in Reference 6.

2. Fonts.02

```
                /.fonts.02
                     :
                     :
       --------------------------------------
       :             :              :        :
       d             d              d        d
       :             :              :        :
    hershey        common          sail      :
       :             :              :        :
    makehf.c      signmkr.c      transfile.c :
    drawhf.c      orfont.c       listfont.c  :
    cnvrt.c       edf.c                      :
    mkvec.c                                  :
                                          userman
                                             :
                                         userguide
                                      userenclosures
                                         printman
```

To obtain a personal copy of the User's Guide, mount both "fonts.01" and "fonts.02" and type "sh /.fonts.02/userman/printman". The manual will be directed to the line printer and the figures will be plotted on the VERSATEC plotter/printer.

LIST OF REFERENCES

1. Barksdale, G.L. Jr. and Meyer, W.B., TPS, A Text Processing System, Naval Postgraduate School, 1976.

2. Barnett, Michael P., Computer Typesetting, The M.I.T. Press, 1965.

3. Berg, N. Edward, Electronic Composition, Graphic Arts Technical Foundation, pp. 6:1-6:30, 1975.

4. Earnest, Les, Find A Font, Stanford University Artificial Intelligence Laboratory, 1976.

5. Hattery, Lowell H. and Bush, George P., eds., Automation And Electronics In Publishing, Spartan Books, pp 9-17, 1965.

6. McCord, B.S., An Enhancement Of The Computer Typesetting Capability Of UNIX, M.S. Thesis, Naval Postgraduate School, Monterey, California, 1977.

7. Ossanna, Joseph F., The NROFF User's Manual, Bell Telephone Laboratories, Incorporated, 1974.

8. Ossanna, Joseph F., The TROFF User's Manual, Bell Telephone Laboratories, Incorporated, 1974.

9. RAMTEK GX-100A Programming Manual, Ramtek Corporation, Appendix B, 1974.

10. Naval Postgraduate School Technical Memorandum, Elementary Magnetic Tape Usage Under OS/MVT At NPS, by Sharon D. Raney, pp. 6-8, November 1976.

11. Naval Postgraduate School Technical Note 0211-08, Procedures For Converting 7-Track Magnetic Tapes To 9-Track Magnetic Tapes, 3rd ed., by Sharon D. Raney, pp 30-32, February 1973.

12. Reference Manual--Graphics Display Unit, Vector General Incorporated, p. 2-12, 1973.

13. System Reference Manual, Hughes Aircraft Company, Industrial Products Division Conographic Products, pp. 13-15, 41-45, 1974.

14. Terminal Control System, Tektronix Incorporated, Infor-

mation Display Division, pp. 1-52, 1974.

15. Thompson, K. and Richie, D.M., The UNIX Programmer's Manual, 6th ed., Bell Telephone Laboratories, Incorporated, Chapter I, 1975.

16. Wolcott, Norman M. and Hilsenrath, Joseph, A CONTRIBUTION TO COMPUTER TYPESETTING TECHNIQUES: Tables of Coordinates for Hershey's Repertory of Occidental Type Fonts and Graphic Symbols, National Bureau of Standards (Special Publication 424), pp. 1-15, 1976.

17. Hamming, R.W., Numerical Methods For Scientists And Engineers, McGraw-Hill, Incorporated, 1973.

18. Naval Postgraduate School Technical Report NPS52Ba77061, A User's Guide To Font Creation And Manipulation At The Naval Postgraduate School, G.L. Barksdale, Jr., P.M. Doyle, B.S. McCord, 1977.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexanderia, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Professor Gerald L. Barksdale, Jr., Code 52Ba Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LCDR Stephen T. Holl, USN, Code 52Hl Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	LT Patrick M. Doyle, USN 58 Bluff Street Dubuque, Iowa 52001	1

Thesis
D7215 Doyle
c.1
170359
An adaptation of the Hershey digitized character set for use in computer graphics and typesetting.