CS 595: GRADUATE PROJECT
FALL 1985

JORGE CEREIJO CACEIRO
# TABLE OF CONTENTS

1: ABSTRACT ................................................................. 1
2: PURPOSE ................................................................. 2
3: ENVIRONMENT ............................................................ 3
4: BACKGROUND ............................................................. 4
5: METHODS ................................................................. 6
6: PROBLEMS ENCOUNTERED ............................................. 12
7: ASSEMBLER OPERATION ............................................... 14
8: LINK PROCESS .......................................................... 15
9: CREATING AND UPDATING LIBRARIES ............................ 16
10: UTILITY FUNCTIONS .................................................. 20
    REFERENCES .......................................................... 42

APPENDICES

APPENDIX A: PL/I PROGRAMS LISTING & OUTPUTS

APPENDIX B: ASSEMBLER PROGRAMS LISTING
ABSTRACT

This system is the result of a graduate project at Corpus Christi State University, and covers the designing, coding and implementation of a graphics library. It is written in 8086/8088 assembler language for a PL/I-86 compiler from Digital Research for the IBM Personal Computer.

The library provides the user with the following utilities:

1- Select Resolution.
2- Draw a polygon.
3- Screencolor.
4- Pen color.
5- XOR Function.
6- Fill irregular areas.
7- Clear the screen.
8- Print "Strings".
9- Line drawing functions.
10- Move functions.
11- Fill rectangles function.

These functions reside in a library called "GRAPHICS" which will interface (through a linkage-editor) with any PL/I program.
PURPOSE

This project was undertaken as partial fulfillment of the requirements for a Master of Science degree in Computer Science at Corpus Christi State University, Corpus Christi, Texas during the fall semester of the school year of 1985.

The objective of the project was to develop a group of utility programs that would give graphics capabilities to a PL/I programs compiled with the PL/I-86 compiler from Digital Research for the IBM Personal Computer.

This group of programs is organized into a library which can be linked with any PL/I program. The library is intended to be used by any computer science professional with interest in PL/I programming, with some added advantages not available at the present time.
ENVIRONMENT

This project is implemented on an IBM-PC microcomputer running under any PC-DOS version and requires at least 256K of memory and two double sided disk drives (360k). The programs can be run with or without a printer, but the use of any IBM-PC compatible printer would be required if the user decides to combine the use of printed reports with the screen output graphics.

The programs that are contained in the graphics library were written using RASM-86 assembler which is an 8086/8088 assembler provided with the PL/I compiler by Digital Research. The PL/I programs that use this library were written using the PL/I compiler from Digital Research.

The total space used by the graphics library is 7k.
BACKGROUND

One of the reasons for choosing this graphics library project was my own personal interest in working with the PL/I compiler for business applications. The addition of this graphics library will provide future users with resources that were not available before. Moreover, this researcher was interested in acquiring the experience and knowledge that the development and implementation of this library provides.

PL/I is a powerful language that can be used to solve both business and scientific applications. Computer graphics is a topic of rapidly growing importance in the computer field. It has always been one of the most visually spectacular branches of computer technology, producing images whose appearance and motion make them quite unlike any other form of computer output. Computer graphics is also an extremely effective medium for communication between man and computer.

Finally, Assembly language has always been the fastest and most powerful language available for a given computer. It is essential in programs where pure speed of operation is important, such as graphics and sorting. It is also the only language that can make use of all hardware features of a particular machine. With higher-level languages, such as BASIC or Pascal, the programmer is always insulated from the computer by the language itself, having as much latitude as the one conveyed by the language itself.
For all these reasons, this researcher decided to combine these three powerful tools (PL/I, Assembler and graphics) to improve the existing version of PL/I.
DESCRIPTION OF THE MAJOR COMPONENTS

MOTOROLA 6845 CRT CONTROLLER.

This device provides the necessary interface to drive a raster scan CRT. How the CRT knows when to turn the beam on and off is governed by reading the pattern of bits stored in the display buffers. Both the Monochrome and the Color/Graphics board contain memory chips, called display buffers, that are used to store the definition for the screen display. The Monochrome board has a 4K buffer and the Color/Graphics board has a 16K buffer.

THE EQUIPMENT FLAG

The ROM BIOS listing in the IBM Personal Computer Technical Reference manual contains a section called "ROM BIOS Data Areas." The segment address of this section of code is 40h. At offset address 10h in this segment is the equipment flag. (That's absolute address 00410h.) When the PC is first turned on, a ROM BIOS routine reads the settings of your peripheral hardware switches and stores their on/off states in the equipment flag location. To change displays, we must "fool" the BIOS routines into thinking that the switch settings have changed. This is done by putting new values in certain bits of the memory (Figure 2.). Specifically, we change bit 4 from 1 to 0 if we want to change from monochrome to color. Bit number 5 is always
set on, unless we want to change to the 40 character mode, which means that the text will appear in forty columns on the screen instead of 80. The other bits do not apply to the video display at all, but their settings must not be lost when bits 4 and 5 are changed.

**DISPLAY BUFFER**

The display buffer resides in the processor-address space, starting at address hex B8000. It provides 16K bytes of dynamic read/write memory. A dual-ported implementation allows the processor and the graphics control unit to access the buffer. The processor and the CRT control unit have equal access to his buffer during all modes of operation, except in the high-resolution alphanumeric mode.

**CHARACTER GENERATOR**

This attachment utilizes a ROM character generator. It consists of 8K bytes of storage that cannot be read from or written to under software control. This is a general-purpose ROM character generator with three different character fonts. Two character fonts are used on the color/graphics adapter: a 7-high by 7-wide double-dot font and a 5-wide by 7-high single-dot font.
ROM BIOS Video Routine

The BIOS is a set of 8088 programs that are stored permanently in the Personal Computer. When the computer is turned on, these programs receive control and initialize the entire computer system. In addition, they provide the minimum software support necessary to control the various devices that may be attached to the computer.

Graphics Mode

Besides the alphanumerical mode, the IBM-PC or any compatible computer can be placed in the graphics mode. In the graphics mode, pictures may be drawn on the screen. Every picture is made up of a set of dots. The graphics mode allows us to draw individual dots in the screen. There are two screen resolutions available in the graphics mode. In the high-resolution mode, an image with 640 horizontal by 200 vertical dots may be produced; each dot may be either black or white. In the medium-resolution mode, an image with 320 horizontal by 200 vertical dots may be produced. The vertical resolution for both modes is the same. The horizontal resolution mode is twice as fine as the horizontal resolution for the medium-resolution mode. However, color dots can be produced in the medium-resolution mode.

In the medium-resolution mode, each dot may be one of four pre-selected colors. One of the colors may be selected from the 16 colors in table 2. This color is called the background color.
<table>
<thead>
<tr>
<th>INVOKE VIA</th>
<th>INPUT REQUIREMENTS</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT 10H</td>
<td>AH = 0, AL = mode; AL = 4; 320x200 color; AL = 5; 320x200 bw; AL = 6; 640x200 bw</td>
<td>Set display mode</td>
</tr>
</tbody>
</table>

**TABLE 3.**
<table>
<thead>
<tr>
<th>COLOR</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK</td>
<td>0</td>
</tr>
<tr>
<td>BLUE</td>
<td>1</td>
</tr>
<tr>
<td>GREEN</td>
<td>2</td>
</tr>
<tr>
<td>CYAN</td>
<td>3</td>
</tr>
<tr>
<td>RED</td>
<td>4</td>
</tr>
<tr>
<td>MAGENTA</td>
<td>5</td>
</tr>
<tr>
<td>BROWN</td>
<td>6</td>
</tr>
<tr>
<td>WHITE</td>
<td>7</td>
</tr>
<tr>
<td>GRAY</td>
<td>8</td>
</tr>
<tr>
<td>LIGHT BLUE</td>
<td>9</td>
</tr>
<tr>
<td>LIGHT GREEN</td>
<td>10</td>
</tr>
<tr>
<td>LIGHT CYAN</td>
<td>11</td>
</tr>
<tr>
<td>LIGHT RED</td>
<td>12</td>
</tr>
<tr>
<td>LIGHT MAGENTA</td>
<td>13</td>
</tr>
<tr>
<td>YELLOW</td>
<td>14</td>
</tr>
<tr>
<td>HIGH INTEN. WHITE</td>
<td>15</td>
</tr>
</tbody>
</table>
The other three colors may be either red, green, and yellow or cyan, magenta and white. Even though there are many different colors to choose from, only four may appear on the screen at the same time.

The color adapter board may be initialized to one of the resolutions described above by using the BIOS call shown in table 3.

**The Color Graphics Memory Map**

The contents of the adapter memory determine the image on the screen. The medium-resolution mode allows 320 by 200, or 64000, dots. Since the adapter has 16K of memory, four dots must be represented in one byte of adapter memory. This leaves two bits to represent each dot. Two bits can have four different combinations, which map to the four possible colors in the medium-resolution mode.

Each dot on the screen has a row and column coordinate. The memory that maps to the odd row numbers starts at offset address 0. The memory that maps to the odd row numbers starts at offset address 2000H. See figure 3A. It takes 80 bytes to represent a row of dots. Each byte is broken up into four sets of two bits each, as shown in figure 3B. Each of these two-bit sets is called a dot position. The dot position range from 0 to 3. Each dot position determines the color of a particular dot.

To set a particular dot to a color, we must first get the offset address of the beginning of the 80 bytes that represents the row that contains the dot.
Then the particular byte within the 80 bytes must be found, and the correct two bits within that byte must be set to the bit combination that represents the desired color.

The row number of the desired dot coordinate is used to determine the offset address of the base of the 80 bytes. This is shown in figure 4. The lowest bit of the row number determines which half of memory to access. The value of the remaining 7 bits (0-99) is multiplied by 80 (the number of bytes that represent a row) to get the offset within the correct half of memory.

The column number is used to determine the correct byte and bits within the 80 bytes of interest. This is shown in figure 5. The column number is a nine-bit ranging from 0 to 319. The high-order seven bits, which can range from 0 to 79, determine which byte within the 80 bytes (selected above) contains the correct bits to set. To access the correct byte, we can add the value of the offset address obtained from the row number to the value contained in the seven bits. The result will be the offset address for the correct byte to change.

The bits to be used in the selected byte are determined by the low-order two bits of the column number. These low-order two bits form a value from 0 to 3. This value represents the dot position. If the low-order two bits of the column number are 00, then the high-order two bits of the selected byte must be changed. If the low-order two bits of the column number are 11, then the low-order two bits of the selected byte must be changed.
OBTAINING THE 80-BYTE OFFSET ADDRESS

MAJOR OFFSET ADDRESS

0 = Initial Offset of 0000H
1 = Initial Offset of 2000H

SECONDARY OFFSET

BITS 1 - 7:

7-BIT VALUE (0-99)
Add 80 times this value to the offset address.

FIGURE 4.
Determining the bits to set in 340 x 200

(COLUMN NUMBER 0-319)

BITS 1-0 = DOT POSITION WITHIN BYTE
BITS 8-2 = COLUMN OFFSET
7 BIT VALUE (0-79)
ADD THIS VALUE TO THE 80-BYTE OFFSET ADDRESS TO GET THE CORRECT BYTE OFFSET ADDRESS.

Figure 5.
If we want to change a single dot on the screen, we must make sure that we do not change the color of the other dots that are represented in the same byte. When making a change, we must first read the appropriate byte from memory. The desired two bits can then be modified. After this is done, the byte can be written back to the correct offset address.

The mapping for the 640-by-200 display image is similar to what we have just described for 320-by-200 medium-resolution. There are still 80 bytes of memory needed to represent each row of dots. The base of the 80 bytes is calculated the same way as before. The only difference is that now the column number is a value from 0 to 639. Each byte represents 8 dots. One bit per dot is used. A 0 is used for black and a 1 is used for white. There are not enough bits for any color information.

The selection of the correct bit to set within the 80 bytes is shown in figure 6. As before, the high-order seven bits determine the correct byte within the 80 bytes. The three low-order bits form a value from 0 to 7. This value determines which bit in the byte to set. The high-order bit is location 0, and the low-order bit is location 7.
Determining the Bit to Set in 540x250

(Column Number 0-639)

9 8 7 6 5 4 3 2 1 0

Bits 2-0 = Dot Position Within Byte
Bits 9-3 = Column Offset
7 Bit Value (0-79)
Add This Value to the 80-Byte Offset Address to Get the Correct Byte Offset Address.

Figure 6.
Addressing Example for Medium Resolution

Row number = 17.
Column number = 17.

ROW NUMBER = 17

First Bit = 1
then GOTO
ODD SCAN LINES

First Offset: 16 x 80 = 1280 byte offset

Column Number

Second offset = 1280 + 8 = 1288

Bits 0-1 represent the pixel: 01
indicates 2nd pixel

Therefore the second pixel represented by
byte 1288 is turned on.
Problems Encountered

RASM-86.

1.) The assembler compiler is not as flexible as other assembler compilers for the 8086/8088 chip.

2.) The documentation is poorly written, making it more difficult to debug problems.

3.) Absolute memory addressing, for addresses out of the program segment is not done like in other 8086/8088 assembler compilers and the appendix dedicated to the RASM-86, has no information about this topic.

4.) A single memory to memory data movement instruction is not provided, so 2 instructions memory to register and register to memory must be used.

PL/I-86.

1.) The documentation for the PLI-86 compiler is sometimes misleading. Updates to the manual are found in the "HELP" files on disk, but the manuals does not mention this. A good example of this is the "CALL" instruction. The instruction must be

< CALL proc-name (argument-list) > but the manuals for the cases of procedures with no argument-list simple omitted the argument-list. The reality is that procedures with no argument-list have to use null-parameter list instead. The correction is noted in a "HELP" file, but this was discovered only after many hours of debugging.
2.) Another example can be found in the section used to explain internal-data representation. The internal representation of character-strings does not behave as explained in the documentation.

   a.) Characters are transposed in memory for example: CALL STORE ("ABCD") is represented internally as BADC with a CNTR-END ASCII code at the end.

   b.) For character-strings with "n" bytes declared with the VARYN-condition, PL/I is supposed to reserve (n+1) bytes where the first byte is used to specify the length of the string. This doesn't happen at least in the case where strings are handle with RASM-86 assembler routines.
RASM-86

ASSEMBLER OPERATION

RASM-86 processes an 8086 assembly language source file in three passes and produces an 8086 machine language object file. RASM-86 can optionally produce three output files from one source file.

SOURCE FILE ----> RASM-86 -------> a.) LIST FILE (Name.LST)  
(Name.A86)  
 b.) OBJECT FILE (Name.OBJ)  
c.) SYMBOL FILE (Name.SYM)

The LST file contains the assembly language listing with any error messages. The OBJ object file contains the object code in Intel 8086 relocatable object format. The SYM symbol file lists any user-defined symbols.

The three files have the same filename as the source file. For example, if the name of the source file is BIOS88.A86, RASM-86 produces the files BIOS88.OBJ, BIOS88.LST, and BIOS88.SYM.

Invoking RASM-86

Invoke RASM-86 with a command in the form:

RASM86 source file

The filespec has the form:

[d:]filename[.typ]

where

d: is an optional drive specification denoting the source file's location. The drive specification is not needed if the source is on current drive.
filename is a valid filename of 1 to 8 characters.
typ is a valid filetype of 1 to 3 characters, usually A86.

RASM-86 accepts a source file with any filetype. If you omit the filetype from the command line, RASM-86 searches the directory for the specified filename with the filetype A86.

The following are some examples of valid RASM-86 commands:
A> rasm86 b:bios88
A> rasm86 bios88.a86
A> rasm86 d:test

THE LINK PROCESS

LINK-86 is the Digital Research linkage editor that combines relocatable object files into a file that runs under DOS. The object files can be produced by Digital Research's 8086 language translators such as RASM-86, PL/I-86, and CB86, or by other translators that produce object files using a compatible subset of the Intel 8086 object module format.

LINK-86 accepts two types of object files. The first type is an object file containing a single object module. This type generally has the filetype OBJ, and is produced by a language translator. The second type is a library file which is an indexed library of object modules. A library file has a filetype L86, and is generated by the library manager, LIB-86, in the Intel 8086 object module format. LINK-86 can search such a library file and select only those modules needed by the other
programs being linked.

LINK-86 produces three files:

1.) A Run (EXE) File
2.) A Symbol Table (SYM) File
3.) A Map (MAP) File.

The EXE file contains a memory image of the program that runs directly under DOS. The SYM file contains a list of symbols from the object files, and their offsets, and is suitable for use with a symbolic instruction debugger. The MAP file contains information about the layout of the EXE file.

**Invoking LINK-86**

You invoke LINK-86 with a command of the form:

`LINK86 {file =} file1 {,file2, ...... ,filen}`

If you enter a filename to the left of the equal sign, LINK-86 creates the output files with that name and the appropriate filetypes. For example, the command

A> link86 myfile = parta,partb,partc

creates MYFILE.EXE and MYFILE.SYM.

If you omit the filename, LINK-86 creates the output files using the first filename in the command line. For example, the command

A> link86 parta,partb,partc

creates the files PARTA.EXE and PARTA.SYM.

**CREATING AND UPDATING LIBRARIES**

Lib-86 is a utility program for creating and maintaining library files that contain 8086 object modules. These modules can
be produce by Digital Research's 8086 language translators such as RASM-86, PL/I-86, and CB86, or by any other translators that produce modules in Intel's 8086 object module format.

You can use LIB-86 to create libraries, as well as append, replace, select, or delete modules from an existing library. You can also use LIB-86 to obtain information about the contents of library files.

**LIB-86 Operation.**

When you invoke LIB-86, it reads the indicated files and produces a Library file, a Cross-reference file, or a Module map file as indicated by the command line. When LIB-86 finishes processing, it displays the Use Factor, which is a decimal number indicating the percent of the available memory that LIB-86 used during processing.

**Creating and Updating Libraries**

You can create or update libraries using a command line of the general form:

```
LIB86 <library file> = <file 1> [[options]]

{<file 2>,.....<file n>].
```

LIB-86 creates a Library file with the filename given by <library file>. If you omit the filetype, LIB-86 creates the Library file with filetype L86. LIB-86 reads the files specified by <file 1> through <file n> and produces the library file. If <file1> through <file n> do not have a specified filetype, LIB-86 assumes a default filetype of OBJ.
Creating a New Library.

To create a new library, enter the name of the library, then an equal sign followed by the list of the files you want to include, separated by commas. For example,

A>lib86 newlib = a,b,c
A>lib86 newlib.186 = a.obj,b.obj,c.obj
A>lib86 math = add,sub,mul,div

The first two examples are equivalent.

Adding to a Library.

To add a module or modules to an existing library, specify the library name on both sides of the equal sign in the command line. The library name appears on the left of the equal sign as the name of the library you are creating. The name also appears on the right of the equal sign, with the names of the other file or files to be appended. For example,

A>lib86 math = math.186,sin,cos,tan
A>lib86 math = sqrt,math.186

Deleting a Module.

The command for deleting a module or modules from a library has the general form:

lib86 <new library> = <old library> [DELETE [<module specifiers>]]

where <module specifiers> can contain either the names of single modules, or congeries of modules, which are specified using the name of the first and the last modules of the group, separated by a hyphen. For example,
A>lib86 math = math.186 [delete [sqrt]]
A>lib86 math = math.186 [delete [add, sub, mul, div]]
A>lib86 math = math.186 [delete [add - div]]
Call proc-name(--,--,--,--,--,--);

Call A(x1,x2,x3);

Call B();

2000

<table>
<thead>
<tr>
<th>1000 :</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>B%</td>
<td></td>
</tr>
</tbody>
</table>

2000 : address of parameter 1
3000 : address of parameter 2
4000 : address of parameter 3
5000 :
6000 :
7000 :
8000 :
9000 :
10000 : parameter n
UTILITY FUNCTIONS
Function CLARES

This function clears the color graphics screen and also the memory locations used to store the X and Y values.

Input: None

Output: Just to the screen.

Format: CALL CLARES()

Description: One of the nice options from your PL/I language program is to be able to clear the screen. In BASIC you do this by executing a CLS instruction, but there is no such instruction in PL/I. However, a ROM routine makes it fairly easy. This function is called Scroll Active Page Up. This routine thinks in the screen as a window with many rows high and so many columns wide. The upper left-hand corner of the screen is the starting place for these measurements, since it is here that both the column and row numbers are zero. The row numbers run down the screen from 0 to 24, and the column number run across the screen from 0 to 79 (since it is a 25 by 80 screen). The Scroll Up function requires coordinates for two corners of a rectangular window: the upper left-hand corner, and the lower right hand corner. These coordinates must be placed in the following registers:

CH <----- Upper left row
CL <----- Upper left column
DH <----- Lower right row
DL <----- Lower right column
The AH register must hold a 6 to designate the Scroll Up function, and AL must be 0, to inform the function to clear the screen. Some languages that provide graphics capabilities clear the screen by turning "off" and "on" the specific resolution-mode, this option is available in our system.
**Function** COLOR

This function activates the medium-resolution (320 x 200) mode.

**INPUT:** NONE  
**OUTPUT:** NONE  
**FORMAT:** CALL COLOR()

**Description:** This function modifies the Equipment flag, and sends the appropriate message to ROM BIOS, thus switching your system into the 320 x 200 color resolution. The ROM BIOS routine used is 10h and register "AL" must be equal to 4. The character resolution is 25 x 80 in this particular resolution-mode.
**Function HIGH**

This function activates the high-resolution mode (640 x 200).

- **INPUT:** NONE
- **OUTPUT:** NONE
- **FORMAT:** CALL HIGH()

Description: This function modifies the equipment flag, selecting the 80 x 25 character resolution, and sends the appropriate message to ROM BIOS, thus switching your system into the 640 x 200 resolution-mode. The ROM BIOS routine being used is 10h and the "AL" register must be set to 6, which specifies the high resolution-mode. The only colors allowed in this mode are black and white.
Function MONO

This function activates the monochrome resolution-mode.

INPUT: NONE
OUTPUT: NONE
FORMAT: Call MONO()

Description: This function modifies the Equipment flag selecting the 80 x 25 character resolution and sends the appropriate message to the ROM BIOS, thus switching system into the alphanumeric mode. The ROM BIOS routine begin used is 10h and the "AL" register must be set to 2.
**Function PLOTDOT**

This routine plots a dot in an \((x,y)\) coordinate on the screen.

**INPUT:** Upon entry:
- x-coordinate of the point
- y-coordinate of the point

**OUTPUT:** Just to the screen

**FORMAT:** Call PLOTDOT\((x,y)\)

**Description:** This routine plots a point on the screen, regardless of the resolution-mode (320x200 or 640x200). The pixel at that specified location is given a specified color depending on the pen color.
**Function XORS**

This routine plots a point on the screen regardless of the resolution mode using the "exclusive or" operation. The pixel at the specified location is colored with a color obtained by "exclusive oring" its original color with a specified color.

**INPUT:** Upon entry:
- x-coordinate of the point
- y-coordinate of the point

**OUTPUT:** Just to the screen.

**Description:** This function is performed by making a few modifications on the ROM BIOS function that plots a point on the screen. The "AL" register that is used to specify the color will contain a 1 in Bit 7 to specify that the color value is exclusive OR'D with the current contents of the pixel.
Function WCHAR

This routine plots a character in the current (x,y) coordinates.

INPUT: Upon entry:
Character to be printed

OUTPUT: Character at (x,y) coordinates.

FORMAT: Call WCHAR('character')

Description: For read/write character interface while in graphics mode, the characters are formed from a character generator image maintained in the System ROM. Only the 1st 128 characters are contained there.
Function  WSTRING

This routine prints a message on the graphics screen, using
the WCHAR routine.

INPUT:  Upon entry:
    String to be displayed on the screen.

OUTPUT: String is displayed on the screen.

FORMAT: Call WSTRING(string)

Description: This function displays a message on the screen. It
starts by displaying the first character of the
string, then it continues to the next one and so on
until the last character is displayed. The length of
the string has to be defined by a DECLARE statement.
Variable strings are not permitted, and fixed length
strings are recommended.
Function PENCOLOR

This routine selects the color to be used for displaying dots and characters on the screen.

INPUT: Upon entry:
   color specification

OUTPUT: NONE.

FORMAT: Call PENCOLOR(color)

Description: This routine is used to specify the foreground color that makes up the character image. Each dot may be one of four colors. The first color may be one of the 16 possible background colors. This is called the background color. The other three colors may be either red, green, and yellow or cyan, magenta, and white. Only one set of four colors is available for a given display image. In the high-resolution mode, a black-and-white image with 640 horizontal by 200 vertical dots is displayed. The numeric value to represent each color is as follows:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREEN</td>
<td>------</td>
</tr>
<tr>
<td>RED</td>
<td>------</td>
</tr>
<tr>
<td>YELLOW</td>
<td>------</td>
</tr>
<tr>
<td>BLUE</td>
<td>------</td>
</tr>
<tr>
<td>CYAN</td>
<td>------</td>
</tr>
<tr>
<td>MAGENTA</td>
<td>------</td>
</tr>
</tbody>
</table>
**Function SCREEN**

This function selects the background color.

**INPUT:** Upon entry:

- Color value for background.

**OUTPUT:** NONE

**FORMAT:** Call SCREEN(color-value)

**Description:** While the foreground color makes up the image, the background color surrounds the character. The sixteen available colors are:

- **BLACK**
  <--------- 0
- **BLUE**
  <--------- 1
- **GREEN**
  <--------- 2
- **CYAN**
  <--------- 3
- **RED**
  <--------- 4
- **MAGENTA**
  <--------- 5
- **BROWN**
  <--------- 6
- **WHITE**
  <--------- 7
- **GRAY**
  <--------- 8
- **LIGHT BLUE**
  <--------- 9
- **LIGHT GREEN**
  <--------- 10
- **LIGHT CYAN**
  <--------- 11
- **LIGHT RED**
  <--------- 12
- **LIGHT MAGENTA**
  <--------- 13
- **YELLOW**
  <--------- 14
- **HIGH INTENSITY WHITE**
  <--------- 15
Function DRAWLINE.

This routine draws a line joining coordinates \((x_1, y_1)\) and \((x_2, y_2)\) in the current pen-color.

**INPUT:** Upon entry:

- \(x_1\) = contains x-coordinate of starting point.
- \(y_1\) = contains y-coordinate of starting point.
- \(x_2\) = contains x-coordinate of ending point.
- \(y_2\) = contains y-coordinate of ending point.

**Description:** This routine was developed using Bresenham's algorithm. Any diagonal line can be represented by the formula:

\[
y = mx + b
\]

where \(m\) is the slope of the line, and \(b\) is a constant which determines the \(Y\)-intercept. Now, when the points are drawn on the line, it is a simple procedure to change the \(X\) coordinate; we simply add one to it; so \(X\) goes from the starting \(X_1\)-coordinate up to the end point at \(X_2\)-coordinate. But it is not so simple to determine the corresponding \(Y\) coordinates. Bresenham's algorithm involves keeping track of a number, called an error term, which is related to the difference between where the pixel should go, if it could be drawn right on the line, and where it must go, since it
can only occupy integer pixel locations. Each time that we move over one unit in the X direction, we add a certain constant to this error term. If the resulting new value for the error term is big enough, we increment Y (so the pixel at the next higher value of X is plotted one point higher on the Y-axis than before), and also subtract a given constant from the error term. See figure 10.
ERRORTERM = 0

X AND Y = STARTING POINT OF LINE

SLOPE <= 1

PLOT DOT AT X, Y

INCREMENT X

ADD DELTA_Y TO ERRORTERM

IS ERRORTERM <= HALF X ?

NO

SUBTRACT DELTA_X FROM ERRORTERM

INCREMENT Y

DO YOU HAVE ALL DELTA_X POINTS YET?

NO

YES

EXIT

FIGURE 10.
ERRORTERM = 0

X AND Y = STARTING POINT OF LINE

SLOPE > 1

PLOT DOT AT X, Y

INCREMENT Y

ADD DELTA_X TO ERRORTERM

IS ERRORTERM <= HALF Y ?

NO

SUBTRACT DELTA_Y FROM ERRORTERM

INCREMENT X

DONE ALL DELTA_Y POINTS YET ?

NO

YES

EXIT

FIGURE 10b.
Function DRAWREL.

This function is similar to the DRAWLINE routine, the only difference is that the coordinates \((x2,y2)\) are relative displacements with respect to \((x1,y1)\). This means that the final \((x2,y2)\) coordinates are equal to \((x1+x2,y1+y2)\).

INPUT : Upon entry:

\(x2\) = relative displacement with respect to \(x1\).

\(y2\) = relative displacement with respect to \(y1\).

OUTPUT : Just to the Screen.

FORMAT : Call DRAWREL\((x2,y2)\)

Description: This function checks the coordinates, or position of the cursor and adds the value of \((x2,y2)\) to get the end point of the line to be drawn. The origen will be represented by the original \((x1,y1)\) coordinates.
**Function** POLYABS.

This function draws a polygon of n sides.

**INPUT:** Upon entry:

- **n:** number of sides of the polygon.
- **x:** an array with all the x1....xn vertices of the polygon.
- **y:** an array with all the y1....yn vertices of the polygon.

**Description:** Our basic primitive surface is a polygon (a many sided figure). A polygon is represented by a number of line segments connected end to end to form a closed figure.
Function POLYREL.

This routine draws a relative polygon into the screen.

INPUT: Upon entry:
X: array with relative displacements on the x-axis.
Y: array with relative displacements on the y-axis.
n: number of sides of the polygon.

Description: This routine is similar to the POLYABS routine.
The only difference is the use of relative displacements instead of absolute displacements.
**Function FILL.**

This routine fills an area on the graphics screen with a specified color. It starts "painting" at a "seed" position, filling a region bounded by a "boundary" color.

**INPUT:** Upon entry:
- x-coordinate of seed point.
- y-coordinate of seed point.
- paint color.
- boundary color.

**OUTPUT:** Just to the screen.

**FORMAT:** Call FILL(x,y,boundary-color,paint-color).

**Description:** The region must be completely surrounded by a boundary drawn in the boundary color. Any paint color can obstruct the filling process, acting just like a boundary. This algorithm checks for cases where there is just one pixel in a line, and also where there exist 180 degrees in a figure.
Function WAITS.

This routine simulates a wait state in the graphics mode so the print screen function (Shift + screen) can be used.

INPUT:  NONE
OUTPUT: NONE
FORMAT: Call WAITS()

Description: This routine is really a DOS function that reads a character from the input buffer. The routine waits for a character to be input from the keyboard. It is used instead of the input functions of PL/I.
**Function FILLXOR.**

This function is similar to the FILL function. It only differs from that previous function in that each pixel in the figure is colored with a color obtained by "exclusive oring" its original color with a specified color.

**INPUT:** Upon entry:
- x-coordinate of seed point.
- y-coordinate of seed point.
- boundary color.
- color mask for xor.

**OUTPUT:** Just to the screen.

**FORMAT:** Call FILLXOR(x,y,color-boundary,color-mask).

**Description:** This routine follows the same rules as its close relative the FILL function.
**Function MOVEABS**

This function is used to move the actual pen position using absolute coordinates.

**INPUT:** Upon entry:
- \( x_1 \): to specify the absolute \( x \)-coordinate.
- \( y_1 \): to specify the absolute \( y \)-coordinate.

**OUTPUT:** NONE.

**FORMAT:** Call MOVEABS(\( x_1, y_1 \)).

**Description:** This function just changes the value of the \((x,y)\) pen-position with the new values. These \((x,y)\) coordinates are represented in two contiguous memory locations.
**Function MOVEREL.**

This function changes the actual value of the pen-position. It differs from the MOVEABS function in that the displacement values are relative.

**INPUT:** Upon entry:

- \( X_1 = \) X-axis relative displacement.
- \( Y_1 = \) Y-axis relative displacement.

**OUTPUT:** NONE

**FORMAT:** Call MOVEREL\((X_1,Y_1)\).

**Description:**

- New-X-coordinate = Old-X-coordinate + \( X_1 \).
- New-Y-coordinate = Old-Y-coordinate + \( Y_1 \).

The memory locations representing the pen position are updated with the new values.
Function \textbf{SETBOX.}

This routine fills a rectangular box in the color graphics screen with a given color.

INPUT: Upon entry:
\begin{itemize}
  \item x-coordinate of upper left corner is in \textit{x1}.
  \item y-coordinate of upper left corner is in \textit{y1}.
  \item x-coordinate of lower right corner is in \textit{x2}.
  \item y-coordinate of lower right corner is in \textit{y2}.
  \item color of the rectangle.
\end{itemize}

OUTPUT: Just to the screen.

FORMAT: Call \textbf{SETBOX}(\textit{x1, y1, x2, y2, color}).

Description: The routine starts filling the line at coordinates \((x1, y1)\) to \((x2, y1)\), then it jumps to the next line of coordinates \((x1, y1+1)\) and so on until the last line from \((x1, y2)\) to \((x2, y2)\) is filled with the chosen color.
REFERENCES:


42