

Honours Project
University of Canterbury
Computer Science Department
1983

Design and Implementation of a
TROFF postprocessor for Typesetting
on a Printronix 300

written by
Anthony M. Verdellen

supervised by
Dr. B. J. McKenzie

Table of Contents

	page
1. Background	1
2. Project Definition	6
2.1 TROFF	7
2.2 Printronix 300	8
2.3 The Hershey Fonts	10
3. Decoding TROFF Output	13
4. Printronix Output Routines	16
5. Character Fonts	17
5.1 Font Selection	17
5.1.1 Times Roman Font	18
5.1.2 Times Italic Font	18
5.1.3 Times Bold Font	19
5.1.4 Special Mathematical Font	19
5.2 Filling Algorithm	20
5.3 Font Scaling and Storage	24
6. Phototypesetter-Simulator Postprocessor	27
7. Conclusion	29
8. References	30

Appendices

Appendix A - GIMMS System Fonts	32
Appendix B - Decoding Program	37
Appendix C - Plotting Routines	43
Appendix D - Roman Font	47
Appendix E - Italic Font	49
Appendix F - Bold Font	51
Appendix G - Greek Font	52
Appendix H - Filling Routines	54

1. Background

Almost every computer system will have, within its set of peripheral devices, at least one device that produces output in a human readable form. Many of these output devices are very limited in the form of output that they produce. Devices like VDUs, matrix printers and line printers have limited character sets, fixed size characters and will not allow characters to appear at any arbitrary position on a page. One output device that does not suffer from these limitations is the phototypesetter.

A phototypesetter forms characters on photographic paper by passing light through a character font stencil, through a series of lenses and onto the photographic paper. Because the optical system of the phototypesetter can direct light anywhere on the photographic paper characters can appear at any position on the output page that the user desires. The lens system is under the control of the user and therefore by selecting appropriate lenses the user can produce characters of varying size from a single character font stencil. With most phototypesetters a number of character fonts can be mounted at one time and as a result the user can select any of these fonts at will. Because the photographic paper is a continuous roll any page length can be catered for. The ability of the phototypesetter to position characters anywhere on an output page means that characters may overlap one another and this gives the user the power to create new characters by overprinting a number of characters (eg the Greek letter phi can be formed by overprinting 'O' and '/').

A phototypesetter is a very powerful and versatile document formatting tool and indeed many published items, such as articles in technical journals, are produced with the aid of

phototypesetting techniques as opposed to the more traditional mechanical typesetting. Despite its power the phototypesetter is very hard to use because the commands used to control it are at a very low level. The user must, for example, specify explicitly all the vertical and horizontal positioning concerned with indenting, justification and spacing. Document formatting with low-level phototypesetter commands is like writing a major piece of software in machine language. The need for some form of intermediary between the user and the phototypesetter prompted the development of a phototypesetting text processor called TROFF [1] [2] [9].

TROFF was developed by J. F. Ossanna at Bell Laboratories during the early to mid-1970s. It was written for the Unix operating system in the programming language C. TROFF accepts an ordinary text file interspersed with document formatting commands and produces output to control a Graphic Systems phototypesetter. Users of TROFF have full use of all the features of the phototypesetter but can also allow some facilities such as centering, indenting, spacing and justification to be handled automatically. Special features called traps and diversions allow the user to write TROFF code to deal with page headers and footers and also footnotes. It would be impossible to give a complete description of TROFF in this document, it is sufficient to say that TROFF is a powerful and flexible phototypesetting document formatter.

A major problem with TROFF is that, like using the phototypesetter directly, operations must be specified at a level of detail and in a form that is too difficult for most people to use effectively. Document formatting with TROFF is similar to writing a major piece of software in assembly language. Because of the deficiencies of TROFF various macro packages and

preprocessors have been developed to enable the user to work at a higher level than is possible with raw TROFF.

A number of macro packages are available which utilise the macro facility of TROFF to provide some high-level pseudo-commands in addition to the usual TROFF commands. The -ms macro package provides macro commands to enable some features to be taken care of automatically, including headers, footers, section headings, multiple columns and indenting. The use of macro packages provides the user with a number of high-level commands that help to avoid some of the detail of pure TROFF. Using the -ms macro package to perform document formatting is comparable to writing major software in assembly language with the use of a number of predefined system routines. TROFF preprocessors on the other hand present the user with a complete formatting language that is a superset of TROFF.

$$\frac{1}{2\pi} \int_{-\infty}^{\sqrt{y}} \left(\sum_{k=1}^n \sin^2 x_k(t) \right) (f(t) + g(t)) dt$$

```

.EQ
1 over (2 pi) int from {- inf} to {sqrt y}
left ( sum from k=1 to n sin sup 2 x sub k (t) right )
left ( f(t) + g(t) right ) dt
.EN

```

Figure 1

EQN[3] is a TROFF preprocessor that provides a high-level declarative language for specifying mathematical equations within TROFF-prepared documents. Figure 1 shows an example of a

mathematical equation followed by its EQN definition. From this example it can be seen that the processor permits mathematical equations to be specified in a logical and orthogonal manner. EQN accepts normal TROFF input which contains EQN definitions delimited by .EQ and .EN and produces an output file suitable for input to TROFF.

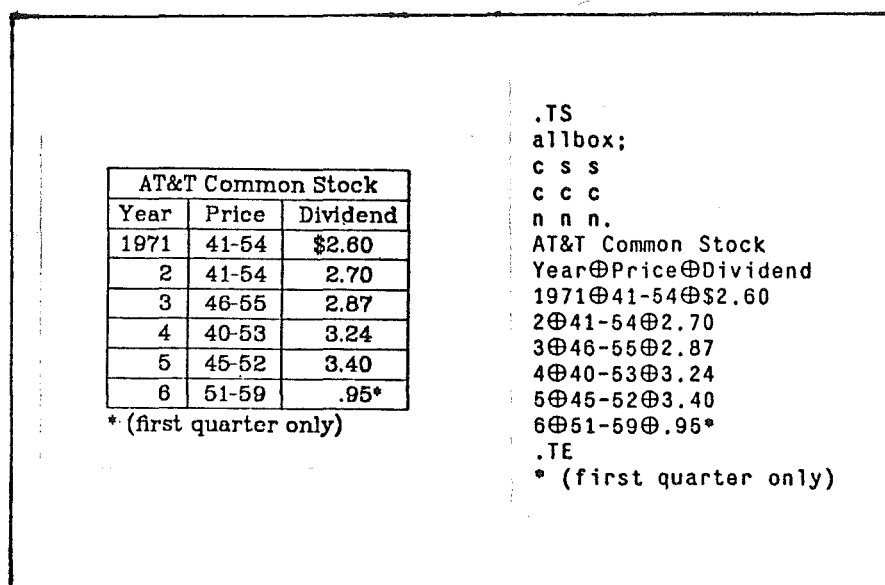


Figure 2

TBL[4] is another TROFF preprocessor. TBL allows the user to format tables in much the same way that EQN allows the user to format mathematical equations. Figure 2 shows a sample table with its corresponding TROFF definition. TROFF commands may be included within TBL definitions thus providing a high degree of flexibility when describing tables. As with EQN, TBL produces an output file containing text and TROFF commands.

Because the TROFF preprocessors have a language definition

independent of TROFF the user can work at a fairly high level of abstraction and does not have to get his/her hands dirty with TROFF. The preprocessors designed for TROFF make document formatting comparable to writing software in a high level language.

It is obvious that TROFF, along with its macro packages and preprocessors, is a powerful phototypesetting tool that would be an asset to any computer system that has the need to provide document formatting facilities.

2. Project Definition

The background material gave a brief description of the document formatting facilities available with a TROFF implementation and the associated Graphic Systems phototypesetter. The University of Canterbury has available on the Prime 750 computer system a working TROFF implementation (including macro packages and preprocessors) but does not have a phototypesetter on which to produce output. To utilise the potential document formatting facilities provided by TROFF it is necessary to produce output on some other device. The only suitable hard copy graphics output device available on the Prime is the Printronix 300 line printer. The Printronix 300 produces output on normal computer output paper but has the capability to treat an output page as a 792 x 792 matrix of (slightly overlapping) dots.

The aim of this project is therefore to design and implement a program that accepts the output from the TROFF package and produces the equivalent of phototypesetter output on the Printronix. Since the Printronix when operating in graphics mode simply plots points the program must be responsible for the creation and the scaling of the character images to be plotted.

For the purposes of character generation a file on the Prime containing the Hershey fonts was made available. The Hershey fonts are a series of character fonts designed by A. V. Hershey where each character is described by pen movements on a 200 x 200 grid.

To realize the aim of this project it is necessary to combine three (presently unrelated) components, namely TROFF, the Printronix 300 and the Hershey fonts. What follows is a brief description of the information known about each at the time when this project was started.

2.1 TROFF

For the purposes of this project the only real interest in TROFF is the output that it produces. It is necessary to be able to decode the output from TROFF to determine the actions the phototypesetter is expected to perform in response. Decoding of the TROFF output will make it possible to simulate the actions of the phototypesetter on the Printronix.

Unfortunately the only documentation available for TROFF was a user manual[1] and a number of tutorials[2][9]. These documents provided excellent instructions for users of TROFF but gave no hint as to how the formatter worked. The only useful information in this area was a listing of TROFF and a listing of TCAT (a phototypesetter-simulator preprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014).

There was virtually no information about the phototypesetter itself apart from a brief description of some operating characteristics (page width etc.) in the user manual. Apart from this the only information available about the phototypesetter was the brand name 'Graphic Systems'.

2.2 Printronix 300

Part of the user documentation[10] for the Printronix 300 gave a full description of the use of the graphics mode (called Plot Mode by the manufacturers).

Plot Mode allows the user to treat each 13.2 in. x 11 in. output page as a 792 x 792 matrix of dots (60 dots per horizontal inch and 72 dots per vertical inch). More specifically a page is treated as 792 rows each consisting of 792 dots.

To specify a row of dots the user first sends a byte with octal value 005. This byte is followed by a number of data bytes each representing 6 dot positions. Finally a line feed character is sent to mark the end of the dot row. Of the bytes that specify dot positions the first byte represents the leftmost 6 dot positions, the second byte represents the next 6 dot positions to the right, and so on. Each of these bytes must have the 7th bit set while the low order 6 bits carry information about dot positions. The dot positions from left to right are specified by the low order 6 bits from lowest to highest respectively. A set bit corresponds to a dot and an unset bit corresponds to a vacant dot position.

To move to a new page the form feed character is used. Figure 3 gives an example of a dot pattern and the corresponding input sequence.

The above information obtained from the Printronix 300 user manual is sufficient for the needs of this project. The appendices contain a number of examples of Plot Mode output.

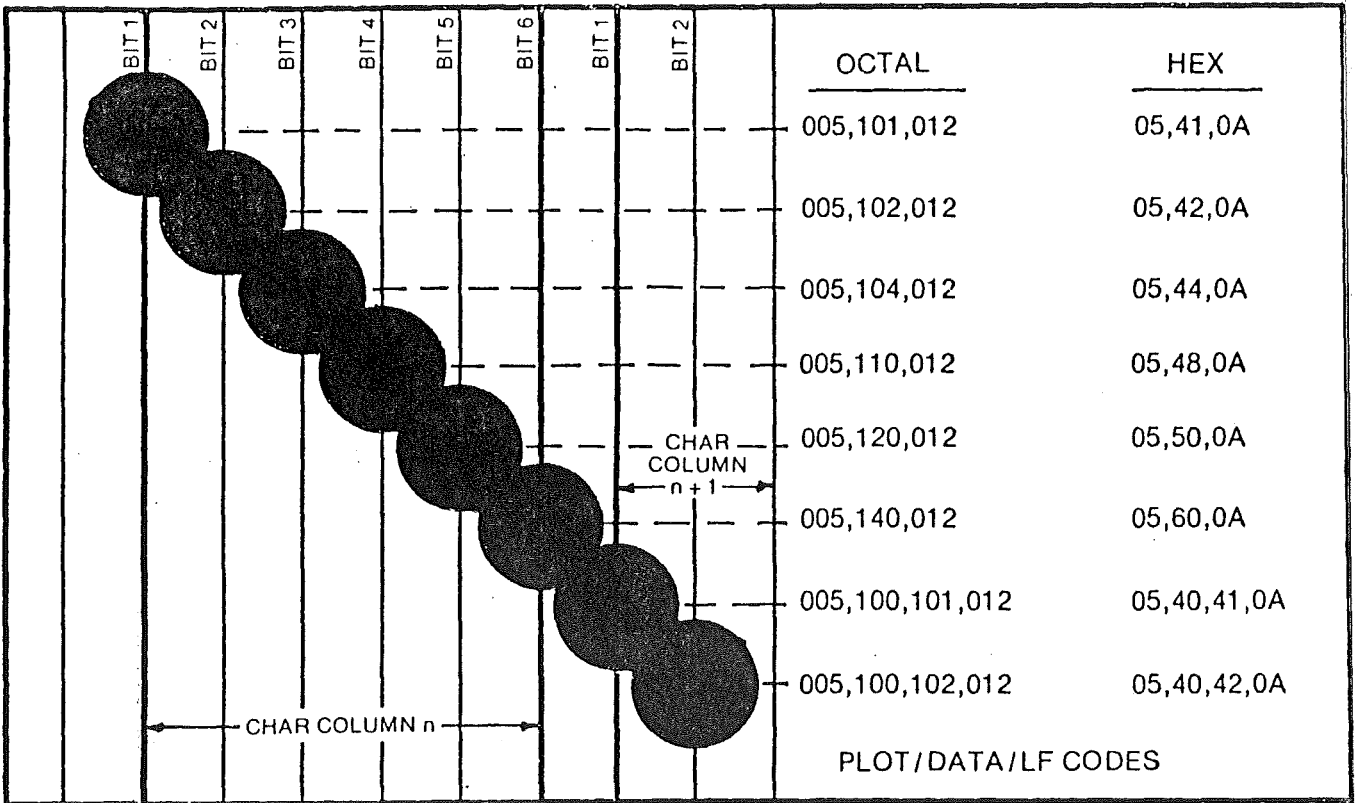


Figure 3

2.3 The Hershey Fonts

The Hershey fonts[5][11] are described on the Prime by a very large file. The fonts supplied are as follows.

font number	description
-----	-----
2 and 3	Special characters, simplex
4 and 5	Special characters, complex
6 and 7	Special characters, small complex
11	Sanserif, cartographic, wide simplex
12	Sanserif, simplex
13	Serif, complex
14	Serif, small complex
15	Sanserif, duplex
16	Serif, triplex
17	Block letters (from L. Johnson)
23	Italic, complex
24	Italic, small complex
26	Italic, triplex
32	Script, simplex
33	Script, complex
41	Greek, cartographic
42	Greek, simplex
43	Greek, complex
44	Greek, small complex
48	Cyrillic, complex
51	Gothic, German
52	Gothic, English
53	Gothic, Italian
60	OCR type
61	Helvetica Medium
62	Clarendon Bold
63	Hershey 15
64	Stephenson Blake

Note: Small complex differs from complex in that complex characters are created with smaller strokes and thus they have a smoother appearance when printed.

Appendix A gives a complete pictorial representation of all the Hershey fonts.

Each Hershey character is described in terms of pen movements on a 200 x 200 grid. Only the outline of the character is given and therefore areas of the character that must be filled need to be recognised by the user and taken care of.

3. Decoding TROFF Output

In order to simulate the actions of the phototypesetter it is necessary to be able to decode these actions from the coded output produced by TROFF. As stated earlier the only applicable documents available were the source listing of TROFF and TCAT. In addition to this it was possible to execute TROFF on an input file and look at the output produced (useful for empirical testing).

Inspection of TROFF revealed that it was a very large and complex program. It was possible to isolate a number of routines involved with producing output for the phototypesetter but the code gave no indication of the phototypesetter actions involved. It was however apparent that output was in a bitwise manner with all eight bits containing information.

The TCAT program was a little more useful in that it too must decode the output from TROFF. A problem with the TCAT program was that the code involved with the interpreting of TROFF output and the code concerned with the control of the Tektronix 4014 was so intermixed that not knowing how the Tektronix was operated made distinguishing between the two very difficult. Along with the problem of scarcity of comments (especially those describing the phototypesetter commands corresponding to the data being read) the task of decoding the phototypesetter commands seemed daunting.

Faced with these problems another approach seemed to be in order. An attempt was made to find out directly what the phototypesetter codes meant by getting in contact with the manufacturers or a user of the machine. Investigation revealed that there were no current dealers in Graphic Systems

phototypesetters in New Zealand and that the closest machine was in Ashburton. A trip to see this phototypesetter and to study its manuals seemed appropriate.

Study of the phototypesetter made the task of decoding TROFF output much easier (and indeed it may have been impossible without the knowledge this study brought). It was obvious that there were only a small number of basic operations that the phototypesetter performed and that these must match one-for-one with the codes produced by TROFF. The basic phototypesetter operations are outlined below.

- (i) vertical motion (both up and down),
- (ii) horizontal motion (both left and right),
- (iii) character size changes,
- (iv) font changes (selection of one of the four mounted fonts),
- (v) upper/lower case changes,
- (vi) write a character at current position,
- (vii) reset.

Unfortunately the phototypesetter studied was controlled by paper tape and hence the codes did not match those produced by TROFF.

By comparing the actions known to be executed by the phototypesetter and the method of decoding used by TCAT it seemed possible to deduce the way in which the phototypesetter commands were encoded by TROFF. Once deductions were made they could be tested by applying TROFF to a test file and comparing the actual result with the expected result. After much trial and error the code was broken and the task of writing a program to do the decoding automatically, began.

At this point it was necessary to decide which programming language was to be used for any programs written as part of this project. The language chosen was C and the reasons for this decision follow. C was a likely choice because all the TROFF software is written in C and this would make the two related programs compatible. Because of the low-level nature of much of the required processing the need for efficiency suggested that a low-level language such as C or BCPL was required. Major portions of the TCAT program were directly applicable to the decoding program that was to be written and using C as the programming language meant that sections of the code could be used untouched.

The logic of the decoding program is far too complex to describe here so appendix B contains a (fully commented) listing of the decoding program along with an example of its operation.

4. Printronix Output Routines

To enable output to be produced on the Printronix in a convenient manner a number of simple, but efficient, routines were written. The basic operations provided by these routines enable a programmer to draw lines and plot points at will on the output page.

Descriptions of the routines follow. The procedure `createmap` creates a memory image of the output page. The procedure `setplot` clears this memory image ready for use. The procedure `bitplot` plots a point while `bitdraw` uses `bitplot` to draw lines. Procedure `sendplot` sends the memory image of the output page to a file ready for spooling on the Printronix. There are two macros called `XTRANS` and `YTRANS` that convert values expressed in 1/1000 inch to dot values. For these routines the origin is at the top left of the page and all coordinate values are positive.

These routines are designed to be as efficient as possible. As a result of this aim the slowest routine, namely `sendplot`, had to be made as fast as possible. The reason `sendplot` is slow is because of the I/O operations involved. To reduce time spent in this routine the memory image of the output page is kept in a form that means `sendplot` need only send the bytes contained in the memory image (plus a few extra) to a file to produce correct output for the Printronix.

Appendix C contains a listing of these routines. Once again the code is written in C. All the examples of Printronix output contained in this document are examples of the use of these routines.

5. Character Fonts

There are various tasks associated with implementing the character fonts that must be generated and stored by the TROFF simulator program.

5.1 Font Selection

TROFF provides the user with four character fonts which are all mounted on the phototypesetter at once. TROFF has commands which allow the user to switch from one font to the other at will. A description of the fonts is given in [1]. A brief description of each of the fonts follows.

- | | |
|----------------------|---|
| Times Roman | - Normal character font, similar to the font that is used in this document.
Basically consists of ASCII characters plus a few extra. |
| Times Italic | - Normal Italic font with the same characters as Roman. |
| Times Bold | - Normal bold font with the same characters as Roman and Italic. |
| Special Mathematical | - Special font containing characters such as Greek alphabet characters, arrow symbols and the mathematical integration symbol. |

It was necessary to select from the Hershey fonts the four fonts that were most similar to those required by TROFF.

A program was written to produce large copies of the various Hershey fonts so that details of the fonts could be seen more easily. From the resultant pictorial representations it was not difficult to choose the fonts that most closely resembled those used by TROFF. The fonts are discussed individually in the following sections.

5.1.1 Times Roman Font

The Hershey font corresponding to this font is font 13 (Serif, complex). A detailed representation appears in appendix D.

It should be noted that only the outlines of the font characters are given and that certain areas of the figures must be filled for the font to be displayed correctly.

5.1.2 Times Italic Font

The Hershey font corresponding to this font is font 23 (Italic, complex). This font appears in appendix E.

As with the Roman font certain areas of the font must be filled. Also some extra lines need to be added to the font so that the areas to be filled become enclosed spaces.

5.1.3 Times Bold Font

The Hershey font corresponding to this font is font 62 (Clarendon Bold). The font appears in appendix F.

Some of the characters (numeral 6, upper case R etc.) have errors in their description and must be corrected. All characters in this font obviously require filling.

5.1.4 Special Mathematical Font

Because this font is not standard it is necessary to build it up from characters from a number of other fonts. The Greek alphabet that is part of the Special Mathematical font is Hershey font 43 (Greek, complex), and is shown in appendix G. As mentioned earlier appendix A shows all Hershey fonts and the Special Mathematical characters required can be identified there.

5.2 Filling Algorithm

Because almost all the characters of the Hershey fonts that are to be used require areas of them to be filled a special technique to do this must be developed.

No purely automatic technique can be used, such as those described by Theo Pavlidis[7], because there are certain parts of some characters that must not be filled (the hole on the 'O' for example). For this reason the areas to be filled must be specified by a human. Probably the easiest way to do this is to add extra information to the Hershey font file specifying areas to be filled.

Henry Lieberman[6] gives a description of an algorithm to fill enclosed spaces when given a point in the area to be filled. This algorithm however does not work perfectly because of the way the Hershey fonts are described along with the limited resolution of the Printronix. Figure 4 shows the intersection of two lines that could be part of a Hershey character. Note that when the enclosed area is filled some white space is left that should have been filled.

To resolve this problem an extra phase is added at the end of Lieberman's algorithm so that unfilled points that are adjacent to three or four filled points are filled as well.

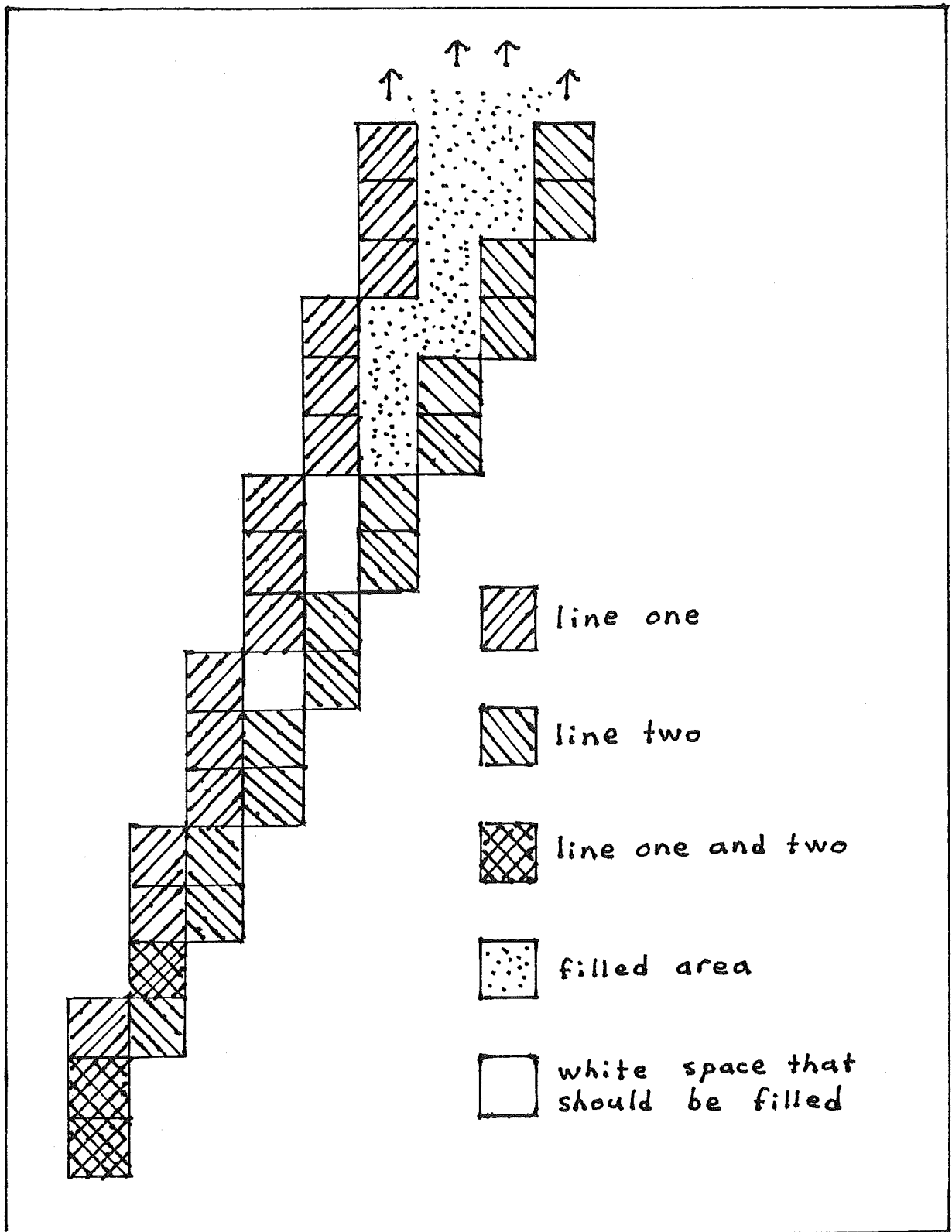


Figure 4

A procedure implementing Lieberman's algorithm was written and is completely independent of the item being filled. It accepts a 2-D coordinate that indicates where to begin filling and uses two procedures (`bitplot` and `point`) that are assumed to already exist. The procedure `bitplot` is used to plot points and the procedure `point` indicates if points are set or not. The procedure is called `fill` and appears in appendix H.

A procedure that implements the extra phase to ensure figures filled with Lieberman's algorithm are filled correctly has also been written. The parameters passed to the procedure describe a region in two-space that must be checked for proper filling. Once again `bitplot` and `point` are used. This procedure only needs to be used once on any region of two-space and this should be only when all areas that need to be filled by Lieberman's algorithm have been. This procedure is called `tidyup` and it too appears in appendix H.

Figure 5 shows a figure that has been built up with a number of straight lines and has then had various enclosed spaces filled with the procedure `fill`. Just before the diagram was written to a file, `tidyup` was used. Note that a side-effect of `tidyup` is that the very tips of very sharp inside corners are filled thus making them look more rounded. This side-effect is desirable for the purposes of filling characters as it will make them appear smoother.

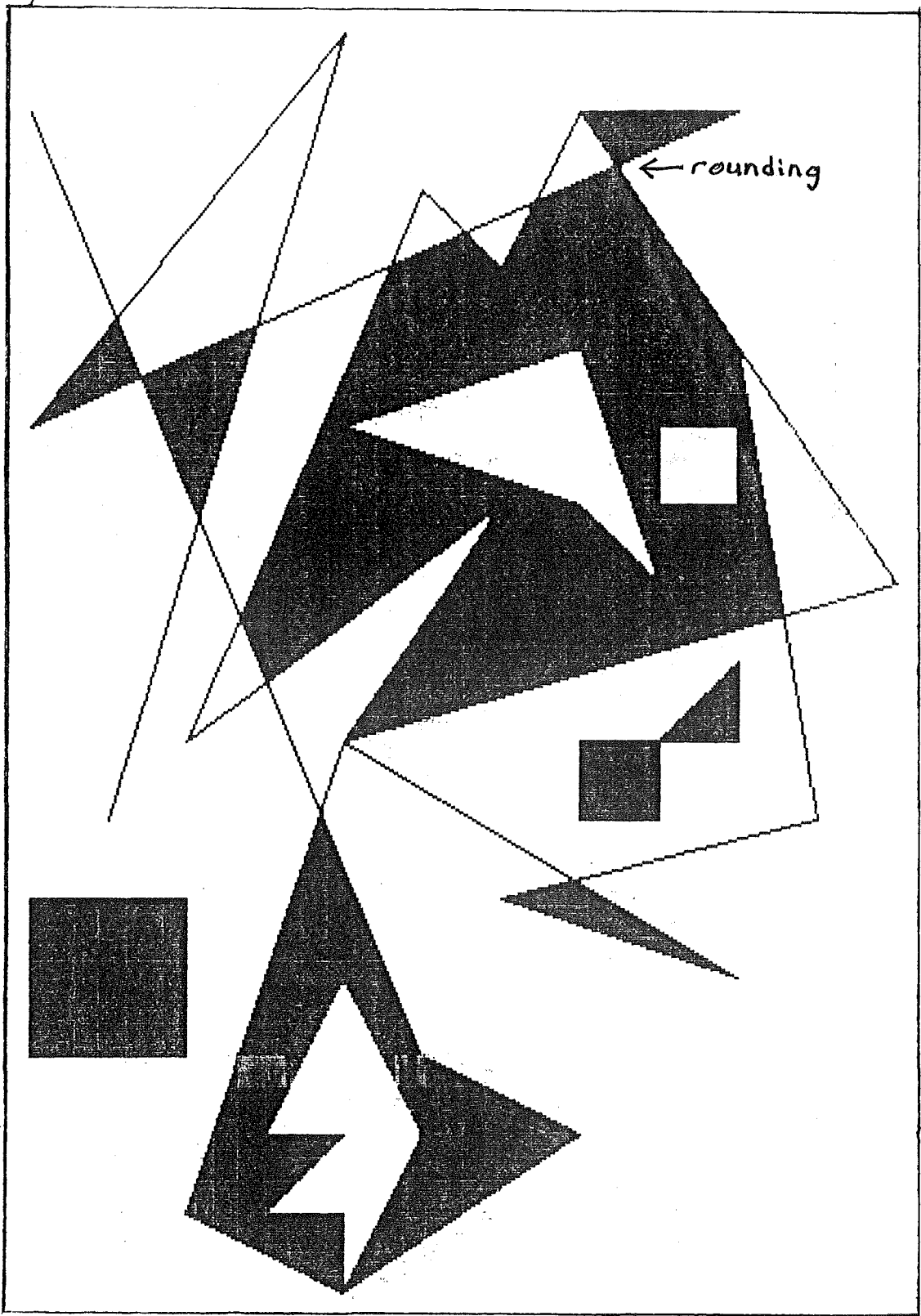


Figure 5

5.3 Font Scaling and Storage

TROFF allows characters to be produced in point sizes of 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24 and 36. This is a range of 1/12 inch to 1/2 inch (one point is 1/72 inch and font size is measured by the height of the main body of a character, the height of a capital H for example). Because TROFF only allows page widths up to 7.54 inch and the paper used by the Printronix is almost twice this width it is clear that users of the TROFF/Printronix interface program should be allowed to specify a scaling factor for their output so that the full page width can be utilised (probably only scale factors greater than one).

If the user is given the ability to scale output then character heights anywhere from 1/12 inch to almost one inch will be allowed. This will make it necessary for the character fonts to be stored in some high-resolution form that can be scaled quickly and to many possible sizes. Because filling operations take a long time it would not really be reasonable to store the character fonts in the form of pen movements and perform the filling each time a character is output. Storage of the character fonts as high resolution bit maps seems to be the only solution. Fonts would then need to be created only once, filled at the same time and stored in a high resolution bit map form in a direct access file. Using this method each character font would occupy at least 64kb of file space.

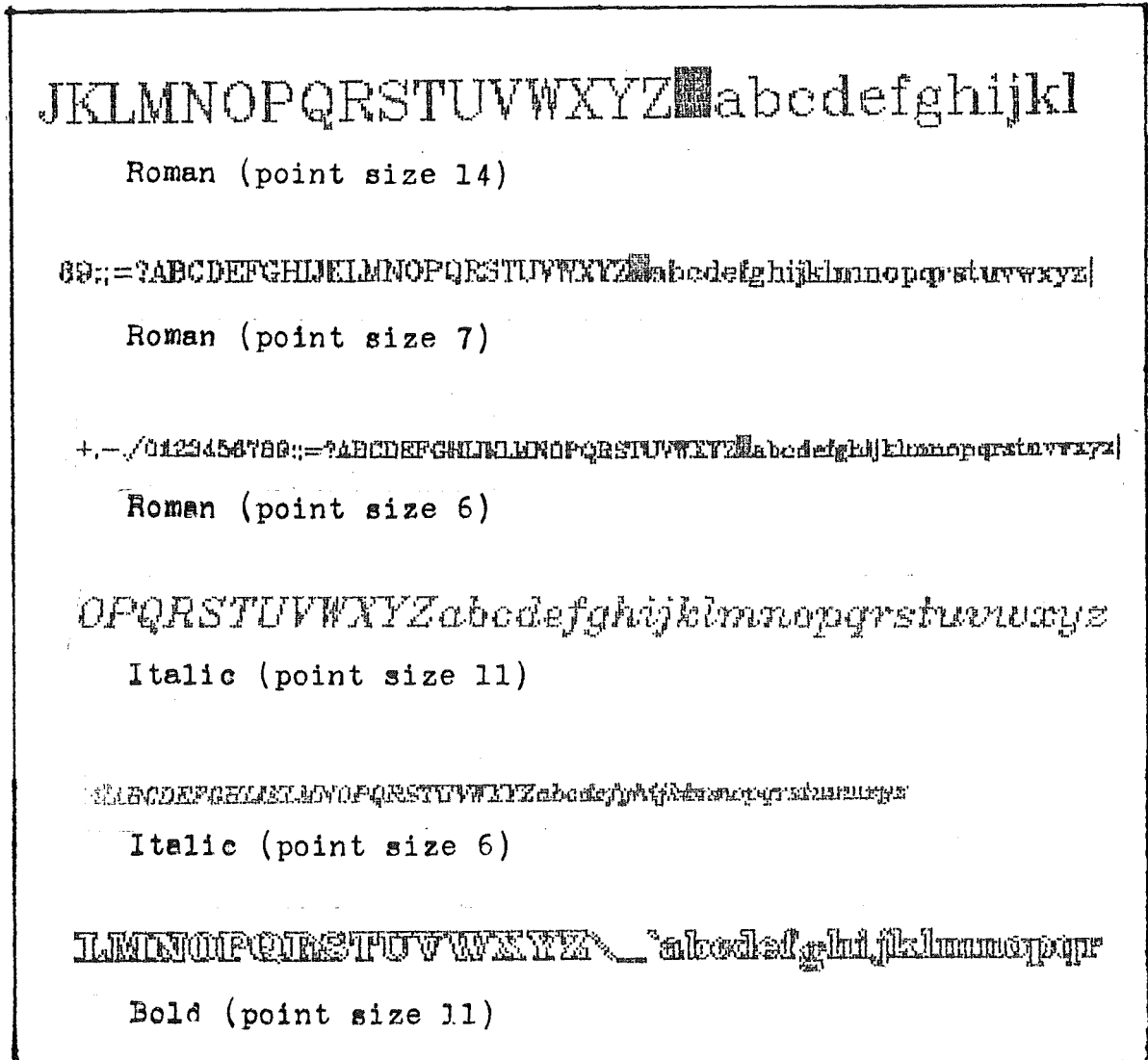


Figure 6

Once the fonts have been stored an efficient but reasonable method of scaling them must be chosen. Figure 6 shows a number of characters scaled using a simple interpolation technique. With

interpolation each pixel in the output is defined by a fixed mapping from a neighbourhood of pixel values in the input pattern. The interpolation method shown sets the output pixel if any of the input pixels in the corresponding neighbourhood are set. The interpolation-scaled characters of Figure 6 show that the result is reasonable but not as good as it could be.

Interpolation is the most commonly used scaling method but there are other methods available. R. G. Casey et al.[8] present a scaling algorithm that results in output that is about as good as possible for an automatic technique. Their algorithm takes into account such factors as local and global symmetries, stroke width and smoothness of contours. The main problem with their algorithm (and other more complex algorithms) is that it is very slow. It requires from 15 to 30 seconds of CPU time on an IBM 3033 computer to map a single high-resolution character from a 120 x 80 array into a 36 x 24 array using Casey's method. As a result of speed requirements and the variety of scaling factors needed simple methods such as interpolation seem to be the most appropriate for this project.

6. The Phototypesetter-Simulator Postprocessor

Previous sections of this document have described the TROFF interpreter program, the Printronix plotting procedures, the scaling technique and the character storage technique. The following is a discussion of the way in which these components should be linked together.

Essentially the TROFF interpreter program should be run and instead of producing descriptions of characters to be plotted it should select the appropriate character of the appropriate font, scale it and place the scaled image at the correct place in the memory image of the output page. When a new page is started the current page should be sent to the output file and the memory image of the page cleared. The only phase left to discuss is how the characters should be buffered so as to reduce disk traffic.

There are various ways of doing this buffering and this is simply a summary of various ideas. When buffering it is possible to buffer a whole font at a time or just individual characters. A whole font will be buffered only if it is assumed that the user will be using the font for a long time (because of the initial overhead cost). This is probably only true for the Roman font as the others are only used for special purposes. It would therefore seem to be best to buffer in terms of single characters and perhaps store the the Roman font permanently.

Another choice that that needs to be made concerns the changing of font size. Should characters be buffered in their scaled form so that when used again they are already scaled or should they be buffered in the same form as in the file so that scaling must always take place? The decision made will depend on whether the user is inclined to change font size often and use

many sizes for an equal amount of time or whether the user tends to spend most time using a single font size. In my opinion users of TROFF would tend to use one font size and only move to other font sizes for short periods (for headings etc.) and hence characters should be stored in their scaled form.

Obviously the replacement strategy used for the character font buffers should be 'Least Recently Used'.

7. Conclusion

A great deal of the work required to realize the aim of this project has been completed but a significant amount remains to be done. The decoding of TROFF output which was perhaps the most difficult (and certainly the least straightforward) part is finished and the resultant program is essentially the front-end of the TROFF postprocessor that is the aim of this project. Routines that simulate the basic actions of the phototypesetter on the Printronix need to be written. Many of these routines are trivial (such as horizontal and vertical motion) and parts of the more complex routines (such as printing characters) have already been written or been described in some detail (for example Printronix control routines and scaling by interpolation).

The major task that remains is the creation of high resolution forms of the character fonts on a direct access file. The way in which the characters are stored must be efficient in terms of file space and in terms of access time.

The aim of this project was to design and implement a phototypesetter-simulator postprocessor for TROFF that produces output on the Printronix 300 and it is unfortunate that this aim has not been fully realized.

- [10] Using the Plot Mode,
Printronic Application Note No. 102370.
- [11] T. C. Waugh, GIMMS System Alphabets, 13 Oct 1977.

Appendix A - GIMMS System Fonts

The following pages give the GIMMS fonts which are equivalent to the Hershey fonts except that font number 1 which is the default GIMMS font is not a Hershey font.

Appendix B - Decoding Program

This appendix contains an example of the operation of the program that decodes output from TROFF as well as a listing of the program.

Example

The following data was input to TROFF and the subsequent output was saved in a file for later input to the decoding program. Line numbers are given and all lines of data start in column one.

```
1 .ps 13
2 .ft R
3 a\FIB\FB c\FR
4 .ps 36
5 .sp 2i
6 \ (14 + \ (12 = \ (34
7 >\h'li'\ (*S = \ (*a\ (*b
```

The input data has the following meaning:

- Line 1 - Change character size to 13 points (this will be rounded up by TROFF).
- Line 2 - Change to Roman font.
- Line 3 - Print 'a',
change to Italic font,
print 'B',
change to Bold font,
print 'c',
change to Roman font.
- Line 4 - Change to 'point size' 36,
output is moved to next line (implicit).

Line 5 - Move down 2 inches.

Line 6 - Print ' $1/4 + 1/2 = 3/4$ '.

Line 7 - Print '>',
move 1 inch to the right,
Print '<Sigma> = <alpha><beta>'.

The output produced by the decoding program follows. Note that coordinates given are in terms of 1/2 points (one point is 1/72 inch) where the origin is the top left corner of the page.

OK, seg decode test /* test contains TROFF output

New Page

Character output

Font is Roman

Point size is 14

Position is (527, 24)

Character is 'a'.

Character output

Font is Italic

Point size is 14

Position is (567, 24)

Character is 'B'.

Character output

Font is Bold

Point size is 14

Position is (651, 24)

Character is 'c'.

Character output

Font is Roman

Point size is 36

Position is (527, 336)

Character is ' $1/4$ '.

Character output

Font is Roman

Point size is 36

Position is (773, 336)

Character is '+'

Character output

Font is Roman

Point size is 36

Position is (1061, 336)

Character is ' $1/2$ '.

Character output

Font is Roman

Point size is 36

Position is (1307, 336)

Character is '='.

Character output

Font is Roman

Point size is 36
Position is (1595, 336)
Character is '3/4'.
Character output
Font is Spc. Math.
Point size is 36
Position is (1841, 336)
Character is '>'.
Character output
Font is Spc. Math.
Point size is 36
Position is (2489, 336)
Character is 'Sigma'.
Character output
Font is Roman
Point size is 36
Position is (2711, 336)
Character is '='.
Character output
Font is Spc. Math.
Point size is 36
Position is (2999, 336)
Character is 'alpha'.
Character output
Font is Spc. Math.
Point size is 36
Position is (3143, 336)
Character is 'beta'
New Page

Listing

A listing of the decoding program appears on the following pages.


```

#include <stdio.h>
/*
TROFF output interpreter
*/

#define DBL 0200

int psize 10; /* point size */
int esc;     /* short horizontal motion */
int escd;   /* x coordinate */
int verd;   /* move up/down */
int esct;   /* move left/right */
int osize 02;
int size 02;
int leadtot -31; /* v coordinate */
int railmag; /* current font */
int lead;    /* distance between lines */
int mcase;   /* upper/lower case */
int stab[] {010,0,01,07,02,03,04,05,0211,
            06,0212,0213,0214,0215,0216,0217};
            /* phototypesetter commands */
int rtab[] {6, 7, 8, 9, 10, 11, 12, 14, 16,
            18, 20, 22, 24, 28, 36, 18};
            /* corresponding point sizes */

int alpha;
extern char *asctab[128]; /* Roman, Italic and Bold */
extern char *spectab[128]; /* Special Mathematical */
FILE *fp; /* input file (TROFF output) */
extern short bin$read;

main(argc,argv)
int argc;
char **argv;
{
    register i, j;
    register char *k;

    if(--argc){
        if((fp=fopen(++argv[0], "r"))==NULL){
            printf("Cannot open: %s\n",argv[0]);
            exit(1);
        }
    }
    bin$read = 1;
    while((i = getc(fp)) >= 0){
        if(!i)continue;
        if(i & 0200){
            esc += (~i) & 0177;
            continue;
        }
        if(esc){
            if(escd)esc = -esc;

```

```

    esct += esc;
    sendpt();
    esc = 0;
}
switch(i) {
    case 0100: /*init*/
        escd =
            verd =
            mcase =
            railmag = 0;
            leadtot = -31;
            sendpt();
            printf("New Page\n");
            continue;
    case 0101: /*lower rail*/
        railmag = & ~01;
        continue;
    case 0102: /*upper rail*/
        railmag = | 01;
        continue;
    case 0103: /*upper mag*/
        railmag = | 02;
        continue;
    case 0104: /*lower mag*/
        railmag = & ~02;
        continue;
    case 0105: /*lower case*/
        mcase = 0;
        continue;
    case 0106: /*upper case*/
        mcase = 0100;
        continue;
    case 0107: /*escape forward*/
        escd = 0;
        continue;
    case 0110: /*escape backward*/
        escd = 1;
        continue;
    case 0111: /*stop*/
        continue;
    case 0112: /*lead forward*/
        verd = 0;
        continue;
    case 0113: /*undefined*/
        continue;
    case 0114: /*lead backward*/
        verd = 1;
        continue;
    case 0115: /*undefined*/
    case 0116:
    case 0117:
        continue;
}

```

```

if((i & 0340) == 0140) { /*leading*/
    lead = (~i) & 037;
    if(verd) lead = -lead;
    leadtot += lead;
    sendpt();
    continue;
}
if((i & 0360) == 0120) { /*size change*/
    i = & 017;
    for(j = 0;
        i != (stab[j] & 017);
        j++);
    osize = size;
    size = stab[j];
    psize = rtab[j];
    i = 0;
    if(!(osize & DBL) && (size & DBL))
        i = -55;
    else if((osize & DBL) &&
        !(size & DBL)) i = 55;
    if(escd) i = -i;
    esc += i;
    continue;
}
if(i & 0300) continue;
i = (i & 077) | mcase;
if(railmag != 03) k = asctab[i];
else k = spectab[i];
if(alpha) sendpt();
printf("Character output\n");
printf("    Font is ");
switch(railmag) {
    case 0: printf("Roman");          break;
    case 1: printf("Italic");        break;
    case 2: printf("Bold");          break;
    case 3: printf("Spc. Math.");    break;
}
printf("\n");
printf("    Point size is %d\n", psize);
printf("    Position is (%d, %d)\n",
    esct,
    leadtot);
printf("    Character is '%s'.\n", k);
alpha++;
continue;
}
exit(0);
}
sendpt() { /* a hangover from TCAT */
    alpha = 0;
    return;
}

```

Appendix C - Plotting Routines

The plotting routines appear on the following pages.

```
/* This file contains listings of procedures that
   produce graphics output on the Printronix 300.
   The UNITS used are 1/1000 inch. */
```

```
#include <stdio.h>
#define XUNITS 13200 /* page width is 13.2 in. */
#define YUNITS 11000 /* page length is 11 in. */
#define XDOTSPERUNIT 0.060 /* 60 dots per horiz. inch */
#define YDOTSPERUNIT 0.072 /* 72 dots per vert. inch */
#define XXDOTS (int) (XDOTSPERUNIT * XUNITS + 0.999)
#define YYDOTS (int) (YDOTSPERUNIT * YUNITS + 0.999)
#define XXBYTES (XXDOTS + 5)/6 /* bytes per dot row */
#define BITMAPSIZ (XXBYTES * YYDOTS + 1)/2
#define TRUE 1
#define FALSE 0
```

```
/* translations from UNITS to dots */
```

```
#define XTRANS(a) (int) (XDOTSPERUNIT * (a) + 0.5)
#define YTRANS(a) (int) (YDOTSPERUNIT * (a) + 0.5)
FILE *outf; /* output file */
short unsigned int *bitmap;
/* A pointer to the memory representation of the
   output page. The bytes in this map correspond
   one-for-one with the bytes that indicate dot
   positions when output to the Printronix */
```

```
createmap()
```

```
/* create memory space for the map */
```

```
{
  bitmap = (short unsigned int *)
           calloc( BITMAPSIZ, sizeof(short unsigned int) );
}
```

```
setplot()
```

```
/* clear the memory image of the page */
```

```
{
  int i;
  for( i = 0 ; i < BITMAPSIZ ; *(bitmap + i++) = 0xc0c0 )
  ;
}
```

```

sendplot()

/* send the current plot page to the file 'outf' */
{
    int byte, j;

    putc('\f', outf); putc(005, outf); putc('\n', outf);
    for( j = 0 ; j < YYDOTS ; j++ ){
        putc(005, outf);
        for( byte = XXBYTES*j ;
            byte < XXBYTES*(j + 1) ;
            byte++ ){
            putc( (int) *(bitmap + (byte>>1)) >>
                ( (byte & 1) ? 8 : 0 ) & 0xff, outf);
        }
        putc('\n', outf);
    }
    putc('\n', outf);
}

bitdraw(x1, y1, x2, y2)
int x1, y1, x2, y2;

/* draw a line from dot coordinates (x1, y1) to (x2, y2) */
{
    int delx, dely, i;
    double m, b;

    delx = x2 - x1; dely = y2 - y1;
    if(!delx && !dely)
        bitplot(x1, y1); /* just one point to plot */
    else{
        if (abs(delx) > abs(dely)) {
            m = (double) dely / (double) delx;
            b = (double) (y2 - x2*m);
            for(i = x1 ; i <= x2 ; i++)
                bitplot(i, (int) (m*i + b + 0.5));
            for(i = x1 ; i >= x2 ; i--)
                bitplot(i, (int) (m*i + b + 0.5));
        }else{
            m = (double) delx / (double) dely;
            b = (double) (x2 - y2*m);
            for(i = y1 ; i <= y2 ; i++)
                bitplot((int) (m*i + b + 0.5), i);
            for(i = y1 ; i >= y2 ; i--)
                bitplot((int) (m*i + b + 0.5), i);
        }
    }
}
}

```

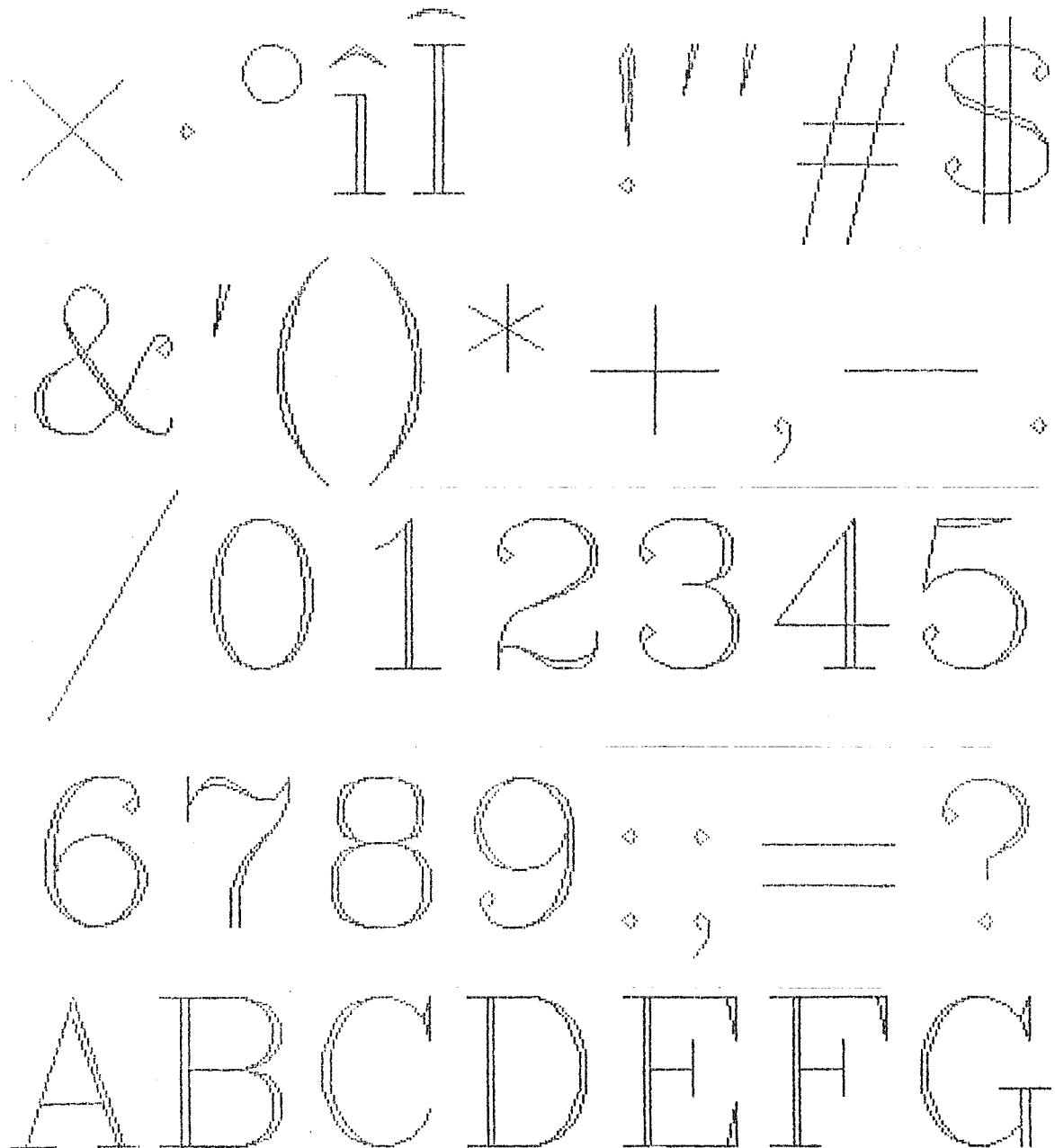
```

bitplot(x, y)
int x, y;
/* plot point (x, y) */
{
    int byte;
    byte = XXBYTES*y + x/6;
    *(bitmap + (byte>>1)) |=
        (1 << (x%6 + ( (byte & 1) ? 8 : 0 ) ) );
}

```

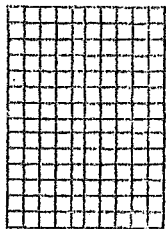
Appendix D - Roman Font

The font displayed is Hershey font 13 (Serif, complex).



H I J K L M N O

P Q R S T U V

W X Y Z  a b

c d e f g h i j k

l m n o p q r

s t u v w x y z |

Appendix E - Italic Font

The font displayed is Hershey font 23 (Italic, complex).

î î̂ A B C D E
F G H I J K L
M N O P Q R
S T U V W X
Y Z a b c d e f

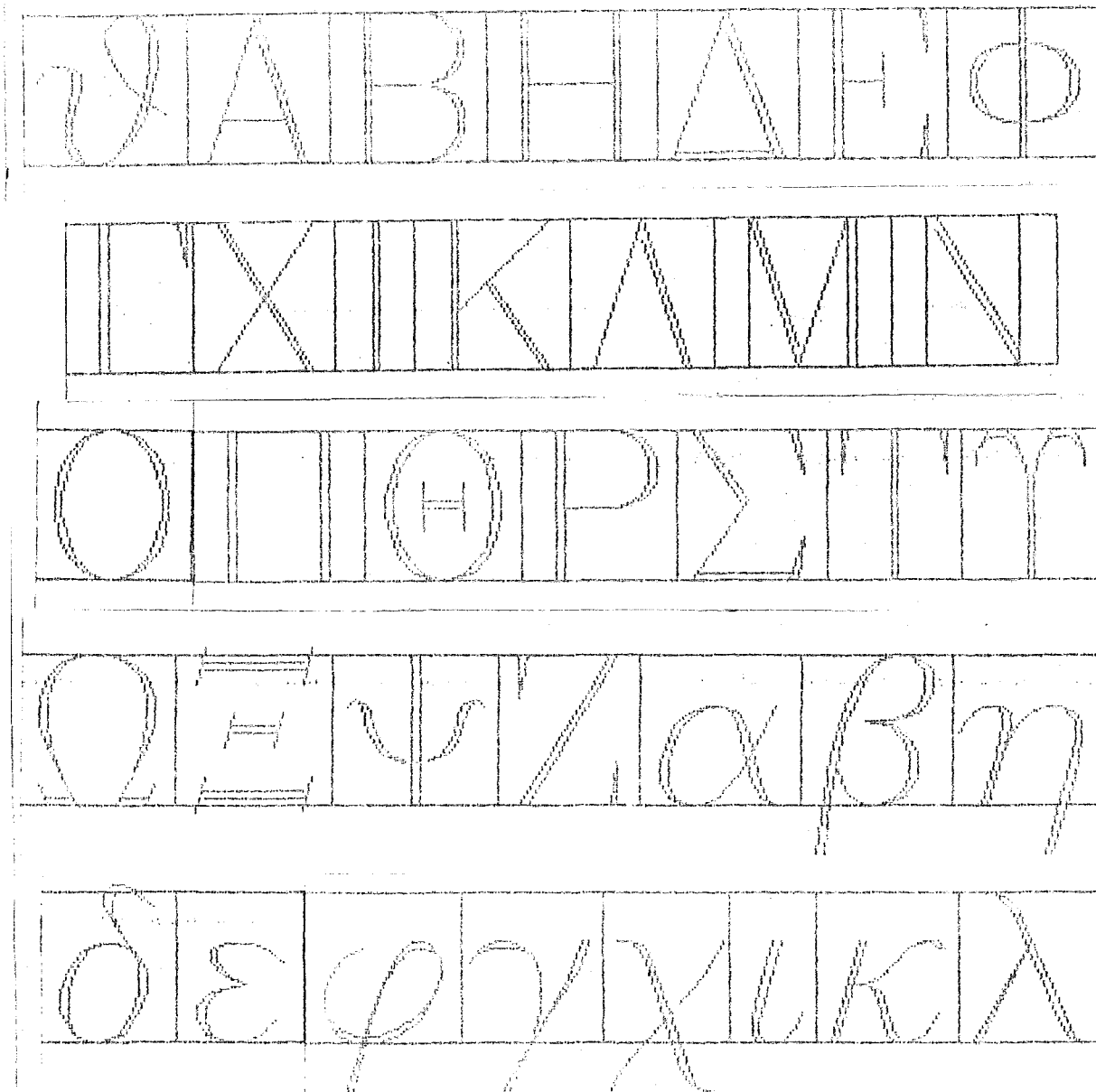
g h i j k l m n

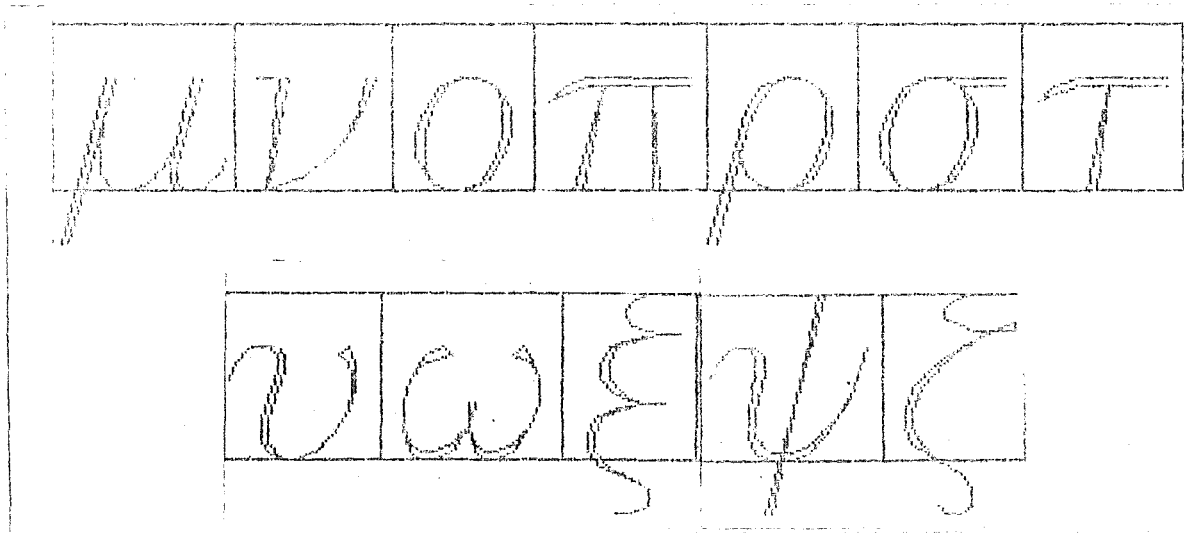
o p q r s t u v

w x y z

Appendix G - Greek Font

The font displayed is Hershey font 43 (Greek, complex).





Appendix H - Filling Routines

The filling routines appear on the following pages.

```

/* This listing contains definitions of general purpose
   filling procedures. It is assumed that procedures
   that do the following are already defined.

```

```

    bitplot(x, y) - To plot a point at (x, y)
    point(x, y)   - Returns a boolean value indicating if
                   there is a point at (x, y)

```

```

*/

```

```

#define POINTCHECK(y, z)    if(point(i+y, j+z))
                            on++;
                            else
                            off++;
                            if(!on) {
                                bitplot(i, j);
                                goto proceed;
                            }
                            if(!off)
                                goto proceed

```

```

#define AGENDASIZE 50
#define UP 1
#define DOWN 0
#define AGENDAC(c) agenda[c][0].lx
    /* number of lines in agenda[c] */

```

```

struct horizline {
    int lx, /* left end */
        hx, /* right end */
        y; /* dot row */
};

```

```

fill(ox, oy)
int ox, oy;

```

```

/* fill the enclosed area that includes the origin
   point (ox, oy) */

```

```

{
    int vertdir; /* current direction of the fill */
    struct horizline agenda[l][AGENDASIZE];
    /* Beginnings of areas yet to be filled.
       Lines in the agenda have been filled.
       Filling starts from line above/below
       this line.
       If line in agenda[UP] then fill up
       else fill down. */

```



```

if(point(ox, oy)) /* if area already filled */
    return;

/* fill horizontal line containing origin
   and add entries to the agenda to indicate a
   fill of areas above and below origin */

findandfill(ox, oy, &(agenda[UP][1].lx),
            &(agenda[UP][1].hx));
agenda[UP][1].y = oy;
agenda[DOWN][1].lx = agenda[UP][1].lx;
agenda[DOWN][1].hx = agenda[UP][1].hx;
agenda[DOWN][1].y = oy;
AGENDAC(UP) = AGENDAC(DOWN) = 1;
vertdir = UP;

/* continue filling until agenda is empty */

while ( AGENDAC(UP) || AGENDAC(DOWN) ) {
    if( !(AGENDAC(vertdir)) )
        vertdir = !vertdir; /* change direction */
    shadevert(agenda, vertdir); /* fill next area */
}

findandfill(x, y, lxptra, hxptr)
int x, y, *lxptra, *hxptr;

/* find the unfilled horizontal line of which (x, y)
   is part, fill it, and return its limits in
   lxptra and hxptr */

{
    int i;

    for ( i = x ; !(point(i, y)) ; i-- )
        ;
    *lxptra = i+1; /* left limit */
    for ( i = x ; !(point(i, y)) ; i++ )
        ;
    *hxptr = i-1; /* right limit */
    for( i = *lxptra ; i <= *hxptr ; bitplot(i++, y) )
        ;
}

shadevert(agenda, vertdir)
int vertdir;
struct horizline agenda[1][AGENDASIZE];

/* select the next line in 'vertdir' from 'agenda' and
   start filling from there */

```

```

{
    struct horizline *last, *curr, dummy1, dummy2, *temp;
    int y, i;

    /* remove the line from agenda and make it the 'last'
       one filled */

    last = &dummy1; curr = &dummy2;
    last -> lx = agenda[vertdir][AGENDAC(vertdir)].lx;
    last -> hx = agenda[vertdir][AGENDAC(vertdir)].hx;
    last -> y = agenda[vertdir][AGENDAC(vertdir)--].y;

    for( ;; ) {

        /* find next line in current direction and fill it */

        y = last -> y + (vertdir == UP ? -1 : 1);
        for( i = last -> lx ; i <= last -> hx &&
            point(i, y) ; i++ )

            ;
        if( i == last -> hx + 1 )
            return; /* dead end */
        findandfill(i, y, &(curr -> lx), &(curr -> hx));
        curr -> y = y;

        /* add to agenda unfilled lines adjacent to
           previous line in current direction */

        addagenda(agenda, last, vertdir);

        /* add to agenda unfilled lines adjacent to
           current line opposite to current direction */

        addagenda(agenda, curr, !vertdir);
        temp = last; last = curr; curr = temp; /* update 'last' */
    }
}

addagenda(agenda, line, vertdir)
int vertdir;
struct horizline agenda[1][AGENDASIZE], *line;

/* add to agenda unfilled lines adjacent to 'line'
   in the 'vertdir' (fill these lines at the same
   time) */

{

    int y, i;

    y = line -> y + (vertdir == UP ? -1 : 1);
    for( i = line -> lx ; i <= line -> hx ; i++ ) {

```

```

if(!point(i, y)) {
    (AGENDAC(vertdir))++;
    findandfill(i,
                y,
                &(agenda[vertdir][AGENDAC(vertdir)].lx),
                &(agenda[vertdir][AGENDAC(vertdir)].hx));
    agenda[vertdir][AGENDAC(vertdir)].y = y;
    i = agenda[vertdir][AGENDAC(vertdir)].hx;
}
}
}

```

```

tidyup(x1, x2, y1, y2)
int x1, x2, y1, y2;

```

```

/* for each point within the boundaries specified by
the two x values and two y values, if 3 or more
of its directly adjacent neighbours are set then
set it too */

```

```

{
    int on, off, i, j;
    for( j = y1+1 ; j < y2 ; j++ )
        for( i = x1+1 ; i < x2 ; i++ )
            if( !(point(i, j)) ) {
                on = -3; off = -2;
                POINTCHECK(1, 0);
                POINTCHECK(-1, 0);
                POINTCHECK(0, 1);
                POINTCHECK(0, -1);
                proceed:
                {

```