PICO MANUAL

William M. Newman

Robert F. Sproull

July 1974

Xerox Palo Alto Research Center

TABLE OF CONTENTS

PREFACE

Pico is a graphics package for people who want to write interactive graphical programs, and for people who have programs to which they would like to add graphical input/output. At present only BCPL programs may call the Pico package, but versions for use with INTERLISP and Smalltalk are on the way. Pico can be used with the following hardware systems:

(a) Any standard Alto, preferably wth 64K memory;

(b) Ben Laws' run-code display and its parent Alto;

(c) The color graphics Nova system.

Pico can handle inputs from mice and tablets; it can generate graphic hardcopy with the aid of the XGP.

About this manual

You will find that this manual consists of four sections:

I: An *introduction*, where some of the essential features of Pico are explained with the aid of examples. Everyone should read this section.

II: A concise description of the *basic functions* of Pico. This section should also be read by every potential user. Once you have read it, you should be able to write your first program using Pico.

III: Instructions on how to *use* Pico -- where to find the necessary files, what to load with your compiled program, and so forth. You will need to read this section in order to run your program.

IV: A description of the more *advanced* features of Pico. You won't need these unless you wish to manipulate curves, perform special transformations, or construct special input schemes.

At the end are several Appendices, describing the free storage system and floating point routines that are integral to Pico, the online character recognizer, and the formats of font and hard-copy files used by Pico.

We hope this arrangement of the manual is agreeable to all. Some readers may need additional background information; if so, they will find some useful references in the Bibliography. Comments about Pico and about this manual will be gratefully received by the authors.

Acknowledgements

Pico was designed and implemented by the following members of the Graphics Systems Group: Patrick Baudelaire, Mike Cole, Bob Flegal, William Newman, Dick Shoup and Bob Sproull. Figures 3, 8 and 9 in this manual were drawn by Bob Flegal with the aid of Smalltalk.

SECTION I: INTRODUCTION

Pico contains functions both for generating graphic output on a display screen or on the XGP, and for handling graphic inputs from a tablet or mouse. These two classes of function, input and output, are kept almost completely separate. We believe programmers will find this separation convenient. We also think it is easier to explain Pico by treating input and output separately: we will discuss output first.

Pictures generated by Pico are made up of basic entities of three kinds, lines, curves and text; areas enclosed by lines and curves may be *filled* with a uniform gray level or, if you have the hardware to do so, with a color. The functions that define these basic entities are called *primitive functions*. One can think of these functions *almost* as if they add lines, curves and so forth directly to the information on the screen. This is not quite true, however. Instead the information is deposited in a *display file*; the screen is not updated from the display file except when the function UPDATE is called. Thus you can obtain a simplified view of the organization of the system by studying Figure 1, ignoring the light gray boxes.

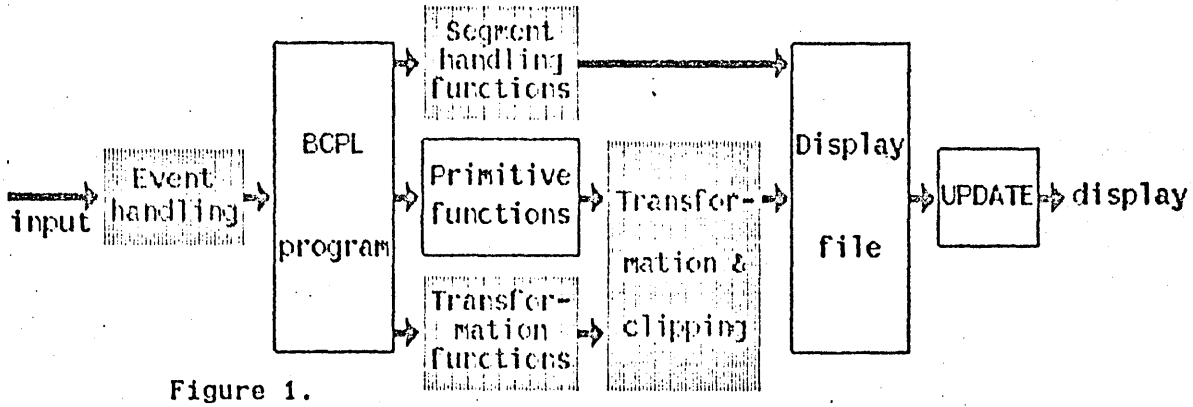


Figure 1.

Since we are dealing with filled areas, it is possible for graphical enities to overlap. Where two or more entities overlap, a simple rule determines what is seen: the thing most recently added to the display file is always visible, and may hide things added less recently. Thus to display text on a gray background, one calls the primitives to generate the gray area, then calls the text-display function. This is illustrated in the example that follows.

The display file is not just a simple list of graphical primitives: it can be divided into *segments*. The use of segments has two major advantages: it permits individual parts of the picture to be changed independently of each other, and it allows things to overlay each other independently of execution sequence. The functions for manipulating segments include routines to create new segments, replace segments, delete them, add to them and change the order in which they are overlaid. Note that none of these functions has any immediate effect on what is visible on the screen: after the appropriate changes to the display file have been made, the UPDATE function must be called to cause the screen picture to change.

Primitive functions allow pictures to be defined in *screen coordinates*. On a standard Alto display, for example, the screen coordinate system places (0,0) at the bottom left-hand corner of the screen, and (605,799) at the top right. This is not always convenient. *Transformation routines* are therefore provided so that parts of the picture may be scaled and rotated, and so that the whole picture may be defined in a coordinate system independent of the particular display in use. The light gray boxes in Figure 1 show how transformation and segment-handling functions relate to the rest of the system.

This completes our brief outline of the graphical output facilities for BCPL. The following example illustrates how they may be used. It generates the "STOP" sign shown in Figure 2. The functions used in this example are described in more detail in Section II.

```
GET "GSDEFS.SR"          This file contains the definitions
                         needed for the use of Pico.
LET MAIN() BE [
INITGRAPHICS()           Initializes the graphics system.
OPENSEG(1)               This states our intention to begin
                         creation of segment number 1.
MOVETO(50,0)             This sets the current (x,y) position
                         to the bottom of the post.
DRAWTO(50,200)           This adds to the display file a line
                         from (50,0) to (50,200). The current
                         position becomes (50,200).
SETCOLOR(GRAY)           This function sets the intensity of
                         the sign's rectangular area.
BEGINFILL()              This indicates the beginning of a
                         "filled" area for the stop sign. The
                         following MOVETO and DRAWTO commands
                         specify the outline of the sign.
  MOVETO(0,200)
  DRAWTO(0,275)
  DRAWTO(100,275)
  DRAWTO(100,200)
  DRAWTO(0,200)
ENDFILL()                Signals the end of the polygon.
SETCOLOR(BLACK)          Specifies intensity for the STOP text.
MOVETO(35,230)           Specifies the starting position of the
                         text.
DRAWTEXT("STOP")         Causes a text string to be added to
                         the display file.
CLOSESEG()               Specifies the end of creation of
                         segment 1 of the display file.
POSTSEG(1)               Specifies that the contents of segment
                         1 are to be shown on the screen the
                         next time the screen is updated.
UPDATE()                 Updates the screen by scan-converting
                         the information in all segments that
                         have been POSTed.
]
```

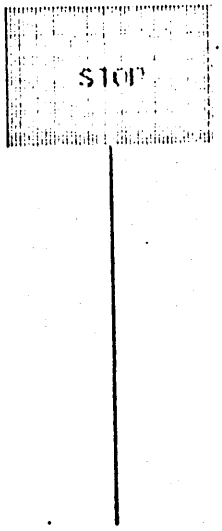
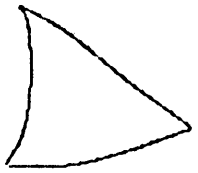


Figure 2.

The next example shows the use of one of the graphic input functions of Pico, use of the DRAW function to perform simple transformations, and also shows how to generate graphic hard-copy. It uses Pico's online character recognizer. We assume that the recognizer has previously been "trained" to recognize two symbols, a triangle and the letter "P", and that a file SYMS.RC has been generated, containing the results of this training session. Now when the user draws a triangle, such as the example shown in Figure 3, a logic symbol for an inverter is added to the picture on the screen. When the user prints "P", a file is generated for producing XGP hard-copy.

Figure 3.



```
GET "GSDEFS.SR"
LET MAIN() BE [
INITGRAPHICS()
SETRECOGNIZER("SYMS.RC")      The recognizer tables are loaded from the file
                              generated during the previous training
                              session.
LET SN=0                      Initialize the segment name sequence.
 [ LET V=RECOGNIZE()          Wait for the user to draw a symbol, then
                              return a vector containing information about
                              the symbol.
 SWITCHON V>>EVENT.CODE
  INTO [                      The CODE entry in V contains the numeric code
                              of the recognized symbol.
  CASE ST:                    The user drew a triangle (trained to return
                              code "T")
   SN=SN+1                    Create a new segment name.
   OPENSEG(SN)                Start a new segment.
   DRAW(INVERTER, TRANSLATE, V>>EVENT.XLEFT, V>>EVENT.YBOTTOM)
                              Draw an inverter, using the procedure given
                              below; position the symbol with its bottom
                              left-hand corner aligned with the
                              corresponding corner of the drawn symbol.
   CLOSESEG()                 Close the new segment.
   POSTSEG(SN)                Add it to the list of posted segments.
   UPDATE()                   Update the screen picture.
   ENDCASE
  CASE SP:                    The user drew a "P".
   PLOTSEGS("INV.XB")         Output a file INV.XB for the XGP.
   ENDCASE
  ]
 ] REPEAT                     Repeat this loop endlessly.
]

AND INVERTER() BE [           Define the procedure to draw the inverter
                              symbol.
 MOVETO(0,0); DRAWTO(0,50); DRAWTO(50,25)
 DRAWTO(51,23); DRAWTO(53,23); DRAWTO(54,25)
 DRAWTO(53,27); DRAWTO(51,27); DRAWTO(50,25)
 DRAWTO(0,0)
]
```

Figure 4 shows a typical plot generated after a sequence of interactions.

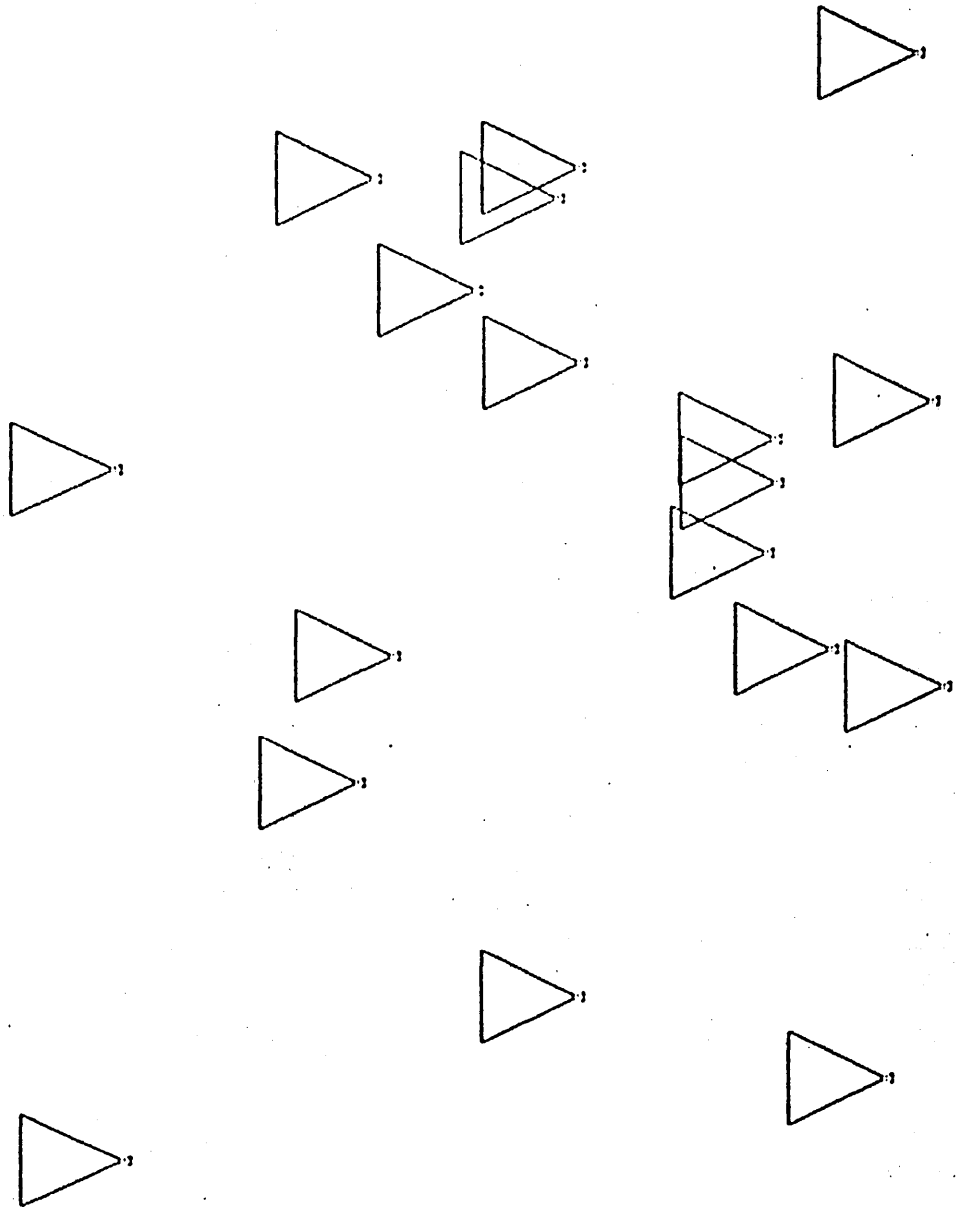


Figure 4.

SECTION II: BASIC GRAPHICS FUNCTIONS

This section describes the basic functions of what may be called the *kernel* of Pico. Programs written using this kernel will work on all three classes of display mentioned above. Each display, however, has certain characteristics of its own: these are mentioned as appropriate below.

GRAPHICAL OUTPUT -- Segment-Handling Functions

The display file is divided into *segments*; each segment can be thought of as an ordered collection of primitive graphical entities. To create a segment, the programmer "opens" a specific segment, specifies the primitive entities that are to be added to the segment, and then "closes" it. Each segment is assigned a 16-bit "name" by the programmer; this name is used if later reference to the segment is necessary.

Note well: none of the following segment-handling functions changes the image visible on the screen.

OPENSEG (segment-name). This function creates a new segment with the specified name. If a segment of the same name already exists, it will be replaced by the new segment. All subsequent graphical primitives are added to this new "open" segment. Before opening the new segment, any other segment still open is closed.

CLOSESEG (). This function closes the currently open segment. Any existing segment with this name is deleted. If no segment is open, CLOSESEG has no effect.

DELETESEG (segment-name). This function deletes the specified segment. If the specified segment does not exist, this function has no effect. DELETESEG never deletes the "open" segment.

APPENDSEG (segment-name). This function opens the specified segment for additions. All subsequent graphical entities are added to the end of the segment. If the specified segment does not exist, APPENDSEG is equivalent to OPENSEG.

POSTSEG (segment-name). This function adds the specified segment to the list of those that should be displayed on the screen. This list is called the "posted" list. If the specified segment is still open at the time of the POSTSEG call, it is closed before posting. Thus the sequence OPENSEG, <graphical primitives>, POSTSEG is sufficient. If the specified segment does not exist, this function has no effect.

UNPOSTSEG (segment-name). This function removes the specified segment from the posted list. The graphical entities within the segment are unaltered. At some later time, the same segment may be posted again. If the specified segment does not exist, this function has no effect.

RENAMESEG (old-segment-name,new-segment-name). This function has no effect on the contents of the display file, but merely changes the

name of the segment specified by "old-segment-name" to "new-segment-name." If a segment with name "new-segment-name" already exists, it is deleted. If no segment named "old-segment-name" exists, the RENAMESEG function has no effect.

CLEARSEGS (). Deletes all segments in the display file, including any segment currently open. No changes are made to the image on the display screen.

GRAPHICAL OUTPUT -- Updating the Display

UPDATE (). This is the *only* function that causes the screen to be updated, other than PLOTSEGS which performs an UPDATE in the process of generating a hard-copy file (see below). If any segments have been altered (created, unposted, posted, deleted, etc.) since the previous call to UPDATE, the picture on the screen is changed appropriately.

Each graphical object in the display file has a "priority" associated with it. When the screen is UPDATED, it may happen that two distinct graphical objects may appear at the same spot on the screen. In Figure 3, for example, the characters "STOP" and pieces of the sign polygon fall on the same dots on the screen. In this case, the graphical object with the highest priority is displayed. The priority rule is very simple:

- Within a segment, the priority order corresponds to the order in which the graphical objects were added to the segment; objects added last have highest priority and thus overlay objects added earlier. It is for this reason that the characters STOP take priority over the sign polygon.

- Between segments, the signed 16-bit integer name is used to decide priority; segment A overlays segment B if A > B. The RENAMESEG function is provided so that inter-segment priorities may be rearranged.

GRAPHICAL OUTPUT -- Graphical Primitives

Graphical primitives are used to specify straight and curved lines, polygons, filled curves (figures whose outlines are curves), and text. These entities are transformed, clipped, and then added to the currently open segment. The color or intensity of entities is defined with the SETCOLOR function.

MOVETO (x,y)
DRAWTO (x,y)
These functions specify the coordinates of line endpoints; MOVETO sets the "current position" to (x,y). DRAWTO draws a vector from the current position to (x,y) and then sets the current position to (x,y). The coordinates are signed 16-bit quantities; since they will typically be transformed (see below), the coordinate system can be chosen by the programmer.

DRAWTEXT ("text-string")
The specified text string is displayed, starting at the current position, and then in subsequent horizontal character positions. Note that no transformations are performed on characters, other than the translation implied by setting the starting position with a MOVETO. A standard font is used unless the program indicates otherwise with the SETFONT function (see below).

BEGINFILL ()
ENDFILL ()
These functions permit MOVETO and DRAWTO functions to be used to specify a filled polygon. A simple example of polygon specification is BEGINFILL MOVETO, DRAWTO, DRAWTO, DRAWTO, ENDFILL. This would normally produce a three-sided polygon. If the locations specified by the initial MOVETO and the final DRAWTO do not coincide, however, Pico automatically inserts a DRAWTO to close the polygon. The examples below demonstrate this.

"Holes" may be specified inside polygons by means of several MOVETO, DRAWTO, DRAWTO, DRAWTO... sequences within one BEGINFILL, ENDFILL pair. Thus we can produce Figure 5 with the following statements:

```
BEGINFILL()
    MOVETO(0,0); DRAWTO(20,40); DRAWTO(40,0);
    MOVETO(10,10); DRAWTO(20,30); DRAWTO(30,10);
ENDFILL()
```

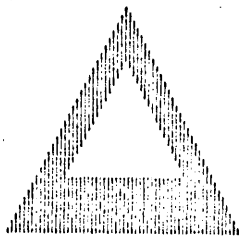


Figure 5.

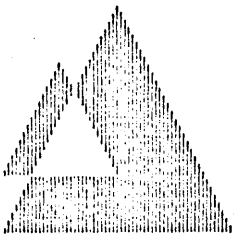


Figure 6.

The closed curves specified within one BEGINFILL, ENDFILL pair may cross, producing effects such as Figure 6, in which the inner triangle of Figure 5 has been displaced.

SETCOLOR (gray-level)
SETCOLOR (red-component,green-component,blue-component)
The SETCOLOR function may be used with a single argument to set gray levels between 0 (the default value, representing black) and 255 (representing white). Three manifest constants, BLACK, GRAY (=127) and WHITE, may be used where appropriate. The effect of SETCOLOR will vary somewhat with different output devices: the color graphics system, if used in black-and-white mode, will generate 256 different

gray levels, the run-code display produces 32, but the XGP and the standard Alto display produce only eight dot patterns of differing densities. Note that large black areas do not reproduce on the XGP.

With three arguments, SETCOLOR may be used to generate colors on the color graphics system. Components must be in the range 0 to 255.

SETBACKGROUND (gray-level).
SETBACKGROUND (red-component,green-component,blue-component)
This function specifies the intensity of the background, i.e. the intensity that is displayed where no graphical entity is visible. Values have the same range and interpretation as in the SETCOLOR function. Either WHITE (the default value) or BLACK should be used with the standard Alto display, to save memory.

SETCOLOR and SETBACKGROUND may be called at any point in the program. In certain situations their effect is deferred, however: SETCOLOR, if called after a BEGINFILL, will take effect only after the corresponding ENDFILL; the effect of SETBACKGROUND is seen only when UPDATE is next executed.

GRAPHICAL OUTPUT -- Transformations

The first example of Section I defined a picture entirely in the *screen coordinate system*. This system is always in effect unless the program specifies otherwise. There are three main reasons why you may wish not to use screen coordinates:

1. You may wish to use symbols that are defined in local coordinates, and that are to be scaled, rotated or translated before they are displayed, like the 'inverter' in our second example.

2. You may wish to define pictures too big to fit on the screen, and then to select parts of such pictures to be displayed at various enlargements.

3. You may wish to write programs that are not affected by the different screen characteristics of the different displays.

Pico includes a number of transformation functions that cater to these needs. To understand them, it is important to realize that Pico in fact allows you to look through a conceptual *window* at a large *page* of graphical information. This page, and the rectangular window onto it, may use a coordinate system quite different from the screen's. Normally one will define the window size with the SETWINDOW function, before opening the segments on which this window operates. When Pico constructs a segment of display file, it transforms everything into the *page coordinate system*; then it 'clips' away everything lying outside the window, and transforms the rest into screen coordinates. If, as in our earlier examples, SETWINDOW is not called, Pico uses default values that equate the page and screen coordinate systems.

Symbols included in the page information must be transformed from their local coordinate system into page coordinates, and Pico provides a DRAW

function to do this: DRAW uses the notion of describing symbols as *display procedures* (see Reference 1). Essentially, every time a symbol is to be added to the currently open segment, a call is made to DRAW, specifying (a) the name of the procedure defining the symbol, and (b) the transformations to be applied to the symbol in order to place it correctly in the page space.

The full form of the DRAW function's calling sequence is as follows:

```
DRAW(procedure-name,procedure-arg1,procedure-arg2,...
   SCALE,sx, [[sy,] sw,]
   ROTATE,theta,
   TRANSLATE,translation-x,translation-y)
```

where:

'procedure-name' is the name of the display procedure, and 'procedure-arg1' etc., are its arguments, if any;

sx/sw and sy/sw are the scale factors in the x- and y- direction; if sy is omitted, the procedure is scaled by sx/sw in both directions, while if both sy and sw are omitted, sx is used as an integral enlargement factor in both directions;

theta is the anti-clockwise rotation in degrees;

translation-x and translation-y are translations in the x-and y-directions.

The DRAW function assembles all the transformations together into a single matrix, combines this with any existing transformation, and then calls the named procedure. The resulting transformation is applied to all the primitives called by this procedure. When the procedure returns, DRAW restores whatever transformation was previously in effect, and then itself returns. This mechanism permits display procedures to include calls to other such procedures via the DRAW function.

The full form of the DRAW calling sequence is rarely necessary. Any identity transformations may be omitted, and the display procedure need not have arguments. If two or more transformations are given, they will effectively be performed in the order specified. The order given above, SCALE-ROTATE-TRANSLATE, is the normal sequence to use in transforming symbols.

The SETWINDOW function is called as follows:

```
SETWINDOW (xleft,ybottom,xright,ytop).
```

This function defines a rectangular *window* onto the page information, using *page coordinates*. The bottom left-hand corner of this rectangle is at (xleft,ybottom), and the top right-hand corner is at (xright,ytop). All information lying outside this window is excluded from the displayed picture.

The following program illustrates the use of SETWINDOW and DRAW. It generates the output shown in Figure 7 overleaf.

```
GET "GSDEFS.SR"

LET MAIN() BE [
 INITGRAPHICS()
 SETWINDOW(-750,-1000,750,1000)
                          Set up a window 1500 x 2000 units, centered at
                          the origin of the page coordinate space.
 OPENSEG(1)
 DRAW(TRIANGLE,"1",TRANSLATE,-100,100)
                          Draw the TRIANGLE symbol, positioned at (-
                          100,100) and labeled with the figure "1".
 DRAW(TRIANGLE,"2",SCALE,2,3,TRANSLATE,200,200)
                          Draw the triangle at (200,200) at 2/3 full size,
                          labeled "2".
 DRAW(TRIANGLE,"3",SCALE,6,4,3,ROTATE,30,TRANSLATE,50,-600)
                          Draw the triangle, scaled by 2 and 4/3 in the x-
                          and y-directions, rotated anti-clockwise through
                          30 degrees, and positioned at (50,-600). Label
                          this triangle "3".
 POSTSEG(1)
 UPDATE()
 ]

AND TRIANGLE(STR) BE [    Now define the TRIANGLE display procedure.
 MOVETO(0,0)
 DRAWTO(100,400)
 DRAWTO(200,0)
 DRAWTO(0,0)
 MOVETO(100,120)          Position the label.
 DRAWTEXT(STR)
 ]
```

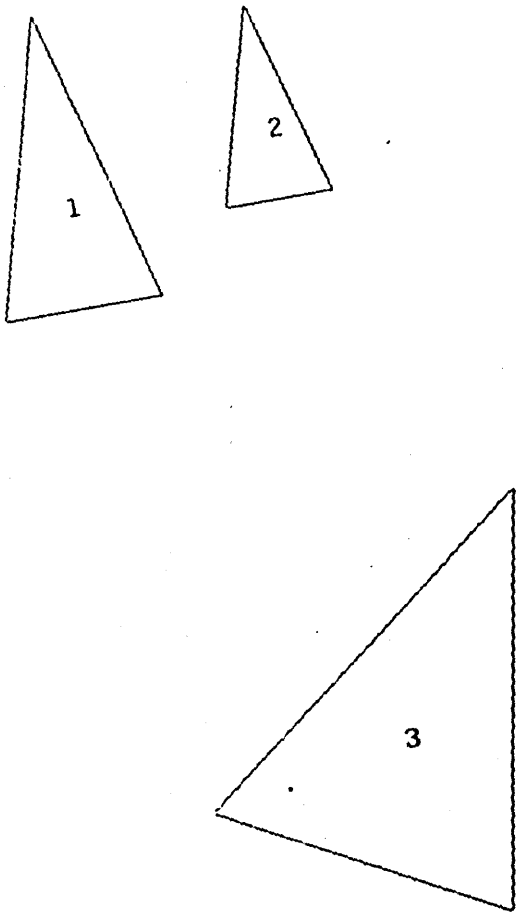


Figure 7.

GRAPHICAL INPUT

Three basic functions are provided by Pico for graphic input. The first accepts an (x,y) *position* from the tablet stylus or mouse; the second accepts a *stroke* generated in a single sweep of the stylus or mouse; the third accepts one or more strokes and attempts to *recognize* the character or symbol they represent. All three return x and y values converted to *page coordinates*.

Whenever one of these functions is called, the program waits until the stylus switch, or one of the mouse switches, is depressed and released by the user. The RECOGNIZE function waits an additional interval in case the user wishes to add more strokes. The input data is then returned as a pointer to a vector, which may be accessed with the aid of a BCPL structure provided for the purpose. Thus no input is ever received from these functions until the stylus or mouse switch has been pressed and released.

The three basic functions are as follows:

READPOSITION (). After the stylus or mouse switch is released, this function returns a pointer to a vector (V, say), containing:

in V>>EVENT.X) V>>EVENT.Y)	the page coordinates of the cursor when the switch was pressed.
in V>>EVENT.SWITCH	switch number (tablet always returns 1).

READSTROKE (). While the stylus or mouse switch is depressed, a trail of 'ink' records the path followed; after the switch is released, the function returns a pointer to a vector V containing:

in V>>EVENT.XLEFT) V>>EVENT.YBOTTOM) in V>>EVENT.XRIGHT) V>>EVENT.YTOP)	The page coordinates of the bottom left and top right corners of the rectangle enclosing the stroke.
in V>>EVENT.STROKE	A pointer to another vector, containing in its first word a count N of the number of recorded points, and then N pairs of x and y coordinates recording in page coordinates the path of the stylus or mouse (see Figure 8). V>>EVENT.STROKE is zero if the stylus or mouse did not move while the switch was depressed.

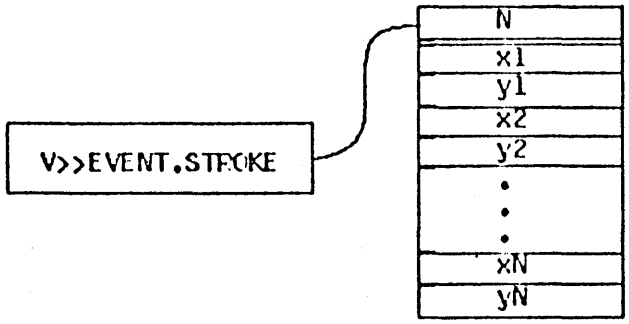


Figure 8.

RECOGNIZE (). This function continues to collect strokes until the switch remains released for at least one second (a parameter that may be altered, see Section IV). An attempt is then made to recognize the stroke or strokes by matching them against some predefined descriptions. RECOGNIZE returns a pointer to a vector V containing:

in V>>EVENT.XLEFT V>>EVENT.YBOTTOM V>>EVENT.XRIGHT V>>EVENT.YTOP	) The page coordinates of the) corners of the rectangle) enclosing the character) (see Figure 9).
in V>>EVENT.CODE	The numeric code of the recognized symbol (normally the ASCII code in the case of a character).
in V>>EVENT.CONF	The confidence, in the range 0 to 100, with which the symbol was recognized.

The vectors in which input information is returned are provided by Pico, and therefore should not be declared by the user program. Note that these vectors are re-used the next time an input function is called, so the relevant information must be extracted before another input function is called.

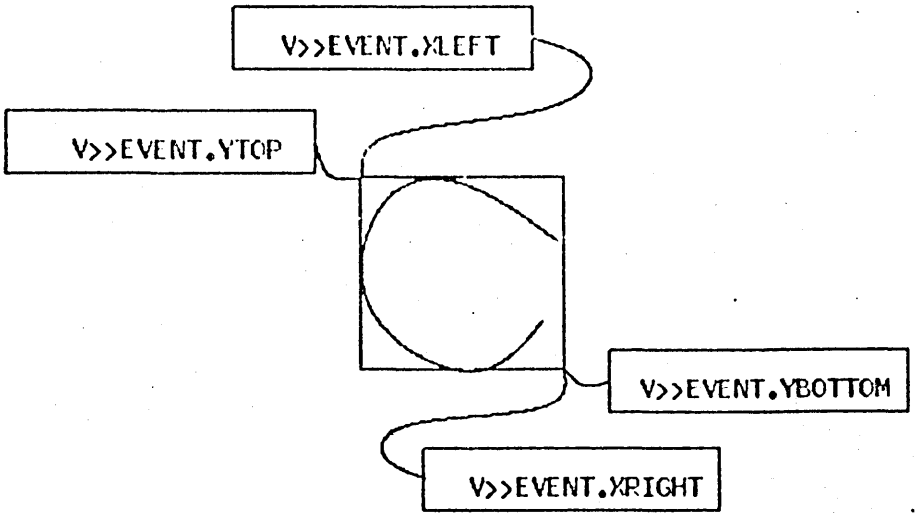


Figure 9.

Three other functions are useful for input:

HITDETECT (x,y [,x-tolerance,y-tolerance]). This function is useful for determining what the user is pointing at. It checks each displayed entity for overlap with the rectangle whose center is at (x,y) in screen coordinates, and whose "half-size" is x-tolerance by y-tolerance. If any entity overlaps, HITDETECT returns a pointer to a vector, V say, containing the following information:

in V>>HIT.SEGNAME	The name of the segment nearest to (x,y). In ambiguous cases, the highest name is returned.
in V>>HIT.DX)	The horizontal and vertical distance from

V>>HIT.DY) (x,y) to this nearest segment.

If tolerance values are omitted, HITDETECT uses the largest positive integer. If there is no overlapping entity, HITDETECT returns zero.

SETRECOGNIZER ("filename"). This function sets up the tables used by the RECOGNIZE function, by reading in a file of the given name. Previously a file of this name should have been created by using a training program (see Appendix 3). An argument of zero will clear the tables. This function returns FALSE if no file was found, TRUE otherwise.

CLEARINK (). Clears the ink from the screen, by performing an UPDATE.

HARD COPY

PLOTSEGS ("filename") This function writes out a file for the XGP, using the current contents of the display file. Hard-copy may be produced by sending this file to any XGP Nova, and then running the XPLOT program (See Section III). If no file name is given, unique names in the sequence P00.XB, P01.XB, ... P99.XB are used. PLOTSEGS always updates the screen contents as it generates the file.

MISCELLANEOUS

Several miscellaneous functions complete the kernel facilities of Pico:

SETFONT ("font-name"). This specifies the character font to use in all subsequent DRAWTEXT calls. The font-name is the name of a disk file in "CC" format (standard "CU" fonts may be converted to this format with a program described in Appendix 4). Several standard "CC" fonts can be found on the MAXC <GRAPHICS> directory. SETFONT returns FALSE if the specified font file could not be found; otherwise it returns TRUE.

CHARPROPERTIES (character-code). This function is used to furnish details about any character in the current font. It returns zero if the character is undefined in the font; otherwise it returns a pointer to a vector containing:

in V>>CHAR.WIDTH	The width of the character in screen coordinates.
in V>>CHAR.HEIGHT	The height of the character above the base line.
in V>>CHAR.DESCENT	The descent of the character below the base line.

RESETGRAPHICS (). This function should be called before returning to the operating system to ensure that the display is returned to its normal state.

INITGRAPHICS ([frame-space]). This function initializes Pico. Its

single optional argument may be used to create a larger or smaller run-time frame space for BCPL (default value is 1000 decimal). The function returns a pointer to a table of device-dependent parameters. These may be accessed with the aid of a BCPL structure definition and some manifest constant definitions, provided for the purpose.

V>>PICO.TYPE	Type of display device that this version of the graphics system will drive. This will be equal to STDALTO if configured for a standard Alto, RUNALTO if configured for an Alto with a run-length coded display device, or COLORNOVA if configured for the color video system (NOVA).
V>>PICO.TABLET	This is TRUE if a tablet is available on the machine in use, FALSE otherwise.
V>>PICO.XLEFT V>>PICO.YBOTTOM V>>PICO.XRIGHT V>>PICO.YTOP	Limits of the screen coordinate system.

GSTYPEFORM (format1,item1,format2,item2,...format*n*,item*n*).
This routine may be used for general-purpose string output to the console. It accepts from one to eight *items*, each preceded by a *format* in the shape of an integer from 0 to 10. The format number indicates how the item is to be displayed. Formats 0 and 1 treat the item as a string pointer and as a character code, respectively. Formats from 2 to 10 may be used to print integers to any radix in that range. For example,

```
GSTYPEFORM(0,"The octal value of ",10,100,1,$*N,0,"is ",8,100)
```

would generate:

```
The octal value of 100
is 144
```

On the Nova, the output of GSTYPEFORM is sent to the console; on any Alto, it is sent to the system area of the standard display.

Various pieces of ancillary software are included in the graphics system. These consist of some BCPL packages that Pico uses, and that the user may also find useful:

Free storage allocation. The INITGRAPHICS call "grabs" a substantial amount of available memory for use in building display files, font tables, etc. The user may make use of the free storage functions at any time after the INITGRAPHICS call has been issued. See Appendix 1 for documentation on these subroutines.

Floating point routines. These routines, described in Appendix 2, are available for users. Pico takes care to make all of its functions transparent to the contents of the floating-point accumulators.

SECTION III: USE OF PICO

COMPILING AND LOADING

Before a graphics program can be successfully compiled, loaded and run, two vital files must be on the user's disk-pack. These are:

GSDEFS.SR. This is the source file containing definitions of external procedure names, structures and constants used by Pico programs.

and one of the following:

APICO.BR. The version of Pico for use with the standard Alto display;
BPICO.BR. The version for use with Ben Laws' run-code display;
CPICO.BR. The version for use on the Color Graphics Nova.

A third file is generally essential:

DEFONT.CC. This is the standard Alto font in .CC format. A font file such as DEFONT will be needed if any text display is attempted. Additional font files are available.

These files, and all others relating to Pico, are stored on the <GRAPHICS> directory on MAXC. They may be copied to disk-packs using NEWMCA, MINX or any other path. To simplify the transfer process, three Dump files are kept on the <GRAPHICS> directory, containing the essential files for the three different displays. These three files, and their contents, are:

APICO.DM: APICO.BR, GSDEFS.SR and DEFONT.CC
BPICO.DM: BPICO.BR, GSDEFS.SR and DEFONT.CC
CPICO.DM: CPICO.BR, GSDEFS.SR and DEFONT.CC.

The procedure for compiling and loading a Pico program is as follows:*

1. Make sure that the three essential files are on your disk-pack. If they are not, copy (in binary mode) the appropriate .DM file from <GRAPHICS> and type:

   LOAD/V xPICO.DM

   where x is A, B or C as appropriate.

---

* Due to a temporary anomaly, the files APICO.BR and BPICO.BR cannot presently be loaded by the Alto BLDR. You must therefore substitute in their place about twelve separate .BR files. These files are for the time being included in APICO.DM and BPICO.DM, together with a .CM command file for use in loading. The command files are called APICL.CM and BPICL.CM. After completing steps 1 and 2, you should edit the command file to include the name of your program or programs, and then type @APICL.CM@ or @BPICL.CM@ to invoke loading. When this anomaly is eradicated, APICO.DM and BPICO.DM will be modified to match the description above.

2. Compile your source program. This program should include the statement GET "GSDEFS.SR" at its head.

3. Load the program with one of the following commands:

   On the standard Alto:
   BLDR 600/W <your program> APICO INITALTOIO

   On the run-code display Alto:
   BLDR 600/W <your program> BPICO INITALTOIO

   On the Color Graphics Nova:
   BLDR 600/W <your program> CPICO IO1 IO2

   The 600/W switch setting is necessary to increase the space for static variables.

GENERATING HARD COPY

After the program has generated a hard-copy file, the file must be copied over to an XGP and printed. The copying process should be performed with the aid of the Ethernet or MCA, whichever is appropriate. To print the file (let us say it is called P00.XB), type the following command to the XGP Nova:

```
XPLOT P00.XB
```

After the usual preamble, the XGP will produce a one-page printout. Several file-names may be included in the one XPLOT command in order to print more than one hard-copy file:

```
XPLOT P00.XB P01.XB P02.XB
```

Switches may be used to vary some of the plotting parameters: a number may be given in place of the file-name argument, followed by a slash, followed by a switch:

n/E	Sets enlargement to n (1,2,3,4; default 1)
n/L	Sets left margin to n (0-1200; default 100)
n/T	Sets top margin to n (0-2000; default 100)
n/S	Sets number of scan-lines per page (default 2000/enlargement)

SECTION IV: ADVANCED FUNCTIONS

The functions described in this section are not particularly difficult to use, but are probably likely to be used less frequently than those described in Section II. They fall into four categories: those for performing special transformations, those for handling input events, the DRAWCURVE function for drawing curves, and some miscellaneous other functions.

CURVE DRAWING

DRAWCURVE (x',y',x'',y'',x''',y''')
This function may be used in conjunction with MOVETO to draw parametric cubic curves. DRAWCURVE draws a curve from the present (x,y) position through a locus specified by the first, second, and third derivatives of the curve at the point (x,y). The curve traced out is the locus of (X,Y) defined parametrically by values of t between 0 and 1 in the equation:

$$X = x'''t^3/6 + x''t^2/2 + x't + x$$
$$Y = y'''t^3/6 + y''t^2/2 + y't + y$$

where (x,y) is the current position. Values of X and Y are transformed by whatever transformation is in effect, before the curve is displayed. The six parameters are pointers to packed floating-point numbers (two-word format).

Bob Flegal's knot-selection and spline-solving software is available (although not within Pico) for calculating derivative values from knot lists and other representations such as hand-drawn input, or points and boundary conditions.

Note that filled curves can be specified by calling BEGINFILL, following this with calls to MOVETO and DRAWCURVE, and terminating with ENDFILL.

TRANSFORMATIONS

This section describes the primitive transformation functions used to implement the DRAW function. Pico maintains a "current transformation matrix," a 3x3 homogeneous transformation applied to each coordinate pair; it also maintains the page-to-screen transformation parameters, and a "clipping region," a region of the screen that describes the limits of the visible display. Internally, Pico also keeps a temporary matrix (TM) that accumulates the effects of a set of transformations specified with TRANSLATE, SCALE and ROTATE. When a graphical primitive is called, the TM is postmultiplied by the current transformation matrix and the result replaces the current transformation matrix.

SETMATRIX (pointer-to-3x3-matrix). Sets the current transformation matrix from the matrix specified by the pointer. Whenever a new segment is opened, the matrix is automatically set to the identity matrix. The matrix is stored in packed floating-point format.

SAVEMATRIX (). Saves the current transformation matrix on a stack, and sets the TM matrix to the identity matrix.

RESTOREMATRIX (). Restores the current transformation matrix from the stack.

TRANSLATE (translation-x,translation-y). Postmultiply the TM by the matrix specifying translation through (translation-x,translation-y).

SCALE (sx [[,sy],sw]). Postmultiply the TM by the matrix specifying scaling by factors (sx/sw,sy/sw). If sy is omitted, the scale factors are sx/sw in both directions; if sy and sw are omitted, the scale factor is sx in both directions.

ROTATE (rotation-in-degrees). Postmultiply the TM by a matrix specifying rotation through the specified angle about the origin.

COS (integer-degrees). This function returns, in floating-point accumulator 1, the value of the cosine of the angle specified in the call.

The above functions are used in transforming information into page coordinates. As explained in Section II, the SETWINDOW function may be used to select a rectangular region of the page for display on the screen. Pico in fact allows control not only over this window, but also over the *viewport*, a rectangular region on the screen onto which is mapped all the information lying within the window:

SETVIEWPORT (xleft,ybottom,xright,ytop). This function specifies the limits, in screen coordinates, of the viewport within which subsequent graphical information is to be displayed on the screen.

Thus SETWINDOW effectively says, "show me this much of the page", and SETVIEWPORT says, "show it to me in this region of the screen". The SETWINDOW and SETVIEWPORT functions should be called *before* creating the display file segments on which they are to operate, much as SETWINDOW is called at the start of the example on page 14. Several different viewports may be used in generating one display, thus:

```
LET MAIN() BE [
  INITGRAPHICS()
  SETWINDOW(wx11,wyb1,wxr1,wyt1)      // set first window
  SETVIEWPORT(vx11,vyb1,vxr1,vyt1)    // and first viewport
  OPENSEG(k)
  ......                              // define first part of picture
  POSTSEG(1)
  SETWINDOW(wx12,wyb2,wxr2,wyt2)      // set second window
  SETVIEWPORT(vx12,vyb2,vxr2,vyt2)    // and second viewport
  OPENSEG(m)
  ......                              // define second part of picture
  POSTSEG(n)
  UPDATE()                            // update screen
  .......
```

INPUT

It is not always possible to predict which device will next generate an input to an interactive program. The user may type on the keyboard, point with the stylus or draw a stroke. The READPOSITION, READSTROKE and RECOGNIZE functions described in Section II are designed for applications where one can predict the order in which inputs occur. In cases where the order of inputs is *not* known, it is necessary to use a more general set of input routines that handle *events*. These routines collect events from the input devices and store them in a queue in their chronological order of occurrence. The program may call functions to wait for the next event to arrive in the queue, to determine what sort of event it was, to read the input data, and to delete the event from the queue.

An event is any one of the following:

1. A keystroke;
2. A stroke, generated by pressing and releasing the stylus or mouse switch; the device may or may not be moved while the switch is depressed.
3. A timeout event: the timer is always started on completion of a stroke, and stops either when it times out, or when another stroke is completed, whichever happens first. In the latter case, no event is generated. On completion of timeout, the character recognizer attempts to recognize all the strokes in the queue. If the queue is empty of strokes (i.e. stroke events have been deleted as they happen), no event is generated; otherwise the recognizer's best guess is returned in the event data.

Whenever an event occurs, all events of other types are automatically deleted from the queue. It is therefore unnecessary to delete events except to prevent invocation of the recognizer.

The following functions are provided for event-handling:

GETEVENT (). This function waits until the next event occurs, and then returns to the program a vector, V say, containing in V>>EVENT.TYPE the type of event (1, 2 or 3 as above). According to this value, the rest of V contains:

if V>>EVENT.TYPE equals 1 (keystroke):

V..EVENT.CODE	the ASCII code of the character;
V>>EVENT.KEYS	four words containing the status of the keyboard, in Alto format.

if V>>EVENT.TYPE equals 2 (stroke) or 3 (timeout):

V>>EVENT.CODE	the code of the recognized character, an eight-bit integer on which the recognizer has previously been trained (see Appendix 3); zero in type-2 events.

V>>EVENT.XLEFT	) the coordinates of the bottom left and
V>>EVENT.YBOTTOM	) top right corners of the rectangle
V>>EVENT.XRIGHT	) surrounding the stroke or strokes;
V>>EVENT.YTOP	) these are in *screen coordinates*.
V>>EVENT.INKED	TRUE in the case of an inked stroke, FALSE otherwise;
V>>EVENT.CONF	Confidence (0 to 100) with which the character was recognized;
V>>EVENT.STROKE	Pointer to stroke vector, in *screen coordinates*, stored as in Figure 8; type-3 events return a vector, in identical format, containing the coordinates of the stroke centers.
V>>EVENT.SWITCH	Switch number (tablet always returns 1).

DELETEVENT (). This function deletes the most recent event. If no events remain in the event queue, this function has no effect.

CLEAREVENTS (). This function clears the event queue of all events.

SETINPUTPARAMETERS (timeout-interval,ink-tolerance,sample-interval)
This function may be used to modify parameters controlling event-handling. It specifies the timeout interval for character recognition, in milliseconds (default is 1000), the distance to be moved by the stylus or mouse before inking begins (default is 4 screen units), and the minimum distance between points recorded in the stroke vector (default is 4 screen units). If any of these arguments are negative, the default values are inserted in their place.

SCREENTOPAGE (screenx,screeny,pointer-to-pagex,pointer-to-pagey).
This routine may be used to convert coordinates back from screen coordinates to page coordinates; it uses the most recent window and viewport settings. The third and fourth arguments should be pointers to two locations where the page-coordinate equivalents of the first two arguments are to be stored.

REFERENCES

[N&S]. W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics*, McGraw Hill, 1973.

[TENGR]. W.M. Newman and R.F. Sproull, "An Approach to Graphics System Design," Proceedings of IEEE, April 1974. (Available as CSL Graphics archive 3GR-013)

[NCC]. W.M. Newman, "An Informal Graphics System Based on the LOGO Language," Proceedings 1973 National Computer Conference.

Appendix 1: Free Storage Routines

A free-storage package is provided as an integral part of Pico. The package provides the following procedures for allocating and releasing variable size blocks:

INITFREESTORE (S).
Organizes the free storage space as one large block of size N, such that frame space of S words is if possible made available. INITFREESTORE sets up the appropriate globals:

FIRSTBLOCK:pointer to first block,
LASTBLOCK: pointer to last block,
AVAILMAX: maximum size of available block (see GETBLOCK),
AVAILTOTAL:total size of free space.

The last two variables are declared external in GSDEFS.SR.

INITFREESTORE returns the actual size of free storage, i.e. the initial setting of AVAILTOTAL.

GETBLOCK (N).
GETBLOCKX (N).
Returns a pointer to the first free word of a block of size N. N is the actual number of usable words. The actual size of the block will be between N and N+*E*, so that no blocks of size smaller than *E* -- a small number -- will exist (if N<*E*, N is set to *E*).

GETBLOCK and GETBLOCKX differ in the way error returns are handled. GETBLOCK returns 0 (i.e. *FALSE*) if no block of size N is available. The global AVAILMAX will then contain the size of the larger available block. Notice that the content of this location is only meaningful in this context. It is up to the caller to verify the value returned and decide whether to call again with a smaller value (smaller than AVAILMAX). GETBLOCKX will instead print a message and exit.

GETBIGBLOCK (N)
GETBIGBLOCKX (N)
Returns the biggest block of size greater than N. Error returns are as explained above.

PUTBLOCK(BLOCK-POINTER)
Returns a block to free storage, merging it into a larger block if possible. Also checks that the boundary tags are correct. The argument should be a pointer previously returned by GETBLOCK or GETBIGBLOCK.

TRIMBLOCK(BLOCK-POINTER,FREE-WORD-POINTER)
Returns to free storage the unused words at the end of a block if there are more than *E* of them),and resets the boundary tags. The first argument is the usual block pointer; the second argument is a pointer to the first unused word of the block.

The free storage allocation procedures use the "boundary tag" technique (Knuth, vol. #1, p.#435). A free block of storage is structured as follows:

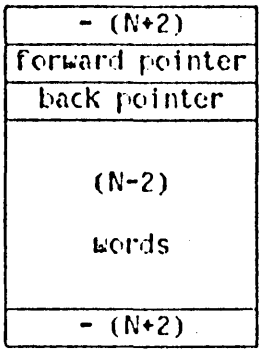


Figure 10.

A reserved block looks like:

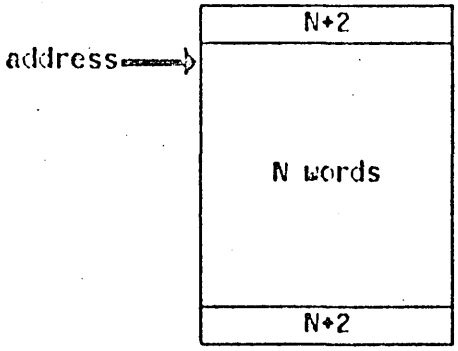


Figure 11.

Appendix 2: Floating-point Routines

The floating-point routines described below will run on a standard Alto (<GRAPHICS>FLOATALTO.BR) or on a NOVA (<GRAPHICS>FLOATNOVA.BR).

There are 16 floating-point accumulators, numbered 0-15. Each stores a 16-bit binary exponent and a 32-bit mantissa. These accumulators may be loaded, stored, operated on, and tested with the operations described below.

Conventions for the description: 'acnumber' refers to an accumulator number (0-15); 'arg' is either an accumulator number (if 'arg' < 16) or a pointer to a packed (2-word format) floating point number; 'ptr-to-fp-number' is a pointer to a packed (2-word format) floating point number. If a function returns a value, the symbol "==>" is used to show the result; functions that do not have the "==>" following them return their first argument as a result.

FLD (acnumber,arg)
Load the specified accumulator from source specified by arg.

FST (acnumber, ptr-to-fp-number)
Store the contents of the accumulator into a 2-word packed floating point format. Error if exponent is too large or small to fit into the packed representation.

FTR (acnumber) ==> integer
Truncate the floating point number in the accumulator and return the integer value. Error if number in ac cannot fit in an integer representation.

FLDI (acnumber,integer)
Load-immediate of an accumulator with the integer contents (signed 2's complement).

FNEG (acnumber)
Negate the contents of the accumulator.

FAD (acnumber,arg)
Add the number in the accumulator to the number specified by arg and leave the result in the accumulator.

FSB (acnumber,arg)
Subtract the number specified by 'arg' from the number in the accumulator, and leave the result in the accumulator.

FML (acnumber,arg) [also called FMP]
Multiply the number specified by 'arg' by the number in the accumulator, and leave the result in the ac.

FDV (acnumber,arg)
Divide the contents of the accumulator by the number specified by arg, and leave the result in the ac. Error if attempt to divide by zero.

FCM (acnumber,arg) ==> integer
Compare the number in the ac with the number specified by 'arg'. Return

```
-1 IF ARG1 < ARG2
 0 IF ARG1 = ARG2
 1 IF ARG1 > ARG2
```

FSN (acnumber) ==> integer
Return the sign of the floating point number.

```
-1      if sign negative
 0      if value is exactly 0 (quick test!)
 1      if sign positive and number non-zero
```

FLDV (acnumber,ptr-to-vector)
Read the 4-element vector into the internal representation of a floating point number. The 4-word vector is arranged as follows: a word for sign (-1 means negative; 0 positive), a word of signed exponent; two words of mantissa.

FSTV (acnumber,ptr-to-vector)
Write the accumulator into the 4-element vector in internal representation.

The 2-word packed format is:

The first word is:

```
sign -- 1 bit
exponent -- excess 128 format (8 bits)
        will be complemented if sign negative
mantissa -- first 7 bits
```

The second word is:

```
mantissa -- 16 more bits
```

Note this format permits packed numbers to be tested for sign, to be compared (by comparing first words first), to be tested for zero (first word zero is sufficient), and (with some care) to be complemented.

If you wish to capture errors, put the address of a BCPL subroutine in the static FPerrprint. The routine will be called with one parameter:

```
0       Exponent too large -- FTR
1       Exponent too large -- FST
2       Dividing by zero -- FDV
3       Ac number out of range (any routine)
```

The floating-point routines use a work area, pointed to by the static FPwork, for storage of all accumulators, etc. The first word of that area is its length. If FPwork is changed to point to another work table of adequate length, the subroutines will use it for working area. This permits subroutines to save and restore the contents of the floating-point accumulators.

Appendix 3: Training the Character Recognizer

A program called TRAINER has been written to enable users to set up files for the recognizer. The compiled version of this program, suitable for use on a standard Alto, is TRAINER.DM (a dump file) on <GRAPHICS>.

To start the program, type TRAINER. You will be asked if you want to add to an existing file: if so, type the file name, followed by <return>; if not, just type <return>. Then a display will appear similar to the one overleaf, and you will be asked to draw a character.

Every time you draw a character, TRAINER will try to recognize it. If it fails, it will say so, and you should point to the letter that corresponds to the symbol you drew. If it succeeds, you may point anywhere to confirm correctness. If you wish to train to a character other than an upper-case letter, point to the @ character and then give the required octal value. If you draw an inaccurate symbol, you may reject it by pointing to the "rej" symbol.

The three targets at the top of the screen are to be used to clear the screen of ink, to file the results of a training session, and to exit from TRAINER.

As training proceeds, large amounts of memory may be used up. You can compact by writing out a file, then starting TRAINER again and reading in the file. You should do this if areas of the screen become unreceptive to ink.

WRITING YOUR OWN TRAINER

Three special functions exist, in a file called GSTRAIN.BR on <GRAPHICS>, that may be used in the construction of training programs. These three functions are:

INSERTPROPS (). If called after calling RECOGNIZE or after receiving a type-3 event, this function saves the properties used by the recognizer. They may later be inserted in the tables by means of the INSERTCODE function.

INSERTCODE (code). This function enters into the recognizer tables the properties saved by INSERTPROPS, identifying them with the specified eight-bit code.

WRITEPROPTAB (filename). This function writes out the recognizer tables onto a file of the specified name.

exit clean file

@ A B C D E F G H I J K L M N O P Q R S T U V H X Y Z rej

Figure 12.

Appendix 4: Creating .CC Font Files

Font descriptions in a format acceptable to the graphics system (hereafter called CC format) can be created from any font in ".CU" format. All fonts currently created at PARC are available in this format (consult Ben Laws, or the <LAWS> directory on MAXC, for available styles). A LISP program is used to create CC files from this format. The dialog below illustrates the use of this program (characters typed by the user are underlined):

```
@LISP

INTERLISP-10 xxxx

Good afternoon, faithful.

←LOAD(<GRAPHICS>CHAIN.COM)
...
CHAINFNS
←CONVERT()
Filename in .CU format to be converted ....GACHA.CU
GACHA.CU;1
Filename for .CC version....GACHA.CC
GACHA.CC;1
Baseline for this font ....4
...
...
..<prints each character code in decimal as it is processed>
...
←LOGOUT()
@
```

If error messages are generated, consult the Graphics Group.

Appendix 5: The XPLOT File Structure

An .XB file consists of a header, followed by a texture table, followed by any number of scan-line-streams. Each scan-line is processed in order of its appearance in the file, from top to bottom of the page.

The header is a 4-word block that specifies:

word 0:	enlargement	(1)	/E
word 1:	left margin	(100)	/L
word 2:	top margin	(100)	/T
word 3:	scan-line-streams/page	(2000/enlargement)	/S

These specify the coordinates (in XGP resolution units) of the upper left-hand corner of the picture, the enlargement (integer from 1 to 4), and the number of scan-line specifications in the file that should fall on one the XGP page. If an entry is zero, it is replaced by the default listed in parentheses. The entries may also be overridden by switches specified in the command line.

A texture table is a count, n, followed by $2n$ words. The first n are called the T table, the second n the W table.

A scan-line-stream is a count, n, followed by abs(n) words of either run or bit-map data. If n 0, the words are interpreted as bits to be given to the the XGP (high order bit of a word appears left-most on the page). If $n > 0$, the words are interpreted as runs: each word specifies a pattern (H) and a run (R). The high order 8 bits are H, the low order R. The idea is that the pattern specified by H will be repeated for R bit positions on the scan-line. The next (H,R) pair will pick up where the previous left off.

A run is specified by H and R as follows: H is an index into the T and W tables. T[H] is a bit sequence to send to the the XGP; W[H] is the width (or modulus) of the bit sequence (must be between 9 and 16 inclusive). The algorithm for displaying runs (at enlargement 1) is:

```
while R  W[H] do begin
    show the high-order W[H] bits of the pattern T[H].
    R ← R-W[H]
    end;
if R neq 0 then show the first R bits of the pattern T[H]
```

Note: for increased efficiency, H=0 always corresponds to the blank sequence (i.e. white space on the the XGP).

Handy constants: an the XGP page is about 1300 dots across and 2100 scan-lines long. Horizontal and vertical resolutions are thus about 200 dots per inch.