Interactive Color Raster Graphics

This chapter introduces the Interactive Color Raster protocol and describes how to use this protocol in your programs to display color graphics with NCSA Telnet for the Macintosh®, how to control raster graphics windows, and how to display and manipulate color images. The chapter concludes with a sample program you can use as a template for designing programs that use the ICR protocol.

Interactive Color Raster Graphics

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 for the Mac and shares characteristics of the Tektronix graphics terminal emulation protocol. For example, escape sequences control the display.

You can use ICR to write mainframe programs that display color images in their own windows on your Macintosh screen, and you can apply the full range of 256 colors out of a palette of 16 million colors to your graphics displays. The ICR protocol is intended for use on a 256-color Macintosh.

Starting and Quitting ICR Graphics Emulation

To use ICR, you need a program that runs on the remote host computer. This program must give all appropriate commands to conduct ICR graphics emulation. To create an ICR program, work from the protocol description contained later in this chapter in the "Using the ICR Protocol" section and from the example in the "Sample Program for ICR in C" section.

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

To quit ICR emulation, the ICR program on the remote computer can 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 exit NCSA Telnet while some windows remain open, the windows close automatically.

Using the ICR Protocol

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, or display raster graphics.

Begin all ICR graphics sequence commands with the escape sequence ESC^ (escape, caret) to insure that NCSA Telnet can determine the difference between regular text and ICR graphics.

Description of the Protocol

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 the table below.

^ is the caret character (ASCII 94).

Parameters is one or more of the parameters of *X* described in the table below. 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 (^).

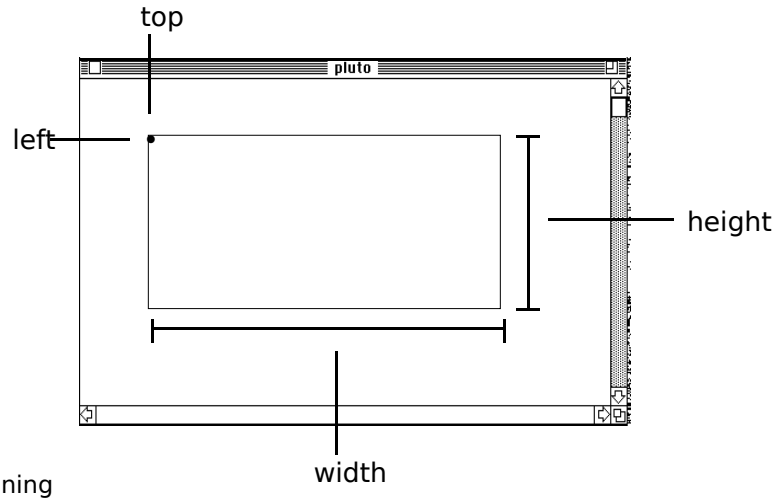
Each command can be followed by a data stream (*data*). If a command requires a data stream, the stream follows the command.

Command characters and their parameters are described in the table below:

Command Parameters Description

W left; top; width; height; display; windowname — Creates a window at the specified location on the screen, where 0, 0 is the upperleft corner of the screen.

- The *left*, *top*, *width*, and *height* integers specify location and size on the screen:



Integer	Meaning
left	the pixel value of the horizontal, or x, location of the upper-left corner of the graphics w
top	the pixel value of the vertical, or y, location of the upper-left corner of the graphics win
height	the number of pixels that comprise the vertical height of the graphics window
width	the number of pixels that comprise 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 a string that distinguishes multiple windows. The windowname assigned to a window is used by all other commands to specify that window.

D windowname — Destroys a window by physically removing it from the screen and memory.

- *Windowname* is the unique name assigned to a window when it is created 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 palette entries are 8-bit R, G, and B, 3 bytes per entry, in that order. The default palette is a straight gray-scale ramp, where 0=black and 255=white.(See "Color Maps" section later in this chapter.)

- 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. The count parameter

should be followed with the command's data stream.
- *Windowname* is the unique name assigned to a window when it is created by the W command.

R x; y; expand length; windowname — Indicates that the data to follow are run-length encoded. (See the "Run-Length Encoding Format" section later in this chapter.)
- The *x* and *y* integers identify the point where the raster line starts and the data follow for length bytes of encoded 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 when it is created by the W command.

P x; y; expand; length; windowname — Indicates that the data to follow are pixel data.
- The *x* and *y* integers identify the point where the raster line starts and the 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 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 when it is created by the W command.

I x; y; expand; length; windowname — Indicates that the data to 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 *length* 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 when it is created by the W command.

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 login 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)

The codes above work to 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: On CTSS, trailing spaces are trimmed, so 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's length on the originating machine before encoding.

Run-Length Encoding Format

Data for the run-length encoded line are first run-length compressed and then ASCII encoded. Therefore 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. (*n*) represents the lower seven bits of the byte. The high bit represents whether the following (*n*) characters are reproduced exactly (high bit = 0) or whether the following single character 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 into 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

Color Maps

You can 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 might be use:

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

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

Note that the count field (12 in this example) is always 3 times the length value (4 in this example).

ICR Graphics Windows

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 dialog box and does not create the window.

Allocating Memory

If you are using MultiFinder, you can prevent running out of memory by setting NCSA Telnet's allocated memory size to a larger value. For example, if you need space for two 256 x 256 image windows, increase the memory for NCSA Telnet by 128K (256 bytes x 256 bytes, or 64K, for each window).

Copying a 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:

Click in the graphics window to bring it to the front.

Choose **Copy** from the **Edit** menu. Now you can paste the graphic into another Macintosh application.

System Color Problems

Image windows use the colors available for display 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.

NOTE: Pressing CONTROL-C or using other methods to interrupt ICR commands, can make NCSA Telnet appear to lock up. (See also "Telnet Options" in Chapter 3, "Advanced Features.") When this occurs, either press RETURN several times or enter commands until the session window resumes activity. It may help to remember that when you issue a drawing command NCSA Telnet expects an influx of a certain number (often hundreds) of bytes of image data to finish drawing the current line.

Sample Program for ICR in C

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

```
/*   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 */
}
```