6 Interactive Color Raster Graphics

Raster graphics generate color images stored as a collection of pixels (dots) arranged in rows and columns. This chapter introduces the Interactive Color Raster protocol and describes how to use it to write programs that display color graphics with NCSA Telnet for the Macintosh. The chapter concludes with a sample C program you can use as a template for designing programs that use the ICR protocol.

Using the ICR Protocol

Interactive Color Raster (ICR) is a protocol for displaying raster graphics on your workstation screen. The ICR protocol controls its own windows through NCSA Telnet and shares characteristics of the Tektronix graphics terminal emulation protocol. One such shared characteristic is that escape sequences control the display.

To run ICR graphics you must have:

- a 256-color Macintosh computer
- an ICR program that runs on your remote host

You can use ICR to write mainframe programs that display color images (in the full range of 256 colors) in their own windows on your Macintosh screen.

Your ICR program must include all commands needed to emulate ICR graphics. To create an ICR program, work from the protocol description contained later in this chapter ("Creating an ICR Program" on page 6–2) and from the sample program ("Sample ICR Program in C" on page 6–6).

Starting and Quitting ICR Graphics Emulation

When the protocol's command for creating a window arrives from a remote host, NCSA Telnet creates a Macintosh window for it. All human-readable text is sent to the session window, while all graphics commands are sent to the proper graphics window.

The ICR program on the remote computer can also remove the window. If it does not, you can delete a graphics window by clicking in the close box in the upper-left corner of the window's title bar. If you close an NCSA Telnet connection while some windows remain open, the windows close automatically.

Allocating Memory

Raster graphics windows require a lot of memory—one byte for each pixel in each graphics window on the screen. If insufficient memory remains to open a new window, NCSA Telnet displays an alert box and does not create the window.

If you are using MultiFinder, you can increase NCSA Telnet's allocated memory size so that you are less likely to run out of memory. For example, if you need space for two image windows and each is 256 pixels x 256 pixels, increase NCSA Telnet's allocated memory size by 128 K (256 bytes x 256 bytes, or 64K, for each window).

Copying an ICR Graphics Window

You can copy the contents of an ICR window to the Macintosh clipboard, then paste it into a program capable of pasting color images.

To copy the contents of a graphics window:

1. Click in the graphics window to bring it to the front.
2. Choose **Copy** from the **Edit** menu.

Now you can paste the graphic into another Macintosh application.

System Color Problems

Image windows use the display colors that are available on your Macintosh screen. When you close graphics windows, the system does not always restore the color environment to its original state, which causes incorrect colors in other windows. We are working to minimize the effects of NCSA Telnet and ICR graphics on your system's color table.

Pressing CONTROL-C (or using other methods to interrupt ICR commands) can make NCSA Telnet appear to lock up. (See also **Send "Are You There?"** and **Send "Interrupt Process"** under "Telnet Commands" on page 3–3.) When this occurs, either press RETURN several times or enter commands until the session window resumes activity. It may be useful to remember that each time you issue a drawing command NCSA Telnet expects an influx of often hundreds of bytes of image data to finish drawing the current line.

Creating an ICR Program

You must write a program that issues graphics commands to NCSA Telnet. NCSA Telnet follows your programs' directions to receive graphics commands, interpret them, create or destroy windows, set the color environment, and display raster graphics.

Escape Sequence

Begin all ICR graphics sequence commands with the escape sequence `ESC^` (escape, caret). This escape sequence ensures that NCSA Telnet can distinguish between regular text and ICR graphics.

ICR Commands

Each ICR command appears in the following form:

`ESC^X;` *parameters* `^` *data*

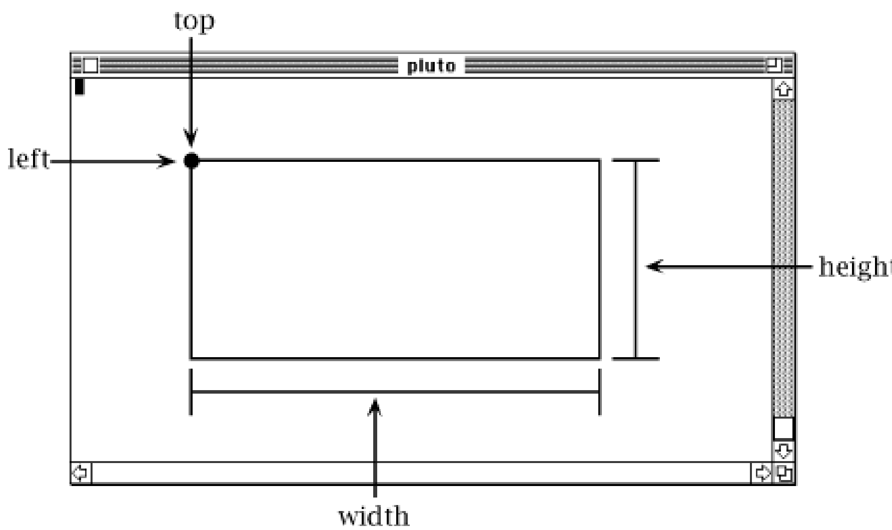
where:

- *X* is one of the command characters (`W`, `D`, `M`, `R`, `P`, or `I`) described in "ICR Command Parameters" on page 6–3.
- `^` is the caret character (ASCII 94).
- *Parameters* is one or more of the options of *X* described in "ICR Command Parameters" on page 6–3. Parameters are always printable ASCII characters and are delimited by commas. If your program omits parameters, NCSA Telnet supplies default values.
- The command is terminated with a caret (`^`).
- Each command can be followed by a data stream (*data*). If a command requires a data stream, the stream follows the command.

ICR Command Parameters

Command characters and their parameters are described below:

Command	Option Parameters	Description
W	left; top; height; width; display; windowname	Creates a window at the specified location on the screen. 0, 0 is the upper-left corner of the screen. • The *left*, *top*, *width*, and *height* integers identify specific locations on the screen:



Integer	Definition
left	the pixel value of the *x* (horizontal) location of the upper-left corner of the graphics window
top	the pixel value of the *y* (vertical) location of the upp-erleft corner of the graphics window
height	the number of pixels in the vertical height of the graphics window
width	the number of pixels in the horizontal width of the graphics window

- The *display* integer identifies the hardware screen number (for machines with multiple screens). This parameter is not applicable to Macintosh systems.
- *Windowname* is the unique name assigned to a window by the W command. The assigned windowname distinguishes between multiple windows. It is used by all other commands to specify that window.

Command	Option Parameters	Description
D	windowname	Destroys a window by physically removing it from the screen and memory. • *Windowname* is the unique name assigned to a window by the W command.
M	start; length; count; windowname	Loads into the graphics window a color map palette (of up to 256 colors) or portion of one. NCSA Telnet assumes that each palette entry is 3 bytes in the order R, G, and B.

		The default palette is a straight gray-scale ramp, where 0 = black and 255 = white.(See “Color Maps” on page 6–5.) • The *start* integer identifies the first entry to change. • The *length* integer indicates the number of entries to change. • The *count* integer indicates the total number of bytes in the data portion. Follow the count integer with the command’s data stream. • *Windowname* is the unique name assigned to a window created by the W command.
R	x; y; expand length;	Indicates the data that follow are run-length encoded. (See “Run-Length Encoding” on page 6–5.) • The *x* and *y* integers identify the point where the raster line starts and data follow for length bytes of encoded data. • The *expand* integer indicates how many times each dimension is to be expanded on your local screen. For example, an expand value of 2 makes the picture 4 times larger. • The *length* integer indicates the encoded length (in bytes) of the data. • *Windowname* is the unique name assigned to a window created by the W command.
P	x; y; expand; length; windowname	Indicates the data that follow are pixel data. • The *x* and *y* integers identify the point where the raster line starts and data follow for length bytes of pixel data. • The *expand* integer indicates the number of times each dimension is to be expanded on your local screen. For example, an expand value of 2 makes the picture 4 times larger. • The *length* integer indicates the encoded length (in bytes) of the data. • *Windowname* is the unique name assigned to a window created by the W command.
I	x; y; expand; length; windowname	Indicates the data that follow are encoded with the IMCOMP compression scheme (4:1 compression). You *must* use the M command before the picture displayed with the I command appears correctly. • The *x* integer indicates the number of pixels per line. One I call represents 4 lines of data. Since IMCOMP is a 4 x 4 square compression scheme, each line of data appears as 4 lines of pixels on the screen. • The *y* integer is required to increment by fours: 0, 4, 8, 12, 16, etc. • The *length* integer indicates the length (in bytes) of the data. Length should be the same as the number of pixels to be displayed. • *Windowname* is the unique name assigned to a window created by the W command.

Color Maps

Use the M command to manipulate the color table for your local display. The format for color map data is a series of color-map entries. Each color-map entry is three bytes: one R (red), one G (green), and one B (blue). For example, to set entries 3 through 7 of the color table, you could use the following M command:

```
ESC^M;3;4;12;wind^RGBRGBRGBRGB
```

where RGBRGBRGBRGB is the list of byte values for the new entries. The actual data transmitted over the line must still be ASCII encoded, but the data start out in this form.

NOTE: The count field is always three times the length value. In the example above, the count field is 12 and the length value is 4.

ASCII Encoding

NCSA Telnet assumes that all parameter values (except ESC) are printable ASCII. This means that the parameters require no special encoding, but data values need help. ESC is an allowable exception on most logon data streams.

Your ICR program must encode 8-bit data values into printable ASCII for transmission. When possible, the values that fall in the printable ASCII range are passed untouched, and all values outside that range are encoded as two bytes.

Use the following encoding for all characters 0–255:

Input	realchar
Transmission	specialchar followed by transchar
Encoding	specialchar = realchar div 64 + 123 transchar = realchar mod 64 + 32
Decoding:	realchar = (specialchar – 123)*64 + (transchar – 32)

These four codes encode data values in printable ASCII character for all characters 0–255, as shown below:

Special	**Range**
123	0–63
124	64–127
125	128–191
126	192–255

Because all encoded characters are preceded by a character in the 123–126 range, you can send all regular characters that are 32–122 (inclusive) without encoding.

WARNING: Because the CTSS operating system trims trailing spaces, you should avoid the values 0, 32, 128, and 192 because they code to `special space`.

NOTE: In the specifications, all data lengths and counts refer to the protocol data, not to the ASCII-encoded data. The length fields for the R, P, and M commands all reflect the data length on the originating machine before encoding.

Run-Length Encoding

Data for the run-length encoded (RLE) line are first run-length compressed and then ASCII encoded. The deciphering process first decodes ASCII to binary and then decodes the run-length binary data.

Using all 8 bits of the byte stream representing the pixels in a given RLE line, start with the control character. The lower seven bits of the byte are represented by *n*. The high bit represents whether the *n* characters that follow are reproduced exactly (high bit = 0) or whether the single character that follows is reproduced *n* times (high bit = 1).

```
Input      1  1  1  1  23  23  23  234  112  33  44  55  42  42  42  42
Tokenized  (128+4)  1  (128+3)  23  (5)  234  112  33  44  55  (128+4)  42
Alternate count, data, count, data
```

After coding in this tokenized form, you know the data length for the R command. (The length is 12 in the example above). Even though ASCII encoding occurs after this step, use the length value from this step.

```
ASCII result  125  36  123  33 125  35  123  55  123  37  126  74  112  33  44  55  125  36  42
```

Sample ICR Program in C

The sample C program shown below is included on the distribution disk. If you are running an active ICR-equipped NCSA Telnet, this program produces a test pattern on your screen. If you do not have ICR, this program produces thousands of encoded characters on your screen.

```
/*   icrtest
*   Produces a test pattern on an ICR compatible display. Demonstrates and provides
*   example code for writing ICR programs.
*
*   National Center for Supercomputing Applications
*   University of Illinois, Urbana-Champaign
*
*   by Tim Krauskopf
*   This program is in the public domain.
*/
#include <stdio.h>

int
       xdim=0,ydim=0;                          /* size of image on disk */

char
       *malloc(),
       *testimage,
       rgb[768];                               /* storage for a palette */

main(argc,argv)
       int argc;
       char *argv[];
       {
       register int i,j;
       register char *p;

       puts("Creating test pattern");

       xdim = 150;
       ydim = 100;

       if (NULL == (testimage = malloc(xdim*ydim)))
              exit(1);
```

```
/*
*  Make the test image in a strange pattern.
*/
       p = testimage;

       for (i=0; i<ydim; i++)
              for (j=0; j<xdim; j++) {
                     *p++ = 50 + (((i & 0xfffffff8) * (j & 7))>>2);
       }

       puts("Displaying test pattern with the Interactive Color Raster protocol");

       rimage(0);                         /* display remote image with [palette] */

}

/******************************************************************************/

/*  rimage
*  Remote display of the image using the ICR.
*  Just print the codes to stdout using the protocol.
*/

rimage(usepal)
       int usepal;
       {
       int i,j,newxsize;
       char *space,*thisline,*thischar;
       register unsigned char c;

/*
*  Open the window with the W command.
*/

(void)printf("\033^W;%d;%d;%d;%d;0;test window^",0,0,xdim,ydim);

/*
*  If a palette should be used, send it with the M command.
*/
       if (usepal) {
              (void)printf("\033^M;0;256;768;test window^");        /* start map */

              thischar = rgb;
              for (j=0; j<768; j++) {
                     c = *thischar++;
                     if (c > 31 && c < 123) {
                            putchar(c);
                     }
                     else {
                            putchar((c>>6)+123);
                            putchar((c & 0x3f) + 32);
                     }
              }
       }
```

```
/*
*  Send the data for the image with RLE encoding for efficiency.
*  Encode each line and send it.
*/
       space = malloc(ydim+100);
       thisline = testimage;

    for (i = 0; i < ydim; i++) {
       newxsize = rleit(thisline,space,xdim);
              thisline += xdim;                 /* increment to next line */

        (void)printf("\033^R;0;%d;%d;%d;test window^",i,1,newxsize);

        thischar = space;
        for (j = 0; j < newxsize; j++) {

/***********************************************************************/

/*  Encoding of bytes:
*
*  123 precedes #'s 0-63
*  124 precedes #'s 64-127
*  125 precedes #'s 128-191
*  126 precedes #'s 192-255
*  overall:  realchar = (specialchar - 123)*64 + (char-32)
*            specialchar = r div 64 + 123
*            char = r mod 64 + 32
*/

/***********************************************************************/

                     c = *thischar++;            /* get byte to send */

                     if (c > 31 && c < 123) {
                            putchar(c);
                     }
                     else {
                            putchar((c>>6)+123);
                            putchar((c & 0x3f) + 32);
                     }
              }
       }

       free(space);
}

/********************************************************************/

/*  rleit
*
*  Compress the data to go out with a simple run-length encoded scheme.
*
*/
```

```
rleit(buf,bufto,len)
       int len;
       char *buf,*bufto;
       {
       register char *p,*q,*cfoll,*clead;
       char *begp;
       int i;

       p = buf;
       cfoll = bufto;                          /* place to copy to */
       clead = cfoll + 1;

       begp = p;
       while (len > 0) {                       /* encode stuff until gone */

              q = p + 1;
              i = len-1;
              while (*p == *q && i+120 > len && i) {
                     q++;
                     i--;
              }

              if (q > p + 2) {                 /* three in a row */
                     if (p > begp) {
                            *cfoll = p - begp;
                            cfoll = clead;
                     }
                     *cfoll++ = 128 | (q-p);   /* len of seq */
                     *cfoll++ = *p;            /* char of seq */
                     len -= q-p;               /* subtract len of seq */
                    p = q;
                     clead = cfoll+1;
                     begp = p;
              }
              else {
                     *clead++ = *p++;          /* copy one char */
                     len--;
                     if (p > begp + 120) {
                            *cfoll = p - begp;
                            cfoll = clead++;
                            begp = p;
                     }
              }

        }
/*
*  fill in last bytecount
*/
       if (p > begp)
              *cfoll = 128 | (p - begp);
       else
              clead--;                         /* don't need count position */

       return((int)(clead - bufto));           /* how many stored as encoded */
}
```