Pixel-type Display

The OMNI 300 Series pixel-type display provides the ability to display 1 to 4 lines of ASCII and international characters, or graphics, such as a company logo. Your application can access a number of pre-defined characters and graphics which are stored in the terminal's font ROM, or you can create your own display characters and graphics.

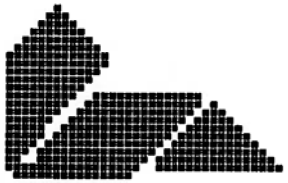


Figure 8-3. Sample Pixel-based Graphic

☞ *Data entry response time can be improved by taking full advantage of a multi-line display (if available). Use one line for prompts and another for data entry. Use data entry routines that support keyboard buffering and terminate display delays as soon as data entry begins.*

Font Basics

VeriFone defines a font as a logical set of characters, all of one size and language. Each font is stored on one or more *font* pages, up to 128 characters per page in the font ROM; up to 16 font pages may be stored on each 32KB font ROM chip.

Each font ROM chip may also include one or more pre-defined graphics or logos. Graphics stored in the font ROM are handled in the same fashion as text characters. To display the graphic image, write the associated character codes to the display, as you would when writing a text character.

Display Data

Unlike communication devices, a write to display occurs immediately. All data written to the display—except graphics written by the `putpixelcol()` function—is stored in an internal display buffer capable of storing a full screen of

data, in any font. *Display activity does not impact data communications.*

ASCII Control Codes

The display driver uses ASCII control codes to control cursor movement and clear the display window. These codes are intercepted by the firmware and do not go to the display or display buffer. Segment-type displays cannot show these control codes.

These control codes are valid for ASCII fonts and other fonts supporting them.

ASCII Control Code	Behavior
0x08 (BS)	Moves cursor back one position on current line in current window.
0x0A (LF)	Moves cursor to beginning of next line in current window, vertically scrolling as necessary.
0x0C (FF)	Clears the current window.
0x0D (CR)	Moves cursor to column one of current line of current window.

The raw mode only supports the above control codes when the page is set to one of the ASCII pages.

Character Spacing

All characters (including commas and periods) use a single character space, or more in the case of a double-wide font. This differs from segment-type displays, where the commas and periods usually occupy the previous character's display area.

Using Display Windows

Applications can create multiple display windows. The firmware allows the physical display window to be handled as a soft window, with the size of a window varying from a single character to a maximum size matching the physical display. Once an application defines a window, all subsequent operations are relative to it. All data sent to the display is assumed to be in the current font. Any character which the display device is unable to write is shown as an underscore character ("_"), ASCII 0x5F.

Creating Custom Graphics and Characters

You can create unique graphics and characters using the VeriFone PC program, *FontDesigner*. Graphics and characters created with FontDesigner may be stored as a data file and downloaded with the application, or you may store them as an include file compiled into your application. To write custom graphics or characters to the display, use the `putpixelcol()` library function.

The firmware does not store your graphic images in the internal display buffer. Thus, the application needs to provide a mechanism to re-display an image if it is cleared. In situations where a graphic image is cleared because an attempt to enter system mode was aborted due to an invalid password entry, you could use the exception trap, via the `arm_guard()` function, to repaint the display. Access this trap using the #define provided in the header file <trap.h> (`BAD_PASSWORD_TRAP`), or by calling the `trap_of()` function.

Fonts and Grids

Characters in a font are created and stored in the font ROM in a specific size, specified in pixel width and height. For example, an 8 x 16 character is 8 pixels wide by 16 pixels high, while a 16 x 16 character is 16 pixels wide by 16 pixels high. The larger the character size, the fewer characters will fit in a full screen.

Each character size has an associated grid, which determines the maximum number of rows and columns the display will have. The following table shows the grid settings available for OMNI 39X terminals (other pixel-type terminals may use different settings).

The names in the *Grid Setting* column are defined in the TXO header file `<dsp_90.h>` and correspond to the values in the *Grid Value* column.

OMNI 39X Terminal Grids and Characters

Grid Setting	Grid Value	Character Size (Wid x Ht)	Display Size (Row x Col)
`GRID_2x18`	0	8x16	2x18
`GRID_3x18`	1	8x10	3x18
`GRID_4x25`	2	6x8	4x25
`GRID_2x18`	0	16x16 (double-wide)	2x9

The pixel-display can simultaneously display different fonts of the same grid size, such as Chinese characters from any of the three Chinese font pages, or ASCII and Arabic characters of the same grid size.

☞ *To display characters from different fonts of the same grid size, use the* `setfont()` *function. This function does not clear the display or the internal display buffer, nor does it change the cursor position or the window setting.*

Mixing fonts of different grid size is not possible, since changing the grid clears the display. It is also unacceptable to mix normal and double-width character fonts of the same grid *in the same window*, such as the use of one window to display both ASCII and Chinese characters.

☞ *Use the* `resetdisplay()` *function to select a new font and grid size. This function will clear the display and the internal display buffer.*

Font Definition Files

Font Definition Files are simple files that provide the following information to the operating system:

- Font Definition File version number—lets you track file revisions
- Character code type—identifies the font code size (i.e., the number of bits used to represent a character in the specific font), and also flags the presence of special character codes; options include:

  0 = 7-bit codes (subset of the 8-bit code size).

  1 = 8-bit codes; requires translation table.

  2 = 16-bit codes; requires translation table.

  3 = combination of 8- and 16-bit codes; requires translation table.

  4 = substitution character codes, such as the 12 ISO substitution characters used to support European variants of the ASCII character sets; requires translation table containing the font ROM address for each of the substitution characters.
- Font ROM pages on which the desired font appears—each font ROM may have up to 16 font pages numbered 0 – 15
- Page and offset of the character to be displayed when an unsupported character code is called (default is the ASCII underscore ("_") character).
- Page and offset for the space character; the space character is required to clear the screen
- Page and offset codes for the backspace, form feed, line feed and carriage return characters
- Any translation tables created to support non-ASCII character fonts, such as Chinese ETEN or European variants; translation tables are used by the operating system to translate 8-bit and 16-bit character codes into font ROM addresses

Basically, Font Definition Files tell the operating system *how* the application will interact with the font ROM. The application may access the font ROM in one of two ways: It can directly access the font ROM using a *raw mode* character selection process (application provides the font ROM page number and character offset for each desired character). Or, the application can write character codes, taken from a translation table, relying on the operating system to translate the character codes into font ROM addresses.

❖ *VeriFone does not currently provide translation tables for any of the supported fonts.*

Default Font Definition File

The operating system firmware contains the default Font Definition File, ASCIIS0.FDF, which supports use of ASCII codes to write a 2 line by 18 character display of ASCII characters. This default setting will remain in effect until changed in the CONFIG.SYS file via the *FONT and *GRID entries or by the `setfont()` or `resetdisplay()` functions.

To display ASCII characters in another grid, such as 3 or 4 lines, or to select any non-ASCII character font, download the appropriate Font Definition File with your application. If you do not download a Font Defintion File, you can only utilize a 2-line ASCII character display.

The sample code disk accompanying this manual includes the following Font Definition Files for your use:

- ASCIIS0.FDF — Supports 2 line by 18 ASCII characters
- ASCIIS1.FDF — Supports 3 line by 18 ASCII characters
- ASCIIS2.FDF — Supports 4 line by 25 ASCII characters
- VFIRAW0.FDF — Supports 8x16 and 16x16 character sizes; 2-line display of 9-18 characters
- VFIRAW1.FDF — Supports 8x10 character size; 3-line display of 18 characters
- VFIRAW2.FDF — Supports 6x8 character size; 4-line display of 25 characters

The first three Font Definition Files display ASCII characters of the corresponding grid ID. The last three Font Definition Files display both ASCII and non-ASCII characters of the corresponding grid ID.

❖ *If you are using mixed languages (e.g., English and Chinese), use a "raw mode" Font Definition File: VFIRAW0.FDF or VFIRAW1.FDF or VFIRAW2.FDF.*

Raw Mode Character Selection

Raw mode refers to a method of character selection whereby the application directly addresses the font ROM. No translation tables are used, and no code translation is required by the operating system.

To use the raw mode, the application selects a font page within the font ROM by writing the font page number (0–15) plus 128 (0x80). Thus, you would write 0x80 to select the first page (0 + 0x80), or 0x84 to select the fifth ROM page (4 + 0x80).

Each font page may have up to 128 characters (0–127). To select a character on the font page, write the character's offset (also called a character font code). Writing a 0 displays the first character on the font page; writing 4 displays the fifth character. See the example file *RAW.C*, page 8-3939.

ASCII fonts are stored in the font ROM according to the ASCII standard; that is to say a character's hexadecimal value and font code are identical. For example, the character "A" has a font code value of 0x41. This means you can write ASCII font characters to the display using normal text strings instead of hexadecimal codes. The compiler will convert the text string to its hexadecimal code equivalent. For example, to display the character "H", which has a font code value of 0x48, your application has the option of writing either 0x48, or the text character "H" (which the C compiler translates to its hexadecimal equivalent, 0x48).

Non-ASCII fonts can either use raw mode to select characters by page and character offset, or create a translation table to take advantage of an existing standard.

ASCII fonts have displayable symbols for all 128 characters (except the ASCII Control Codes).

Fonts that follow an existing standard, but do not support the complete standard, normally use the ASCII underscore ("_") character to display unsupported characters. However, the Font Definition File may specify an alternative character.

Translation Tables

Translation tables may be used to support multiple industry standards for non-ASCII character fonts. A translation table provides a mapping of the codes in a standard character set, such as 16-bit Chinese ETEN codes, to the font ROM addresses of these characters.

Translation tables are downloaded as part of the font definition file for each font requiring such a table, and stored in the terminal's RAM file system.

❖ *VeriFone does not currently provide translation tables.*

Character Font Codes

Font Definition Files contain a *character code type* parameter that defines font code size; this is the number of bits used to represent a character in a specific font. Font code size options are: 7-bit codes (subset of 8-bit), 8-bit, 16-bit, and a combination of 8 and 16 bit codes.

❖ *8-bit, 16-bit and combinations of 8- and 16-bit codes require translation tables listing these codes and their corresponding font ROM addresses. VeriFone does not currently provide any translation tables.*

Fonts using 7-bit codes write values in the range of 0-127. Any value greater than 127 (high order bit set) is treated as a 7-bit value. The ASCII font is an example of this convention.

Fonts with an 8-bit code size use code values in the range of 0-255. ECMA-94 Latin is an example of a font that could be defined as an 8-bit code size. Fonts with a 16-bit code size use 2 bytes to represent each character. The

Chinese ETEN font is an example of a font that could use a 16-bit code size.

The 8- and 16-bit code size combination may be used to combine two different font sets into a single font. An example would be where any byte under 128 translates to ASCII, and any byte over 128 signals the first byte of a 16-bit code. The Japanese SHIFT-JIS standard is one example of a font that can take advantage of this usage.

Selecting a New Font

The application specifies a font for the `setfont()` or `resetdisplay()` function by specifying the Font Definition File's full file name (for example: "ASCIIS0.FDF"). The function searches the file system for the specified file name, and if it is found, changes the current font to the specified font according to the parameters in the Font Definition File.resetdisplay()

The default font is that font current at terminal power up.

The application can optionally select the default font by specifying the keyword "DEFAULT", rather than using the actual file name corresponding to the font.

Double-wide Characters

Some character sets (Chinese ETEN, for example) require two display cells to display a single character. The pixel-type display supports double-wide characters only in the 2-by-18 grid setting (grid 0). These characters can scroll and wrap like single-wide characters, however the following restrictions must be observed when using double-wide characters:

- The window that is to contain double-wide characters must have an even-sized width.
- A window can contain either double-wide characters or single-wide characters, but not both simultaneously.

- A window cannot split or overlap an existing double-wide character. A new window can cover a double-wide character completely or can be adjacent to a complete character, but cannot write over half of a double-wide character already on the screen.
- The cursor cannot be positioned (via `gotoxy()`) to the middle (second half) of an existing double-wide character.

The firmware does not prevent applications from violating any of these restrictions, but the results of doing so are unpredictable.

Pixel Display Functions

This section describes the main pixel-type display functions. It also includes library functions either developed specifically for pixel-type displays, or those functions that take pixel display specific parameters (which must be watched when porting code from a segment-type display application).

❖ *In the following functions, the logical device name* `STDOUT` *may be used in place of the* `hDSP` *parameter.*

open()

Clears the display and places the cursor in the home position.

```
#include <io.h>

hDSP = open("/dev/stdout", 0);
int hDSP;
```

On pixel-type displays, the home position is defined as the top left corner of the display. Opening the display resets the font, grid, contrast, and scrolling mode to their default settings. There are no error conditions. The display does not need to be explicitly opened.

read()

Transfers the contents of the display buffer for the current display window to the application buffer.

```
#include <io.h>

bytes_read = read(hDSP, buffer, size);
int  bytes_read;
int  hDSP;
int  size;
char *buffer;
```

This operation is provided to support program development and remote debugging.

If the font code size for the current font is 16 bits per character, two bytes will be returned for every character in the display buffer. The `bytes_read` value returned from this call will be the number of bytes transferred from the display buffer, not the number of characters. When the code size is 16 bits per character, care must be taken by the application to provide an even numbered size for the buffer to prevent partial character codes from being returned.

During raw mode operations, a read of the display will return a page byte for every character written to the display, regardless of whether the page change was written. That is, 2 bytes will be returned for every character on the display: a page change byte and a page offset byte.

write()

Transfers a specified number of bytes from the application buffer into the internal display buffer and shows the data in the current window on the display.

If the font code size is 16 bits per character, every 2-byte pair written to the display will result in one character being displayed. Partial characters (e.g., a write of a single byte) will not be displayed. The partial character will be buffered until the next write provides the second byte of the code, at which time it will be displayed.

```
#include <io.h>

bytes_written = write(hDSP, buffer, size);
int  bytes_written;
int  hDSP;
int  size;
char *buffer;
```

close()

Closes the display device.

```
#include <io.h>

status = close(hDsp);
int  status;
int  hDSP;
```

The value returned in status will be zero for a successful close and -1 for an error. The error type can be determined by reading `errno`.

window() *(parameters restricted on pixel display)*

Defines a window within the physical display.

```
#include <io.h>

void window(x1, y1, x2, y2);
int x1;
int y1;
int x2;
int y2;
```

The character coordinates x1 and y1 determine the upper left corner of the window and x2 and y2 are the character coordinates for the lower right corner.

If any of the specified coordinates are out of the range of the physical dimensions of the display, minimum/maximum values will be used. The cursor is placed in (1,1) of this window.

- *Note that the maximum sizes for a window are dependent on the current grid setting.*
- *When using double-wide characters, special restrictions apply to this function: The window width must be even, the window must contain only double-wide characters and the window cannot split any double-wide characters already on the display. These restrictions are not enforced by the firmware and may result in undefined behavior if violated.*

gotoxy() *(parameters restricted on pixel display)*

Moves the cursor to a position specified by a column value x and a row value y within the window.

```
#include <txostd.h>

void gotoxy(x, y);
int  x;
int  y;
```

If x or y exceeds the dimensions of the current window then minimum/maximum positions will be used.

This function has the restriction that the cursor cannot be repositioned in the middle of an existing double-wide character. This restriction is not enforced by the firmware and may result in undefined behavior if violated.

setfont() *(pixel display only)*

Changes current font.

```
#include <dis_90.h>

status = setfont(font);
int  status;
char *font;
```

This function changes the font to that specified by `font` if it is valid for the current grid setting and available in the font ROM. If the font is invalid or not supported by the firmware, the status returned is -1. Status is set to 0 on a successful font change.

The `font` parameter is either the Font Definition File name for the desired font, or the keyword "DEFAULT". If it is a file name and the file is not found in the file system, it is not a valid font.

The cursor position and current window setting will not be affected by this function. The display and the internal display buffer will not be affected. The code size will change to the code size required by the new font. See *Character Font Codes* for information on code size.

getfont() *(pixel display only)*

Returns the name of the Font Definition File for the current font setting.

```
#include <dsp_90.h>

int getfont(fontname);
char *font;
```

The size of `fontname` must be at least 13 bytes to accommodate the full name. The return value is always 0. See *Font Definition Files*.

resetdisplay() *(pixel display only)*

Sets grid and font.

```
#include <dsp_90.h>

status = resetdisplay(font,grid_id);
int  status;
char *font;
int  grid_id;
```

Sets the grid and font to the specified values if they are both valid and returns 0 on success; otherwise, the grid and font settings remain unchanged and -1 is returned.

If the specified values are valid, the display and internal display buffer are cleared, the window is reset to the default window, and the cursor is reset to the home position. The scrolling mode and current contrast setting are unchanged by this function.

getgrid() *(pixel display only)*

Returns the current grid setting.

```
#include <dsp_90.h>

grid_id = getgrid();
int  grid_id;
```

getfontinfo() *(pixel display only)*

Returns information about the font page at the index location in the font ROM.

The `index` parameter represents the sequential position of the font page within ROM, where the first font page is 0. The font page information is copied into buffer, which must be at least 24 bytes long.

```
#include <dsp_90.h>

status = getfontinfo(index,buffer);
int  status;
int  index;
char *buffer;
```

The return value is 0 on success or -1 on error; `errno` can be read to determine the type of error. If index exceeds the number of font pages in the font ROM, an error is returned and `errno` is set to `ENOENT`.

This function returns information about the font ROM, not information about the fonts (or Font Definition Files) available via the `getfont()` and `getgrid()` functions.

The format for the information returned in buffer is as follows:

`char`	`rom_name[9]`	Null-terminated name of font ROM
`char`	`font_page[9]`	Null-terminated name of font page
`char`	`x`	Width in pixels of one char
`char`	`y`	Height in pixels of one char
`char`	`grid_id`	Grid setting of the font
`char`	`reserved[3]`	Unused

The application can make successive calls to this function to get information on all the font pages in the font ROM.

setcontrast() *(pixel display only)*

Sets the screen contrast level.

```
#include <dsp_90.h>

void setcontrast(level);
int  level;
```

Set `level` to a value in the range 0 to 15, with 0 being the most contrasted and 15 the least.

☞ *High contrast values (lower contrast), may make the display difficult to read.*

getcontrast() *(pixel display only)*

Returns the current screen contrast level.

```
#include <dsp_90.h>

level = getcontrast();
int  level;
```

Acceptable `level` values range from 0 to 15, with 0 being the most contrasted and 15 the least.

putpixelcol() *(pixel display only)*

Writes a given graphic to the display, starting at the current cursor position.

```
void putpixelcol(buffer,len);
char *buffer;
int  len;
```

Each byte in the buffer represents the on/off state of a column of 8 pixels, where 1 equals pixel on (darkened) and 0 does nothing to the display. This function ORs the pixel state to the current display state (if the pixel is on it remains on), so the application should first clear the display area before calling this function.

The buffer is written by columns from the top of the character cell to the bottom, moving from left to right. Once an entire character cell in the given font is filled, the cursor position is incremented.

Different grid settings take different numbers of bytes to fill one character cell. For example, the 2x18 grid setting requires 16 bytes to fill one character, grid 3x18 requires 10 bytes and grid 4x25 requires 6 bytes.

This function works on the current window and not the entire screen. Data for this function can be created by the PC program *FontDesigner* and linked into the application (or downloaded as a separate data file). Data created by FontDesigner is written for a specific grid setting. The application must set the grid to that setting before calling the `putpixelcol()` function in order for the image to appear properly.

❖ *This function does not support data in 16x16 character size (double-wide, grid 0).*

Related Pixel-Display Functions

The following display functions may also be used with pixel-type displays. A complete description of each function is contained in *Chapter 11, Library Functions.*

- `clrscr()`
- `clreol()`
- `delline()`
- `insline()`
- `wherecur()`
- `wherewin()`
- `wherewincur()`
- `setscrollmode()`
- `getscrollmode()`

Tips for Common Uses of the Pixel Display

This section provides some tips and examples of various ways you may want to utilize the pixel display.

- Displaying two lines
- Displaying three lines
- Displaying four lines
- Displaying characters from various font pages
- Using windows
- Creating a custom logo

Displaying Two Lines

All pixel-display terminals have a default display of 2 lines with 18 characters each. These ASCII characters may be alphanumeric, and both upper and lower case.

The following example displays `Hello, world.` in the default 2-line mode.

Steps for 2 x 18 display example:

1. Compile the program: `txo hello2.c`
2. Download the program: `dl hello2.out`

```
/*  HELLO2.C - Hello, world */
#include <io.h>
main()
{
/* Use default 2-line display mode to display */
/* "Hello, world.".                           */
     write(STDOUT,"\fHello, world.",14);
}
```

Displaying Three Lines

Displaying ASCII characters on the pixel display with 3 lines of 18 characters per line requires two changes from the 2-line display. First, the Font Definition File ASCIIS1.FDF must be downloaded and, second, the display must be reset to use ASCIIS1.FDF and its accompanying display grid.

The following example displays `Hello, world.` in the default 2-line mode, resets the display to use the Font Definition File ASCIIS1.FDF, sets the grid for the 3-line display and displays `Hello, world.` three times.

Steps for 3 x 18 display example:

1. **Compile the program:** `txo hello3.c`
2. **Download the program:** `dl hello3.out -iasciis1.fdf`

```
/*  HELLO3.C -- Hello, world */
/*  Display 3 x 18 mode      */
#include <io.h>
#include <dsp_90.h>
#include <txosvc.h>
main()
{
  int status;
  /* Use default 2-line display mode to display */
  /* "Hello, world.".                           */
     write(STDOUT,"\fHello, world.",14);
  SVC_WAIT(500);
  /* Reset to use 3-line display mode.          */
  status = resetdisplay("ASCIIS1.FDF",GRID_3x18);
  /* If error on reset, display error message.  */
  /* Else, display "Hello, world." 3 times.     */
  if(status == -1)
    write(STDOUT,"No reset",8);
  else {
    write(STDOUT,"Hello, world.",13);
    write(STDOUT,"Hello, world.",13);
    write(STDOUT,"Hello, world.",13);
    }
}
```

Displaying Four Lines

Displaying ASCII characters on the pixel display with 4 lines of 25 characters per line requires two changes from the 2-line display. First, the Font Definition File ASCIIS2.FDF must be downloaded and, second, the display must be reset to use ASCIIS2.FDF and its accompanying display grid.

The following example displays `Hello, world.` in the default 2-line mode, resets the display to use the Font Definition File ASCIIS2.FDF, sets the grid for the 4-line display and displays `Hello, world.` four times.

Steps for 4 x 25 display example:

1. **Compile the program:** `txo hello4.c`
2. **Download the program:** `dl hello4.out -iasciis2.fdf`

```
/* HELLO4.C -- Hello, world */
/* Display 4 x 25 mode      */
#include <io.h>
#include <dsp_90.h>
#include <txosvx.h>
main()
{
  int status;
  /* Use default 2-line display mode to display */
  /* "Hello, world.".                           */
     write(STDOUT,"\fHello, world.",14);
  SVC_WAIT(500);
  /* Reset to use 4-line display mode.          */
  status = resetdisplay("ASCIIS2.FDF",GRID_4x25);
  /* If error on reset, display error message.  */
  /* Else, display "Hello, world." 4 times.     */
  if(status == -1)
    write(STDOUT,"No reset",8);
  else {
    write(STDOUT,"Hello, world.",13);
    write(STDOUT,"Hello, world.",13);
    write(STDOUT,"Hello, world.",13);
    write(STDOUT,"Hello, world.",13);
  }
}
```

Displaying Characters from Various Font Pages

There are three different methods for displaying characters from various font pages:

1. To display a particular font upon power up, change the CONFIG.SYS file.
2. To display a font with the same grid size, use `setfont()`.
3. You can also display any character in the font ROM.

Display Font Upon Power Up

To display a particular font upon power up, change the default configuration via CONFIG.SYS entries `*FONT` and `*GRID` and download the appropriate Font Definition File.

The following example illustrates how to change a 2-line display program to a 4-line display program via the CONFIG.SYS file. The file `HELLO2.C` from "Displaying Two Lines" is used with only a change in the download command line.

Four-line display example:

1. Compile the program: `txohello2.c`
2. Download the program: `dl hello2.out -iasciis2.fdf *FONT=ASCIIS2.FDF *GRID=2`

Display Font with Same Grid Size

To display a particular font with the same grid size as the current font, use `setfont()` to change to the corresponding Font Definition File. Then select a font page within the font ROM by writing the font page number (0-15) plus 128 (0x80).

```
#include <dis_90.h>

status = setfont(font);
int  status;
char *font;
```

Display Any Character in Font ROM

To display any character in the font ROM:

1. Use `resetdisplay()` to select the appropriate grid size and Font Definition File.
2. Select a font page within the font ROM by writing the font page number (0-15) plus 128 (0x80). You may write from different font pages provided they share the same grid ID. Fonts with a character size of 8x16 or 16x16 have a grid ID of 0. Fonts with a character size of 8x10 have a grid ID of 1. Fonts with a character size of 6x8 have a grid ID of 2.
3. To display a character, write its offset. For example, writing a 0 displays the first character on the font page.
4. Download the program with the appropriate Font Defintion File.

The following example displays the first three characters of the fifth font ROM page. The grid size of the fifth font ROM page varies; for a real application, you would know the grid size (`grid_id`).

Steps for displaying first three characters of the fifth font ROM page:

1. Compile the program: `txo raw.c`
2. Download the program: `dl raw.out -ivfiraw0.fdf -ivfiraw1.fdf -ivfiraw2.fdf`

```
/* RAW.C -- Use raw mode of displaying characters on a  */
/* pixel display.                                       */
/* Display first 3 characters of fifth font ROM page.   */
/* For a real application, you would know the grid      */
/* setting required and reset the display accordingly.  */
/* You would not need the union/structure, getfontinfo  */
/* and case statements.                                 */

#include <io.h>
#include <dsp_90.h>
#include <txosvc.h>
```

```
main()
{
  int status;
  char dbuffer[3];

  /* Define structure/union to hold font information.    */
  union {
    struct s_font{
      char rom_name[9];  /* font ROM name                */
      char font_page[9]; /* font page name               */
      char x;            /* width in pixels of one char  */
      char y;            /* height in pixels of one char */
      char grid_id;      /* grid setting required by     */
                         /* this font                    */
      char reserved[3];  /* unused                       */
    } font;
    char font_line[24];
  } fontline;

  /* Get font information on the fifth ROM page.         */
  status = getfontinfo(4,fontline.font_line);

  if (status == 0) {
    /* Reset display mode for fifth font ROM page.       */
    switch ((int)(fontline.font.grid_id)) {
      case 0: status = resetdisplay("VFIRAW0.FDF",GRID_2x18); break;
      case 1: status = resetdisplay("VFIRAW1.FDF",GRID_3x18); break;
      case 2: status = resetdisplay("VFIRAW2.FDF",GRID_4x25); break;
      default: write(STDOUT, "UNKNOWN GRID ID", 16);
    }

    /* Change to fifth font ROM page.                    */
    /* Display first 2 characters on that font ROM page. */
    dbuffer[0] = (char)(0x84);
    dbuffer[1] = 0;
    dbuffer[2] = 1;
    write(STDOUT, dbuffer, 3);
```

```
/* Display third character on fifth font ROM page.      */
    dbuffer[0] = 2;
    write(STDOUT, dbuffer, 1);
  }
  else
    write(STDOUT, "Could not resetdisplay", 22);

}
```

Using Windows

A "window" is a defined area of the display. To use a window, you must first define it with the `window()` function. Subsequent writing to the display writes in that defined window. A window need not be an entire line.

The following example uses a 4-line display, but only scrolls the fourth line (the "window"); the top three lines are stationary.

Steps for using windows example:

1. Compile the program: `txo window4.c`
2. Download the program: `dl window4.out -iasciis2.fdf`

```
/* WINDOW4.C -- Display with a 4-line display but only   */
/* scroll the fourth line, leaving the the top three     */
/* lines stationary.                                     */

#include <io.h>
#include <dsp_90.h>
#include <txostd.h>
#include <txosvc.h>

int x1,y1,x2,y2;
main () {
```

```
  int status;

  /* Reset to use 4-line display mode.                          */
  status = resetdisplay("ASCIIS2.FDF",GRID_4x25);

  /* If error on reset, display error message.                  */
  /* Else, put out messages on first 3 lines of display         */
  /* then scroll last line.                                     */
  if(status == -1)
    write(STDOUT,"No reset",8);
  else {
    write(STDOUT,"*** Some              ***",25);
    write(STDOUT,"***     things stay   ***",25);
    write(STDOUT,"***         the same  ***",25);

    /* Define a window consisting of the last line of           */
    /* the 4-line display.                                      */
    window(1, 4, 25, 4);

    /* Scroll last line.                                        */
    for ( ; ; ) {
      write(STDOUT,"****** Some change ******",25);
      SVC_WAIT(1000);
      write(STDOUT,"                         ",25);
      SVC_WAIT(1000);
      write(STDOUT,"---------------",15);
      write(STDOUT,"\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\",15);
      write(STDOUT,"|||||||||||||||",15);
      write(STDOUT,"///////////////",15);
    }

  }

}
```

Creating a Custom Logo

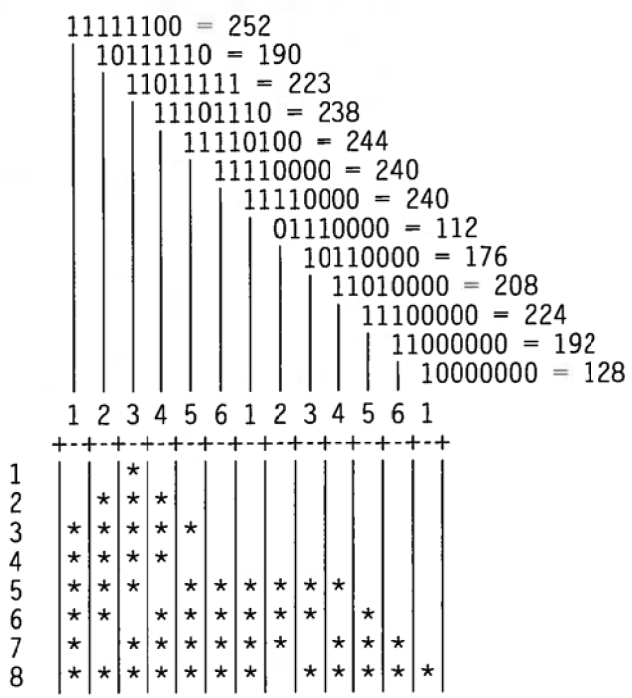
For the best results, a 4-line display is recommended for creating custom logos. The following example utilizes a 4-line display with a character size of 6 x 8. Logos for other display modes can be performed with the appropriate `resetdisplay()` settings and data files.

In this example, an array of characters is created whereby each byte corresponds to a 6-pixel vertical column on the display. Each bit corresponds to a pixel. The lowest pixel in the vertical column corresponds to the most significant bit; the highest pixel corresponds to the least significant bit.

Steps for creating a custom logo (VeriFone logo) example:

1. Draw logo on graph paper.
2. Calculate each element of the array from the darkened areas on the graph paper.
3. Create data file with the array with logo information to be included by the main program.
4. Compile the program: `txo logo1.c`
5. Download the program: `dl logo1.out -vfiraw2.fdf`

In the following example, this is how each element (darkened area) of the logo array is calculated:



This is the main logo program:

```
#include <dsp_90.h>
#include <io.h>
#include <txostd.h>
#include "vfi1.slc"
main()
{
  int status;
  clrscr();
  /* Reset to use 4-line display mode. */
  status = resetdisplay("VFIRAW2.FDF",GRID_4x25);
  if (status == -1)
    write(STDOUT,"No display reset", 16);
  else
    putpixelcol(vfi_logo1,sizeof(vfi_logo1));
}
```

This is the data file with the logo information in the array vfi_logo1:

```
/*********************************************************
 Title                : vfi1
 Format               : OMNI 390
 Character size       : 6x8
 Bytes per character  : 6
 Number of characters
   per 4-line display: 100
*********************************************************/

char vfi_logo1[] = {
/*     1        2        3        4        5        6    1st line */
(char)252,(char)190,(char)223,(char)238,(char)244,(char)240,  /* 3 #  0 */
(char)240,(char)112,(char)176,(char)208,(char)224,(char)192,  /* 3 #  1 */
(char)128,  (char)0,  (char)0,  (char)0,  (char)0,  (char)0,  /* 3 #  2 */
};
```

Beeper Device

The "beeper" is a sound-generating device that allows audible tones to be emitted from the terminal. Two types of sounds are defined: a "normal beep" (1200 Hz for 50 milliseconds), and an "error beep" (889 Hz for 100 milliseconds). In addition, the sound-generating device may be programmed to produce a specific frequency for a specified duration.

By default, terminal key presses are accompanied by a normal beep (key beeps). The application can disable this feature with the call:

```
#include <io.h>

ioctl(kbd_handle, SetCtrl|1, 0);
```

To restore, use `SetCtrl|0`. See the *Keyboard* section for more information.

Beeper Function Calls

open()

Explicitly opens the sound-generating device, returning its associated device handle.

```
#include <io.h>

hBeeper = open("/dev/stderr", 0);
int hBeeper;
```

The beeper does not need to be explicitly opened; the logical device name STDERR, defined in file <io.h> and <config.h>, may be used by other I/O functions in place of the device handle hBeeper.

ioctl()

Produces "normal" and "error" beeps.

```
#include <io.h>

result = ioctl (hBeeper, SetCtrl | type, 0);
int result, hBeeper;
unsigned int type;
```

The action performed is determined by the type parameter in combination with SetCtrl (defined in <io.h>). Valid type options are

0x00FF	Turn the beeper off immediately.
0x0000	Initiate a "normal" beep.
0x0002	Initiate a "normal" beep, wait until beep ends before returning control.
0x0004	Initiate an "error" beep.
0x0006	Initiate an "error" beep, wait until beep ends before returning control.

sound()

Generates a tone at a specified frequency (defined in <notes.h>) for a specified duration and octave. These constants are listed in Table 8-5.

```
#include <notes.h>

sound (note, duration);
int note, duration;
```

The constant NOTE_PAUSE places periods of silence between notes.

The maximum duration of a note is approximately four seconds (4000 ms) and varies with the note being played. If the specified duration exceeds the maximum, it defaults to the maximum duration.

octave \ scale	C	Db	D	Eb	E	F	Gb	G	Ab	A	Bb	B
1												
2									•	•	•	•
3	•	•	•	•	•	•	•	•	•	•	•	•
4	•	•	•	•	•	•	•	•	•	•	•	•
5	•	•	•	•	•	•	•	•	•	•	•	•
6	•	•	•	•	•	•	•	•	•	•	•	•
7	•	•	•	•	•	•	•	•	•	•	•	•
8	•	•	•	•	•	•	•	•	•	•	•	•
9	•	•	•	•	•	•	•	•	•			

Table 8-1. Notes Supported by sound() Function

Due to hardware limitations, the exact note produced may be off by as much as a quarter step, resulting in some notes being noticeably out of tune. Values in the range of NOTE_A5 through NOTE_C8 work best.

close()

Releases the handle associated with the beeper.

```
#include <io.h>

status = close (hBeeper);
int status, hBeeper;
```

Sound Function Example

```
/* BEEPER.C                          */
 /* This program demonstrates the
    use of the beeper functions.  */

#include <stdio.h>
#include <io.h>
#include <notes.h>
#include <txosvc.h>

int beep_handle,i;

main () {

   /* Open beeper. */
   beep_handle=open("/dev/stderr", 0);

   /* Output all four beep types. */
   for (i=0; i<8; i=i+2) {
      ioctl(beep_handle, SetCtrl | i, 0);
      SVC_WAIT(1000);
   }

   /* Issue 100 error beeps, then cut it off. */
   for (i=0; i<100; i++) ioctl(beep_handle, SetCtrl | 4, 0);
   ioctl(beep_handle, SetCtrl | 0x00FF, 0);
   SVC_WAIT(1000);
```

Continued

```
    /* Play beginning of
      'When the Saints Go Marching In' */
    sound(NOTE_C6, 400);   /* Oh      */
    sound(NOTE_E6, 400);   /* when    */
    sound(NOTE_F6, 400);   /* the     */
    sound(NOTE_G6, 1600);  /* saints  */
    sound(NOTE_PAUSE, 100);
    sound(NOTE_C6, 400);   /* Oh      */
    sound(NOTE_E6, 400);   /* when    */
    sound(NOTE_F6, 400);   /* the     */
    sound(NOTE_G6, 1600);  /* saints  */
    sound(NOTE_PAUSE, 100);
    sound(NOTE_C6, 400);   /* Oh      */
    sound(NOTE_E6, 400);   /* when    */
    sound(NOTE_F6, 400);   /* the     */
    sound(NOTE_G6, 800);   /* saints  */
    sound(NOTE_E6, 800);   /* go      */
    sound(NOTE_C6, 800);   /* mar-    */
    sound(NOTE_E6, 800);   /* ching   */
    sound(NOTE_D6, 1600);  /* in      */

    /* Close the beeper. */
    close(beep_handle);
}
```

Card Reader

The card reader can read data simultaneously from tracks 1 and 2, or optionally from tracks 2 and 3 (set at manufacture). This is in accordance with the VISA® Second Generation "Full Service" Manufacturer's Specification Manual.

❖ *A track 1 separator character (^) is interpreted as 00x5E.*

Card Reader Functions

open()

Enables the card reader and returns a handle used in subsequent card reader operations.

```
#include <io.h>

hCard = open ("/dev/card", 0);
int hCard;
```

The terminal ignores all card swipes until this function is called.

read()

Returns card data from tracks 1 and 2, or from tracks 2 and 3. Transfers the contents of the internal card reader buffer to the application buffer.

```
#include <io.h>

bytes_read = read (hCard, buffer, size);
int bytes_read, hCard, size;
char * buffer;
```

If the size parameter is less than the actual number of bytes in the internal buffer, all remaining bytes (unread) are cleared. The internal buffer holds only one data array resulting from a card swipe. If data exists in the internal buffer, the card reader device is temporarily disabled; swipes do not overwrite data in the buffer, they must be cleared with read().

Data returned in the buffer is in the following format:
C1 S1 D1 C2 S2 D2 C3 S3 D3

where:

C1 = 1-byte size of C1+S1+D1
S1 = 1-byte status of track 1 read operation
D1 = zero or more data bytes from track 1

C2 = 1-byte size of C2+S2+D2
S2 = 1-byte status of track 2 read operation
D2 = zero or more data bytes from track 2

C3 = 1-byte size of C3+S3+D3
S3 = 1-byte status of track 3 read operation
D3 = zero or more data bytes from track 3

The status byte (S1, S2 or S3) contains one of the following values:

0 = valid data
1 = no data
2 = missing start sentinel or insufficient data
3 = missing end data or excessive data
4 = missing BCC or BCC error
5 = parity error

The value returned in bytes_read is the size of the entire buffer with all three data structures:

0 = no card data
6 = card was read with errors
>6 = card was read with valid data

If an error occurs unrelated to card data, -1 is returned, with errno set to a specific error value.

Because the size of D1, D2 and D3 may vary, the values of C2, S2, D2, C3, S3 and D3 do not consistently fall into the same position within the application buffer.

Assuming data is read into the following array, use the following indexing method to locate the values of S2 and S3:

```
char card_buffer[120];
S2=card_buff[card_buff[0]+1];
S3=card_buff[card_buff[0]+card_buff[card_buff[0]]+1];
```

write()

While the card reader is normally thought of as a read-only device, the card reader can be "written" to as well. Data written to the card reader is stored in the same internal buffer used by read().

```
#include <io.h>

bytes_written = write (hCard, buffer, size);
int bytes_written, hCard, size;
char *buffer;
```

The internal buffer must be empty before data can be written to it (see read() for details). The written data should conform to the structure used by read().

When the arm_guard() function is used to monitor Trap 57, writing to the card reader causes an event trap.

ioctl()

The card reader supports one control function which indicates whether card data is present in its internal buffer.

```
#include <io.h>

result = ioctl (hCard, GetCtrl | 0, 0);
int result, hCard;
```

The value of result is either 1, indicating that data is present, or 0 (data is not present).

close()

Releases the handle associated with the card reader and disables it from reading subsequent card swipes.

```
#include <io.h>

status = close (hCard);
int status, hCard;
```

Card Reader Example

```
/* CARDREAD.C                                    */
/* Demonstrates use of card reader functions. */

#include <stdio.h>
#include <config.h>
#include <strings.h>
#include <memory.h>
#include <txosvc.h>

int card_handle,i,ptr;
char buffer[20];

main () {

   /* Open card reader. */
   card_handle=open("/dev/card", 0);

   /* Create card reader buffer with Tracks 1 and 2 */
   /* having valid data and Track 3 having no data. */
   buffer[0] = 7;              /* c1: c1+s1+d1 = 7 bytes */
   buffer[1] = 0;              /* s1: valid data         */
   strcpy(&buffer[2],"FIRST"); /* d1: data is "FIRST"    */
   buffer[7] = 8;              /* c2: c2+s2+d2 = 8 bytes */
   buffer[8] = 0;              /* s2: valid data         */
   strcpy(&buffer[9],"SECOND"); /* d2: data is "SECOND"  */
   buffer[15] = 2;             /* c3: c3+s3+d3 = 2 bytes */
   buffer[16] = 1;             /* s3: no data            */
```

Continued