# PROPOSED SMPTE RECOMMENDED PRACTICE



# Test Patterns and Test Images for DPX Leader



## Contents



## 1 Scope

**1.1** This practice defines a series of two-dimensional test pattern files intended as an aid in evaluating film recorders and scanners and verifying compatibility between systems. These patterns are defined only in digital form, but can be written to film via a film recorder and scanned back into the digital domain using a film scanner. It should be noted that, depending on the characteristics and calibration of the film recorder or scanner, the resulting images may not meet the specifications of the patterns given in this practice.

**1.2** DPX test patterns are in two groups of pixel based images for evaluation: geometric and colorimetric. Each file in each group represents an aspect ratio independent, resolution independent, high contrast, non-bandlimited, test image to help in understanding orientation, geometry, focus, and colorimetry. However, the geometry and focus test objectives are illustrated with specific resolutions to convey a clear example to users.

**1.3** Groups of DPX test patterns to evaluate features are illustrated in figure 1 and arranged as follows in a group of six frames:

Test objectives

- Geometric test objectives:
  - orientation, frame 1
  - aspect ratio, frame 1
  - geometry, frame 1
  - steadiness, frame 1
  - focus/resolution, frame 2
- Color reproduction test objectives:
  - gray-scale evaluation, frame 3
  - color evaluation, frame 4
  - DAD test objective, frame 5
  - flat field, frame 6

## 2 Definitions

### 2.1 Film stocks

Film stocks for scanning and recording, such as 5245 and 5248 color-negative stocks, are typical low- and medium-speed camera films intended for general motion-picture production. There are also internegative stocks such as 5244 intended for making color master positives from original color-negative stocks. These films are not intended for projection. A positive color print film intended specifically for making color release prints from original color negatives is 5293. Although there are scanning and recording technologies for digitizing and printing all these film bases, this practice generally refers to the scanning of negative and print films and recording only on negative materials.







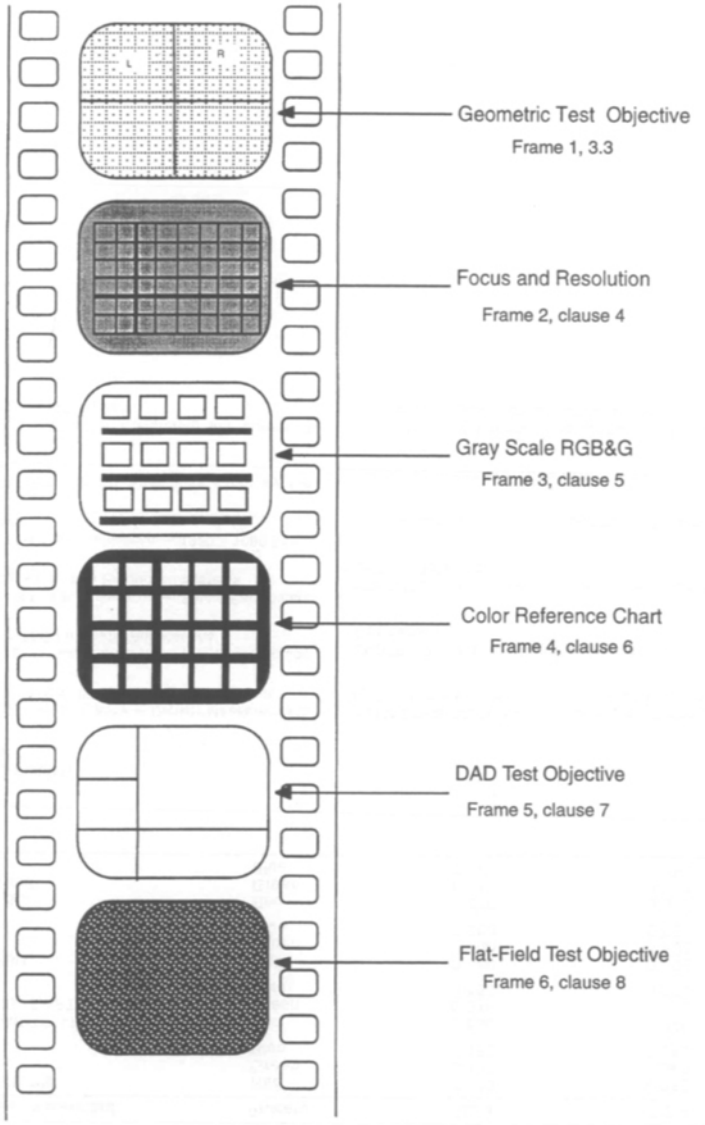



Figure 1 – DPX leader

### 2.2 Digital scanning and recording technologies

There are three main scanning and recording technologies: CRT (cathode ray tube), EBR (electron beam recorders), and LBR (laser beam recorders). CRT technologies are commonly referred to as raster point plotting devices using black-and-white CRTs and color filters to record film materials. Some CRT technologies are capable of scanning as well as recording. EBR technologies are flying-spot devices with the film and spot inside the vacuum chamber for exposure. LBR technologies are characterized by their light source, the laser. They are used to scan and record with tricolor colinear and aligned beams in a flying-spot configuration. CCDs (charge-coupled devices) have been introduced recently for scanning. A CCD is a light detection device and therefore can only be used for digitizing images. In this practice, the word *scanning* refers to the input or digitization segment and the word *recording* refers to the output or film recording segment of the process.

## 3 Geometric test objective

This is frame 1 of the DPX sequence of test patterns outlined in the scope of this practice and shown in figure 1.

### 3.1 Orientation features

The orientation test image (frame 1, figure 1) features a simple text message L for left and R for right, and a machine-readable irregular triangular symbol to help in determining emulsion side and top and bottom of the image. The L symbol is placed in a box to the left side (left of center ruler) of the image and the R symbol is placed in a box on the right side (right of center ruler) of the geometric and steadiness test pattern. There is also a set of binary divided rulers that are placed at the center of the pattern in the vertical and in the center of the pattern for the horizontal axis. These rulers extend from top to bottom in the vertical dimension of the test image and from side to side for the horizontal dimension. The ruler is used as a reference applied to the defined aperture. Digital center is positioned at the center of the camera aperture as defined by annex A. This test image is intended to be an engineering tool to permit quantitative measurements and system adjustments to the output image. This test image is a digital image describing X (across) and Y (down) with positions and offsets at (0,0) the upper left-hand corner of the display. The figure can be replicated to a camera aperture using aperture centers defined in annex A (see figure 2 and 3.3 for definitions).

### 3.2 Aspect ratio features

The aspect ratio test image feature includes a grid to determine the spacing of the pixels, a scale in X and Y dimensions in machine-readable form, and a dataset to determine the shape of the frame, square or unsquare. The test image in figures A.1 and A.2 specifies the dimensions of the camera and scanning apertures with the relative positions of the vertical and horizontal centerlines of the image with respect to the perforations. The aspect ratio of an image is usually expressed as a real number by dividing the image horizontal dimension by the image vertical dimension. This digital image can be used to check the aspect ratio of the film-recording device by recording it to film and comparing it on a pin-block to the film format defined in ANSI/SMPTE 139, 102, 111, and 59 for 35-mm film, and ANSI/SMPTE 215 for 65-mm film (see tables A.1, A.2, and A.3 in annex A). Style A is an Academy aperture for spherical projection (flat), style B is for anamorphic projection (scope), and style C is full aperture (super 35). When an image on film is exchanged for digitization, care should be exercised in the scanning of the full aperture. If the aperture is scanned at less than the camera aperture specification, a subaperture image will be processed through the system. Without rescaling, the frame, when output to film, will result in a subaperture format not conforming to the standard.

### 3.3 Geometric, size and steadiness features

This resolution independent test image (frame 1, figure 1) is produced to test spot repeatability and spot placement or position in X and Y. The size of the image is 24 boxes by 18 boxes. Each box is constructed with 4-pixel wide lines 128×128 pixels whose half widths equal 2 pixels, each 75% red, green, blue, and gray. This will result in a 1.33:1 aspect ratio test pattern of 2400 pixels by 1800 pixels. Inside each box is another test pattern to confirm spot integrity. It consists of a 36×36 pixel grid of nine 12×12 pixel boxes. In each box there are alternating red, green, blue, and gray 3×3 pixel boxes by 3 pixels wide with an empty pixel in the middle. Every other box will have a 1×1 pixel in alternating values as above except for the center of the pattern which has a 7×7 antialiased circle to ensure spot aspect ratio integrity seen in figure 2. Alternate boxes can be made with 2×2 pixel boxes.





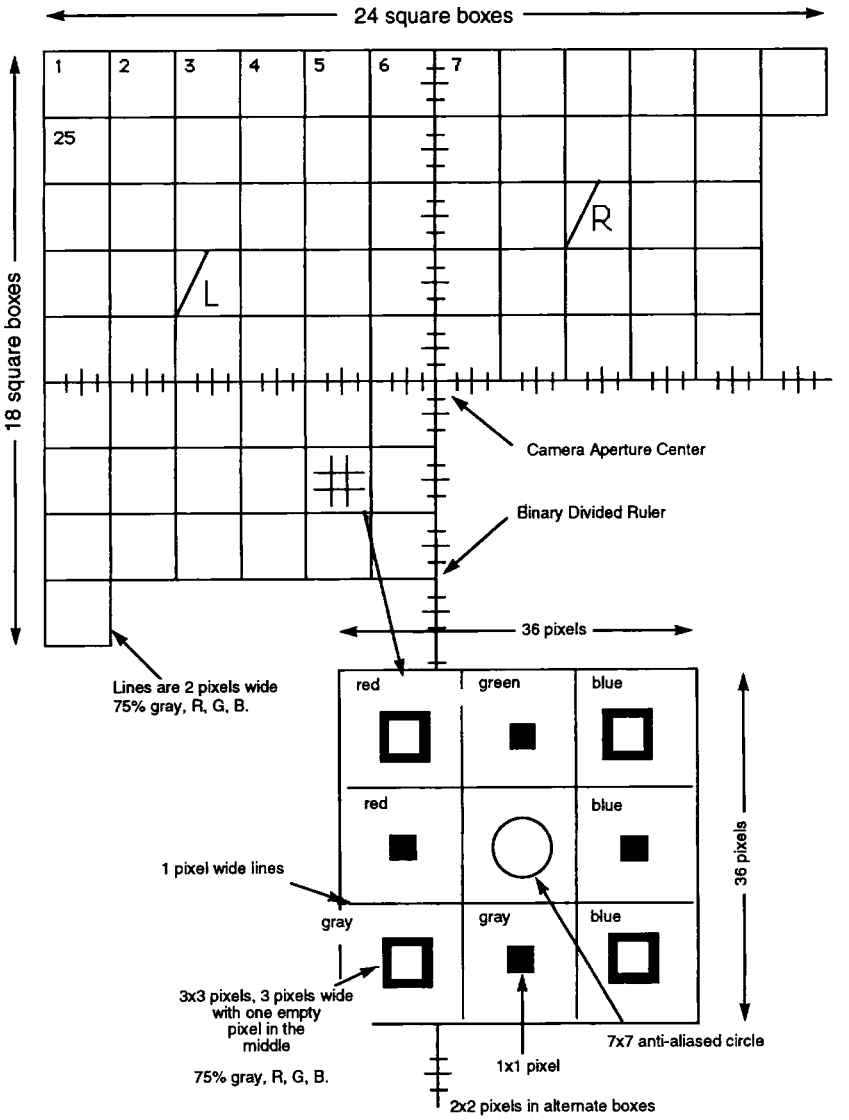



**Figure 2 – Geometric and steadiness test pattern**

This pattern in multiple frame sequences can be an exposed latent image against a known steady grid from any optical printer to determine unsteadiness and integrity of the computer-generated test pattern. The optical test pattern can be exposed on an optical printer of known accuracy (± 0.0002 in). The undeveloped latent image of the optical printer is loaded in the computer output printer. The output printer exposes multiple frames of the above geometric chart and the composite image is developed and screened in a calibrated projector (the code for generating this test pattern is in annex B).

The steadiness test image must have a base using a pin-registered movement from an optical reference camera of known tolerances (± 0.0002 in). The geometric test image with a machine readable center (left, right, lower and upper corners) of a frame can be used to automate the X, Y position measurement in a series of latent image frames (steadiness comparison) with a tolerance of ± 1/2 pixel.

Anamorphic versions of the above test image should be generated in a 2X squeezed format, so that when they are displayed the geometry appears square.

## 4 Focus and resolution features

This is frame 2 of the DPX test pattern sequence illustrated in figure 1.

The focus and resolution features are to characterize and determine the maximum number of line pairs that can be reproduced from a given I/O system. The resolution target outlined is both high and low contrast in primary colors at various positions in the frame. All targets are spatially band-limited to allow subpixel measurements. A digital DPX resolution target and metric are illustrated in figure 3, with a two-dimensional sinusoidal pattern and a one-on-one-off pattern. Annex C contains sample c code to generate a digital zone plate, and annex D contains sample c code to generate a digital DPX resolution target. The digital DPX resolution target is produced by arranging each element in the target consisting of two patterns at right angles to each other. Each pattern contains three lines and two spaces of equal width and length five times the width. The change in pattern size progresses geometrically as the sixth root of two or conversely the spatial resolution count doubles with every sixth element.

The two-dimensional sinusoidal zone plate in figure 3 is made of linear sine waves in log space. In addition to the sinusoidal pattern, a one-on-one-off test pattern in vertical and horizontal dimensions is presented. This pattern measures a recorder's depth of modulation capability and serves to access the fundamental spot characteristics of any recording system.

Each test pattern contains the above information in the following format: The upper left-hand pattern is the resolution target. The upper right target is the zone plate. The lower left target is the vertical one-on (pixel), one-off pattern, and the lower right is the horizontal one-on, one-off. The four-quadrant suite is a 512x512 pixel array (as shown in figure 3). This array is replicated in the test frame from the center outward to fill the frame (shown in figure 1, frame 2).

## 5 Color reproduction gray-scale objectives

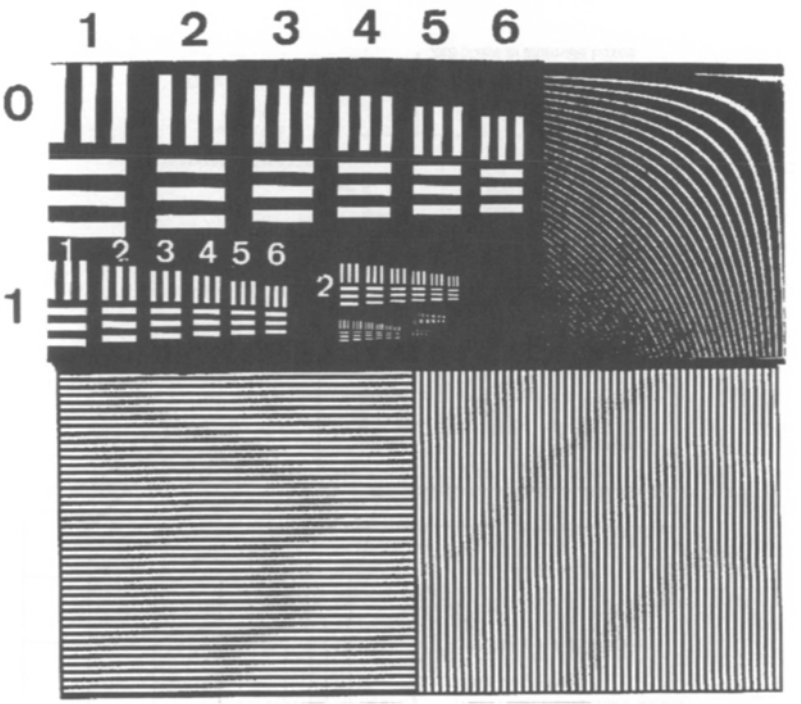
This test pattern is frame 3 of the DPX test pattern sequence shown in figure 1.

Gamma can be specified mathematically and visually as a series of step wedges or step tablets in a single frame or as a series of frames representing the contrast range of values specified for a given scanner and recorder system on a specific film base. A step tablet of 21 steps has been suggested. Careful attention should be paid to the manufactured film base specification to ensure that proper settings are used on the recorder. A typical representation has been suggested. This range is also descriptive of the content of specific scenes in a given shot that provides reflectance characteristics of the test target. It has also been suggested that the range of digital values in the test target exceed available film technologies.

Levels in the gray-scale wedge should be defined as a logarithmic series with approximately equal perceptual steps, and described by a linear scale that is normalized from 0% to 100%, where linear is defined as linear with light. The actual digital code values are computed by multiplying this scale by the reference white-code value minus the reference black-code value and adding the reference black-offset term, e.g., 255 and 0 of many 8-bit RGB systems. It is recognized that images will be converted through gamma-correction lookup tables for display, and that other lookup tables may be used for conversion to logarithmic or other data storage formats.








| | Group number | | | | | | |
|---|---|---|---|---|---|---|---|
| Element No. | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1.00 | 2.00 | 4.00 | 8.00 | 16.00 | 32.0 | 64.0 |
| 2 | 1.12 | 2.24 | 4.49 | 8.98 | 17.95 | 36.0 | 71.8 |
| 3 | 1.26 | 2.52 | 5.04 | 10.1 | 20.16 | 40.3 | 80.6 |
| 4 | 1.41 | 2.83 | 5.66 | 11.3 | 22.62 | 45.3 | 90.5 |
| 5 | 1.59 | 3.17 | 6.35 | 12.7 | 25.39 | 50.8 | 102.0 |
| 6 | 1.78 | 3.56 | 7.13 | 14.3 | 28.51 | 57.0 | 114.0 |

Cycles/mm

**Figure 3 – DPX focus/resolution target**

In practice, the test images should be converted into the same data representation as the images they accompany.

A wedge or tablet can be calibrated with a photoelectric densitometer provided that the geometric and spectral characteristics and calibration of the densitometer are such that it reads visual diffuse density. It is assumed that the film material's spectral response has been matched to the light source with filtration or wavelength selection so printing densities measured match visual densities observed. The step tablet should be constructed along the guidelines set down in figure 4 and figure 1, frame 3 in R,G,B and gray. Color ramps of continuous color are also very helpful in determining quantization errors in computation and in the hardware A/D and D/A converters of any digital system. If a computation or hardware error occurs, the ramp image will be rendered with a noticeable knee or banding to the continuous color ramp images. Each patch is 200 pixels × 120 pixels minimum to allow the sampling by a 1-mm to 3-mm densitometer aperture. The R,G,B ramp patterns are defined as 100 pixel wide strips by the width of the desired frame. They are generated from zero to full on in the particular color ramp being generated.

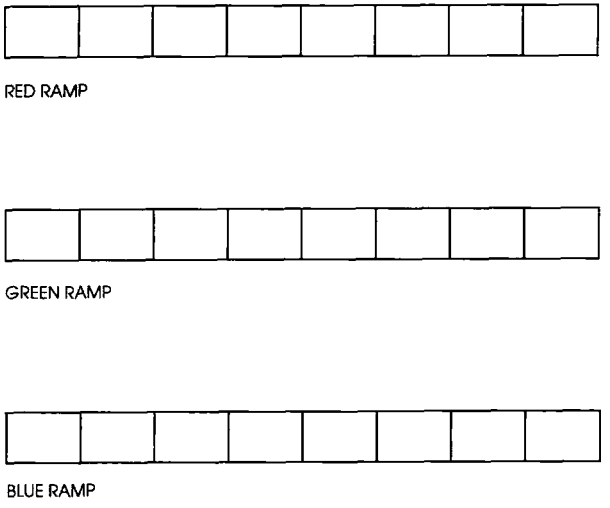



**Figure 4 – Gray-scale test objective**





## 6 Color reference objectives

This is frame 4 of the DPX test pattern sequence illustrated in figure 1.

A physical color reference chart has been developed to facilitate quantitative or visual evaluation of color reproduction processes employed in photography, television, and printing. The 4×6 array of patches, each 50-mm square, includes spectral simulations of light and dark human skin, foliage, and blue sky. Additive and subtractive primaries and a six-step neutral scale are included for analytical studies. Each patch is characterized by a spectral reflectance factor, assigned name: CIE (1931) x,y and Y, Munsell notation, and ISCC-NBS name. The use of SMPTE 303M, tables 3 and 9, in CIE XYZ space is recommended. This standard describes in detail the construction of each test patch. The digital code values for the color patches should be generated based on ITU-R BT.709 phosphors, $D_{55}$ color temperature, assuming a linear metric. Users would have to apply the appropriate transfer function to use it for display or output to film (see figure 1, frame 4).

## 7 DAD test objective

This is frame 5 of the sequence of DPX test patterns shown in figure 1.

DAD is the digital aim density standard and is based on LAD. LAD is the laboratory aim density standard used by film processing laboratories to control density on the film to reduce batch-to-batch variability. The test patch includes in the upper left a full on white patch, the middle left patch is black, and the lower left is a gray patch at 18% gray. The upper right and greater portion of the frame is reserved for an appropriate gray-scale (real) image such as the traditional *China Girl* or *LAD Girl*. The *DAD Girl* reproduction suggestions for replication are: subject head shot in 6100 K, back light is full sun at 10,000 fc with front light at 2500 fc and a 0.6ND filter over a 100-mm lens at t5 on 5245 film. In this example, the digital range is 11 bits, normalized liner scale, 0 to 2047. Flesh tones measured at the neck area in shadow (gray) of red = 700-0-0, green = 0-700-0, blue = 0-0-700 with a $D_{min}$ = code value (intensity number) 62, and gray patch DAD code value of 445. The lower middle to right contains primary color patches of red, green, blue, cyan, magenta, and yellow for color-separation identification (see figure 1, frame 5).

## 8 Gray flat field

This test pattern is frame 6 of the DPX leader shown in figure 1. A full-frame 18% gray flat-field test frame is suggested. This test frame is used in the evaluation of field flatness characteristics of a recorder system (see figure 1, frame 6). Eighteen percent gray means that the value for red, green, and blue pixel is 18% of the intensity range. For example, if an 8-bit 18% gray flat field is being built, the range would be 0 - 255 × 0.18 = value 46 for the red, green, and blue pixels using a normalized liner scale. If the range is 10 bits, the value for gray would equal 369; and for 12 bits, it would be 737.

**Annex A** (informative)
**Reproductions from proposed revision of ANSI/SMPTE 59-1991**

Styles A, B, and C camera aperture image areas from ANSI/SMPTE 59 are given in figures A.1 and A.2 and tables A.1 through A.3.

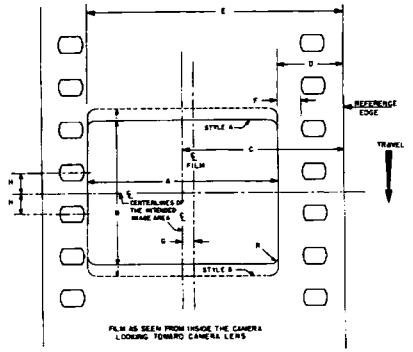



**Figure A.1 – Styles A and B camera aperture image area**

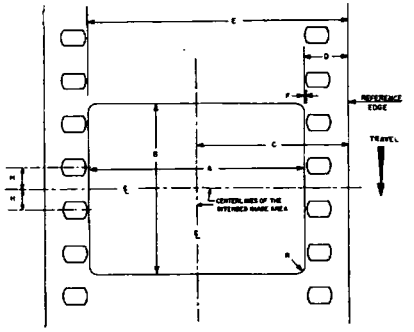



**Figure A.2 – Style C camera aperture image area**

**Table A.1 – Style A**

| Dimensions | Inches | | Millimeters | |
|---|---|---|---|---|
| A | 0.866 | nom | 22.00 | nom |
| B | 0.63 | + 0.02 − 0.00 | 16.0 | + 0.5 − 0.0 |
| C | 0.738 | ± 0.002 | 18.75 | ± 0.05 |
| D | 0.305 | max | 7.75 | max |
| E | 1.171 | min | 29.74 | min |
| F | 0.116 | nom | 2.95 | nom |
| G | 0.050 | nom | 1.27 | nom |
| H | 0.093 | ± 0.002 | 2.36 | ± 0.05 |
| R | 0.03 | max | 0.8 | max |

**Table A.2 – Style B**

| Dimension | Inches | | Millimeters | |
|---|---|---|---|---|
| B | 0.732 | + 0.008 − 0.000 | 18.59 | + 0.20 − 0.00 |

**Table A.3 – Style C**

| Dimensions | Inches | | Millimeters | |
|---|---|---|---|---|
| A | 0.981 | nom | 24.92 | nom |
| B | 0.735 | ± 0.002 | 18.67 | ± 0.05 |
| C | 0.688 | ± 0.002 | 17.48 | ± 0.05 |
| D | 0.198 | max | 5.03 | max |
| E | 1.179 | min | 29.95 | min |
| F | 0.009 | nom | 0.23 | nom |
| H | 0.093 | ± 0.002 | 2.36 | ± 0.05 |
| R | 0.03 | max | 0.8 | max |





**Annex B** (informative)
**Geometric/steadiness test pattern**

```
/* vi:tabstop=4 */

# define DPX_BIGENDIAN          0x53445058      /*
"SDPX" in ASCII */
# define DPX_LITTLEENDIAN       0x58504453      /*
"XPDS" in ASCII */

/*
** "Universal Metric" defines
*/
# define B&W                    0
# define RED_PRINT              1               /*
r, g, b printing density */
# define GREEN_PRINT            2
# define BLUE_PRINT             3
# define RED_ITU                4               /*
# define GREEN_ITU              5
# define BLUE_ITU               6

/*
** Bit definitions for undefined fields
*/
# define UNDEF_8BIT             0xFF
# define UNDEF_16BIT    0xFFFF
# define UNDEF_32BIT    0xFFFFFFFFL


/*
** Defines for image orientation; L == LEFT, R == RIGHT, T
== TOP, B == BOTTOM
*/
# define LR_TB    0         /* The only one supported in
the core set file format */
# define RL_TB    1
# define LR_BT    2
# define RL_BT    3
# define TB_LR    4
# define TB_RL    5
# define BT_LR    6
# define BT_RL    7

/*
** Component Data Packing Method defines
*/
# define CDPM_PACKED      0     /*  Packed into 32-bit
words   */
# define CDPM_FILLED      1     /*  Filled to 32-bit
words   */

/*
** Component Data Encoding Method defines
*/

# define CDEM_NONE        0     /*  No encoding
method applied    */
# define CDEM_RLE         1     /*  Run length
encoding applied    */

/*
** Component data signed flag
*/
# define CDSF_UNSIGNED    0
# define CDSF_SIGNED      1
```



```
/*
** Video Signal Standard defines
*/
# define VSS_UNDEFINED                  0
# define VSS_NTSC                       1
# define VSS_PAL                        2
# define VSS_PAL_M                      3
# define VSS_SECAM                      4
# define VSS_601_525_4_350      /* YCbCr ITU-R BT.601-5
525-line, 2:1, 4:3 asp      */
# define VSS_601_625_4_351      /* YCbCr ITU-R BT.601-5
625-line, 2:1, 4:3 asp      */
# define VSS_601_525_16_9       100     /*   YCbCr
ITU-R BT.601-5 525-line, 2:1, 16:9 asp */
# define VSS_601_625_16_9       101     /*   YCbCr
ITU-R BT.601-5 625-line, 2:1, 16:9 asp */
# define VSS_1050_2_1           150     /*   YCbCr
1050-line 2:1 interlace, 16:9 asp   */
# define VSS_1125_2_1           151     /*   YCbCr
1125-line 2:1, 16:9 asp SMPTE 240M */
# define VSS_1250_2_1           152     /*   YCbCr
1250-line 2:1 interlace, 16:9 asp   */
# define VSS_525_1_1            200     /*   YCbCr
525-line, 1:1 progressive, 16:9 asp   */
# define VSS_625_1_1            201     /*   YCbCr
625-line, 1:1 progressive, 16:9 asp   */
# define VSS_787.5_1_1          202     /*   YCbCr
787.5-line, 1:1 prog., 16:9 asp     */

/*
** Image Transfer Characteristic defines
*/
# define ITC_USER_DEFINED               0
# define ITC_PRINT_DENSITY              1
# define ITC_LINEAR                     2
# define ITC_LOGARITHMIC                3
# define ITC_UNSPECIFIED                4
# define ITC_SMPTE240M                  5
# define ITC_ITU-R BT.709_1             6
# define ITC_ITU-R BT.601_625           7
# define ITC_ITU-R BT.601_525           8
# define ITC_NTSC                       9
# define ITC_PAL                        10
# define ITC_ZLINEAR                    11
# define ITC_ZHOMOGENEOUS               12

/*
** Colorimetric Specification defines
*/
# define CS_USER_DEFINED                0
# define CS_PRINT_DENSITY               1
# define CS_LINEAR                      2
# define CS_UNSPECIFIED                 4
# define CS_SMPTE240M                   5
# define CS_ITU-R BT.709_1              6
# define CS_ITU-R BT.601_625            7
# define CS_ITU-R BT.601_525            8
# define CS_NTSC                        9
# define CS_PAL                         10

*/
** Image Element Descriptor defines
*/
```

```
# define IE_USER_DEFINED        0
# define IE_RED                 1
# define IE_GREEN               2
# define IE_BLUE                3
# define IE_ALPHA               4

# define IE_LUMINANCE           6
# define IE_CHROMINANCE         7
# define IE_DEPTH               8
# define IE_COMPOSITE           9

# define IE_RGB                 50
# define IE_RGBA                51
# define IE_ABGR                52

# define IE_4_2_2               100     /* CbYCrY
        */
# define IE_4_2_2_4             101      /*
CbYaCrYa        */
# define IE_4_4_4               102     /* CbYCr */
# define IE_4_4_4_4             103     /*
CbYCra */

# define IE_USER_2COMP          150     /*      User
defined 2 component element     */
# define IE_USER_3COMP          151     /*      User
defined 3 component element     */
# define IE_USER_4COMP          152     /*      User
defined 4 component element     */
# define IE_USER_5COMP          153     /*      User
defined 5 component element     */
# define IE_USER_6COMP          154     /*      User
defined 6 component element     */
# define IE_USER_7COMP          155     /*      User
defined 7 component element     */
# define IE_USER_8COMP          156     /*      User
defined 8 component element     */

/*
** Header structure for ANSI/SMPTE 268M digital picture
standard.
** The format is referred to as the Digital Picture Exchange
Format
** (or dpx for short).  The image file suffix is ".dpx."
*/

/*
** Individual Channel data
*/
typedef struct
{
        u_long                  dataSign;
        u_long                  minData;
        float                   minQuantity;
        u_long                  maxData;
        float                   maxQuantity;
        u_char                  descriptor;
        u_char                  transferChar;
        u_char                  colorSpec;
        u_char                  bitSize;
        u_short                 packing;
        u_short                 encoding;
        u_long                  dataOffset;
        u_long                  eolPadding;
        u_long                  eoiPadding;
        char                    desc[32];
} DPX_ELEMENT;
```





```
/*
** Generic (Fixed Format) portion of header
*/
typedef struct
{
        /*
        ** File information
        */
        u_long                  magic;
        u_long                  imgDataOffset;
        char                    version[8];
        u_long                  imageFileSize;
        u_long                  dittoKey;
        u_long                  genericHdrLen;
        u_long                  industryHdrLen;
        u_long                  userDefHdrLen;
        char                    fileName[100];
        char                    createDateTime[24];
        char                    creator[100];
        char                    projectName[200];
        char                    copyright[200];
        u_long                  encryptKey;
                /* default is UNDEF_32BIT (OFF)      */
        char                    rsvd1[104];

        /*
        ** Image information
        */
        u_short                 imageOrient;
        /* Standard Orientation is LR_TB */
        u_short                 numElements;
        u_long                  pixelsPerLine;
        u_long                  linesPerImage;
        DPX_ELEMENT             elements[8];
        char                    rsvd2[52];

        /*
        ** Image orientation information
        */
        u_long                  xOffset;
        u_long                  yOffset;
        u_long                  xCenter;
        u_long                  yCenter;
        u_long                  xOrigSize;
        u_long                  yOrigSize;
        char
        sourceImageName[100];
        char                    sourceDateTime[24];
        char                    inputDevice[32];
        char                    inputDeviceSN[32];
        u_short                 borderValidity[4];      /*
XL, XR, YT, YB border   */
        u_long                  aspectRatio[2];
        /* horizontal, vertical         */
        char                    rsvd3[28];
} DPX_HEADER;

/*
** Section II, Motion Picture Industry Specific (Fixed Format)
*/
typedef struct
{
        u_char          filmMfgID;              /*
2 digits from film edge code    */
        u_char          filmType;       /*   2 digits
from film edge code     */
        u_char          offsetInPerfs;  /*   2 digits
```





```
from film edge code     */
        u_char          unused;                 /*
zero, for byte alignment        */
        u_long          prefix;                 /*
6 digits from film edge code    */
        u_long          count;                  /*
4 digits from film edge code    */
        char            format[32];             /*
e.g., Academy, etc              */
        u_long          framePosition;  /*   In the
sequence                        */
        u_long          sequenceLength; /*   In
frames                          */
        u_long          heldCount;              /*
1 == default                    */
        float           frameRate;              /*
of original (frames/s)          */
        float           shutterAngle;   /*   of
camera, in degrees              */
        char            frameID[32];    /*   e.g.,
keyframe                        */
        char            slate[100];             /*
Slate information               */
        char            rsvd[60];       /* TBD
                                        */

} FILM_SUBHDR;

/*
** Section II, Television Industry Specific (Fixed Format)
*/
typedef struct
{
        u_long          timeCode;
                /* SMPTE time code                  */
        u_long          userBits;
        /* SMPTE user bits              */
        u_char          interlace;
        /* 0 == non, 1 == 2:1           */
        u_char          fieldNumber;
        /* 0 or 1                               */
        u_char          videoStandard;
        /* video signal std             */
        u_char          unused;
                /* zero for byte align      */
        float           horizontalSampleRate;   /*
in Hz                           */
        float           verticalSampleRate;
        /*  in Hz                       */
        float           temporalSampleRate;
        /* or frame rate in Hz          */
        float           timeToFirstPixel;       /*

from sync in ms         */
        float           gamma;
        float           blackLevelCode;
        float           blackGain;
        float           breakPoint;
        float           refWhiteLevelCode;
        float           integrationTime;
        char            rsvd[76];
} TV_SUBHDR;

/*
* Section III - User defined (variable length)
* Starts at offset 2048 and has a length defined in the
generic header.
*/

/* vi:set tabstop=4 */
# include <stdio.h>
# include <unistd.h>
# include <errno.h>
# include <sys/types.h>
# include <time.h>
# include <strings.h>
# include <malloc.h>
# include <bstring.h>
# include "dpx.h"

/*
** This was coded on an SGI workstation running OS rev 5.2
** To make: cc -o main main.c
*/

# define SE             strerror( errno )
# define XSIZE          2400    /* (XSIZE * 3) must
be a multiple of 4 (wordlen) */
# define YSIZE          1800
# define BOXSDOWN       18
# define BOXSACROSS     24
# define NUM_CHANS      3
# define FILENAME       "testpat.dpx"
# define BOXSIZE        100
# define PIXBYTES       3       /* bytes/pixel (RGB)
*/

# define LEFT   0
# define RIGHT  1
# define BOTTOM 2
# define TOP            3

# define PLAIN          0
# define GRETICLE       1
# define GRATLEN                10      /* in pixels,
of the short one */

/*
** Globals
*/
char            *progName;
u_char          gray75[] = { 192, 192, 192 }; /*   75%
gray */

/*
** Weights for antialiased circle generation
*/
float circle[] = {

0.1, 0.4, 0.9, 1.0, 0.9, 0.4, 0.1,
0.4, 1.0, 0.6, 0.5, 0.6, 1.0, 0.4,
0.9, 0.6, 0.3, 0.2, 0.3, 0.6, 0.9,
1.0, 0.5, 0.2, 0.0, 0.2, 0.5, 1.0,
0.9, 0.6, 0.3, 0.2, 0.3, 0.6, 0.9,
0.4, 1.0, 0.6, 0.5, 0.6, 1.0, 0.4,
0.1, 0.4, 0.9, 1.0, 0.9, 0.4, 0.1,

};

/*
** Prototypes
*/
void write_dpx_header( FILE * );
```

```
void make_row_of_boxes( int, u_char ** );
void set_box_side( u_char **, int, int, int );
void set_L( u_char **, int );
void set_R( u_char **, int );
void set_test_pattern( u_char **, int, int );
void set_line( u_char **, int );
void set_box( u_char **, int, int, int, int, int );

/*
** This program creates a 24-bit (8 bits/component) test pattern for testing
** orientation, geometry, size and steadiness. The image file is in SMPTE RP 193 DPX
** format. The image is 2400 across by 1800 down. The image is
** byte filled with no encoding/compression.
**
** This image is not designed to be projected with an anamorphic lens.
*/
void
main( int argc, char **argv )
{
        int                     i, j;
        FILE            *fp;
        u_char          *boxes[BOXSIZE];

        /*
        ** Grab program name
        */
        if( (progName = rindex( argv[0], '/' )) == NULL )
                progName = argv[0];
        else
                progName++; /* Step over the '/' */
        /*
        ** Open the file.
        */
        if( (fp = fopen( FILENAME, "w+" )) == NULL )
        {
                fprintf( stderr, "%s: can't create '%s' because %s.", progName,
                        FILENAME, SE );
                fclose( fp );
                exit( -1 );
        }

        write_dpx_header( fp );     /*   write the dpx header */

        /*
        ** Malloc a row of boxes.
        */
        for( i = 0; i < BOXSIZE; i++ )
        {
                if( (boxes[i] = (u_char*)malloc( XSIZE * PIXBYTES )) == NULL )
                {
                        fprintf( stderr, "%s: can't malloc a row of boxes.\n", progName );
                        fclose( fp );
                        exit( -1 );
                }
        }

        /*
        ** Now, actually do the work and write the test image.
```



```
This chunk of code
        ** doesn't know anything about the test image internals.  make_row_of_boxes
        ** contains all the arcane knowledge.
        */
        for( i = 0; i < BOXSDOWN; i++ )
        {
                make_row_of_boxes( i, boxes );

                /*
                ** Write the row of boxes
                */
                for( j = 0; j < BOXSIZE; j++ )
                {
                        if( fwrite( boxes[j], PIXBYTES, XSIZE, fp ) != XSIZE )
                        {
                                fprintf( stderr, "%s: can't write %s because %s.\n",
                                        progName, FILENAME, SE );
                                fclose( fp );
                                for( i = 0; i < BOXSIZE; i++ )
                                        free( boxes[i] );
                                exit( -1 );
                        }
                }
        }

        /*
        ** Clean up...
        */
        for( i = 0; i < BOXSIZE; i++ )
                free( boxes[i] );
        fclose( fp );
        exit( 0 );
}

/*
** Make a row (BOXSACROSS) of BOXSIZExBOXSIZE image boxes. This routine
** knows the intimate secrets of the test pattern
*/
void
make_row_of_boxes( int down, u_char **b )
{
        int i, boxsize, across;

        /*
        ** Clear out old patterns
        */
        for( i = 0; i < BOXSIZE; i++ )
                bzero( b[i], XSIZE * PIXBYTES );

        boxsize = 1;

        /*
        ** On the current row (down), build all the boxes across one at a time.
        */
        for( across = 0; across < BOXSACROSS; across++ )
        {
                /*
```





```
                ** First, fabricate the sides of the current box; this involves
                ** determination of any required gretlcles.
                */
                if( across == BOXSACROSS/2 )
                        set_box_side(   b,    across, LEFT, GRETICLE );
                else
                        set_box_side(   b,    across, LEFT, PLAIN );

                if( across == (BOXSACROSS/2)-1 )
                        set_box_side(   b,    across, RIGHT, GRETICLE );
                else
                        set_box_side(   b,    across, RIGHT, PLAIN );

                if( down == (BOXSDOWN/2)-1 )
                        set_box_side(   b,    across, BOTTOM, GRETICLE );
                else
                        set_box_side(   b,    across, BOTTOM, PLAIN );

                if( down == BOXSDOWN/2 )
                        set_box_side(   b,    across, TOP, GRETICLE );
                else
                        set_box_side(   b,    across, TOP, PLAIN );

                /*
                ** Now see if the current box needs the L or R orientation characters;
                ** otherwise drop in the spot integrity test pattern (with alternating
                ** 1x1 and 2x2 box sizes).
                */
                if( down == 3 && across == 2 )
                        set_L( b, across );
                else if( down == 2 && across == (BOXSACROSS/2)+2 )
                        set_R( b, across );
                else
                        set_test_pattern(   b,    across, boxsize );

                /*
                ** alternate test pattern bottom box size
                */
                if( boxsize == 1 )
                        boxsize = 2;
                else
                        boxsize = 1;
        }
}

/*
** "Draw" a 3x3 box with a hole in it at the specified position and color
*/
void
set_box( u_char **b, int x, int y, int rv, int gv, int bv )
{
        int             i;
        u_char  color[3];

        color[0] = (u_char)rv;
        color[1] = (u_char)gv;
        color[2] = (u_char)bv;

        for( i = 0; i < 3; i++ )
                memcpy( &b[y][x+(i*PIXBYTES)], color, PIXBYTES );

        memcpy( &b[y+1][x], color, PIXBYTES );
        memcpy( &b[y+1][x+(2*PIXBYTES)],    color, PIXBYTES );
        for( i = 0; i < 3; i++ )
                memcpy( &b[y+2][x+(i*PIXBYTES)], color, PIXBYTES );
}

/*
** "Draw" a solid spot of the specified size, position and color
*/
void
set_spot( u_char **b, int size, int x, int y, int rv, int gv, int bv )
{
        int             i, j;
        u_char  color[3];

        color[0] = (u_char)rv;
        color[1] = (u_char)gv;
        color[2] = (u_char)bv;

        for( i = 0; i < size; i++ )
                for( j = 0; j < size; j++ )
                        memcpy( &b[y+i][x+(j*PIXBYTES)], color, PIXBYTES );
}

/*
** Generate the 36x36 pixel test pattern inscribed in every square of the
** test image.
*/
void
set_test_pattern( u_char **b, int box, int size )
{
        int             i, x, y, indx, x1, y1, x2, y2;
        u_char  color[3];

        indx = (BOXSIZE*PIXBYTES) * box;/* left edge of current box */
        indx += (32 * PIXBYTES);                        /* left edge of pattern in box */

        /*
        ** Draw the tic-tac-toe board
        */
        for( i = 0; i < 36; i++ )       /* horizontal bars */
        {
                memcpy( &b[32+12][indx+(i*PIXBYTES)], gray75, PIXBYTES );
                memcpy( &b[32+24][indx+(i*PIXBYTES)], gray75, PIXBYTES );
        }
```

```
        for( i = 32; i < 32+36; i++ )   /* vertical bars */
        {
                memcpy( &b[i][indx+(12*PIXBYTES)],
gray75, PIXBYTES );
                memcpy( &b[i][indx+(24*PIXBYTES)],
gray75, PIXBYTES );
        }

        /*
        ** Draw the four corner boxes with holes in them.
        */
        x1 = indx + (5*PIXBYTES);     /* left side of box */
        x2 = indx + (29*PIXBYTES);
        y1 = 36;                                        /*
 top of box */
        y2 = 61;
        set_box( b, x1, y1, 255, 0, 0 );
        set_box( b, x2, y1, 0, 0, 255 );
        set_box( b, x1, y2, gray75[0], gray75[1], gray75[2]);
        set_box( b, x2, y2, 0, 0, 255 );

        /*
        ** Draw the four solid boxes of the specified size
        */
        set_spot( b, size, x1+(13*PIXBYTES), y1+1, 0,
255, 0 );/* top */
        set_spot( b, size, x1+PIXBYTES, y1+14, 255, 0, 0 );
                /* left */
        set_spot( b, size, x1+(25*PIXBYTES), y1+14, 0, 0,
255 );/* right */
        set_spot( b, size, x1+(13*PIXBYTES), y2+1,
gray75[0], gray75[1], gray75[2]);

        /*
** Now, the antialiased 7x7 circle.
        */
        x1 = indx + (15*PIXBYTES);/* upper left start
coords */
        y1 = 47;

        i = 0;
        for( y = y1; y < y1 + 7; y++ )
        {
                for( x = 0; x < 7; x++, i++ )
                {
                        color[0] = (u_char)(circle[i] *
gray75[0]);     /* set the color (antialias) */
                        color[1] = (u_char)(circle[i] *
gray75[1]);
                        color[2] = (u_char)(circle[i] *
gray75[2]);
                        memcpy(
b[y][x1+(x*PIXBYTES)], color, PIXBYTES );
                }
        }
}

/*
** Draw a 4-pixel wide un-antialiased line from the lower left
of the current
** box to the center of the top.  "Scanline" 0 is at the top.
*/
void
set_line( u_char **b, int box )
{
        int             i, j, indx, d;

        indx = (BOXSIZE*PIXBYTES) * box;   /* left edge
of current box */
        for( d = 0, i = BOXSIZE-1; i >= 0; i--, d = 1 - d )
        {
                indx += (d * PIXBYTES);
                for( j = 0; j < 4; j++ )
                        memcpy(
&b[i][indx+(j*PIXBYTES)], gray75, PIXBYTES );
        }
}

/*
** Draw an angled line above a 4-pixel wide "L" in the
specfied box.
** The values were empirically derived...
*/
void
set_L( u_char **b, int box )
{
        int i, j, indx, start, end;
        set_line( b, box );

indx = (BOXSIZE*PIXBYTES) * box;/* left edge of current
box */

        indx += (BOXSIZE/2) * PIXBYTES;             /*
Step to the middle of the box */

        start = BOXSIZE/3; /*    how far down to start
"drawing" */
        end = BOXSIZE-20;
        for( i = start; i < end; i++ )
                for( j = 0; j < 4; j++ )
                        memcpy(   &b[i][indx   +
(j*PIXBYTES)], gray75, PIXBYTES );

        start = end;
        end = start + 4;
        for( i = start; i < end; i++ )
                for( j = 0; j < 25; j++ )
                        memcpy(   &b[i][indx   +
(j*PIXBYTES)], gray75, PIXBYTES );
}

/*
** Draw an angled line above a 4-pixel wide "R" in the
specfied box.
** The values were empirically derived.
*/
void
set_R( u_char **b, int box )
{
        int             i, j, k, d, indx, start, end, mid, c;

        set_line( b, box );

        indx = (BOXSIZE*PIXBYTES) * box;   /* left edge
of current box */

        indx += (BOXSIZE/2) * PIXBYTES;             /*
Step to the middle of the box */

        /*
        ** Draw the left part of the "R"
        */
        start = BOXSIZE/3;
        end = BOXSIZE-15;
```





```
        mid = start + (end-start)/2;c = start + (mid-start)/2;
        for( i = start; i < end; i++ )
        {
                for( j = 0; j < 4; j++ )
                        memcpy(   &b[i][indx   +
(j*PIXBYTES)], gray75, PIXBYTES );
        }

        /*
        ** Draw the top and middle
        */
        for( i = start; i < start+4; i++ )
                for( j = 0; j < 19; j++ )
                        memcpy(   &b[i][indx   +
(j*PIXBYTES)], gray75, PIXBYTES );

        for( i = mid; i < mid+4; i++ )
                for( j = 0; j < 19; j++ )
                        memcpy(   &b[i][indx   +
(j*PIXBYTES)], gray75, PIXBYTES );

        /*
        ** Draw the foot
        */
        k = indx + (12 * PIXBYTES);

        for( d = 1, i = mid+4; i < end; i++, d = 1 - d )
        {
                k += (d * PIXBYTES);
                for( j = 0; j < 4; j++ )   /*   4 pixels
wide */
                        memcpy(
&b[i][k+(j*PIXBYTES)], gray75, PIXBYTES );
        }

        /*
        ** Finish the "R"
        */
        k = indx + (16 * PIXBYTES);
        for( d = 0, i = start; i <= c; i++, d = 1 - d )
        {
                k += (d * PIXBYTES);
                for( j = 0; j < 4; j++ )
                        memcpy(
&b[i][k+(j*PIXBYTES)], gray75, PIXBYTES );
        }

        for( d = 0, i = c; i < mid; i++, d = 1 - d )
        {
                k -= (d * PIXBYTES);
                for( j = 0; j < 4; j++ )
                        memcpy(
&b[i][k+(j*PIXBYTES)], gray75, PIXBYTES );
        }
}

/*
** Fill in the side of the specified box
*/
void
set_box_side( u_char **b, int box, int side, int type )
{
        int i, j, indx, inc;

        indx = (BOXSIZE*PIXBYTES) * box;/* left edge
of current box */

        /*
        ** Draw the outside border (2 pixel wide lines)
        */
        switch( side )
        {
        case TOP:
                for( i = 0; i < BOXSIZE; i++, indx +=
PIXBYTES )
                {
                        memcpy( &b[0][indx], gray75,
PIXBYTES );
                        memcpy( &b[1][indx], gray75,
PIXBYTES );
                }
                break;

        case BOTTOM:
                for( i = 0; i < BOXSIZE; i++, indx +=
PIXBYTES )
                {
                        memcpy(     &b[BOXSIZE
-2][indx], gray75, PIXBYTES );
                        memcpy(     &b[BOXSIZE
-1][indx], gray75, PIXBYTES );
                }
                break;

        case LEFT:
                for( i = 0; i < BOXSIZE; i++ )
                {
                        memcpy( &b[i][indx],  gray75,
PIXBYTES );
                        memcpy(
&b[i][indx+PIXBYTES], gray75, PIXBYTES );
                }
                break;

        case RIGHT:
                indx += (BOXSIZE-2)*PIXBYTES;
                for( i = 0; i < BOXSIZE; i++ )
                {
                        memcpy( &b[i][indx],  gray75,
PIXBYTES );
                        memcpy(
&b[i][indx+PIXBYTES], gray75, PIXBYTES );
                }
                break;

        }

        if( type == PLAIN )
                return;

        /*
        ** Now draw the required greticle
        */
        switch( side )
        {
        case TOP:
                indx = (BOXSIZE*PIXBYTES) * box;
                inc = (BOXSIZE/4) * PIXBYTES;     /*
1/4 box over */
                for( i = 0; i < GRATLEN; i++ )
                {
```

```
                        memcpy(     &b[i][indx+inc],
gray75, PIXBYTES );
                        memcpy(  &b[i][indx+(2*inc)],
gray75, PIXBYTES );
                        memcpy(   &b[i][indx+(3*inc)],
gray75, PIXBYTES );
                }
                /*
                ** Draw middle one twice as long.
                */
                for( i = GRATLEN; i < 2*GRATLEN; i++ )
                        memcpy(   &b[i][indx+(2*inc)],
gray75, PIXBYTES );
                break;

        case BOTTOM:
                indx = (BOXSIZE*PIXBYTES) * box;
                inc = (BOXSIZE/4) * PIXBYTES;     /*
1/4 box over */
                for( i = BOXSIZE-GRATLEN;      i <
BOXSIZE; i++ )
                {
                        memcpy(     &b[i][indx+inc],
gray75, PIXBYTES );
                        memcpy(   &b[i][indx+(2*inc)],
gray75, PIXBYTES );
                        memcpy(   &b[i][indx+(3*inc)],
gray75, PIXBYTES );
                }
                /*
                ** Draw middle one twice as long.
                */
                for( i = BOXSIZE-(2*GRATLEN); i <
BOXSIZE-GRATLEN; i++ )
                        memcpy(   &b[i][indx+(2*inc)],
gray75, PIXBYTES );
                break;

        case LEFT:
                inc = (BOXSIZE/4);/* 1/4 box down */
                for( i = 0; i < BOXSIZE; i += inc )
                {
                        for( j = 0; j < GRATLEN; j++ )
                                memcpy(
&b[i][indx+(j*PIXBYTES)], gray75, PIXBYTES );
                }
                i = BOXSIZE/2;
                for( j = GRATLEN; j < 2*GRATLEN; j++ )
                        memcpy(
&b[i][indx+(j*PIXBYTES)], gray75, PIXBYTES );
                break;

        case RIGHT:
                indx = (BOXSIZE*PIXBYTES) * box;  /*
left edge of current box */
                inc = (BOXSIZE/4);/* 1/4 box down */
                indx += (BOXSIZE-
GRATLEN)*PIXBYTES;
                for( i = 0; i < BOXSIZE; i += inc )
                {
                        for( j = 0; j < GRATLEN; j++ )
                                memcpy(
&b[i][indx+(j*PIXBYTES)], gray75, PIXBYTES );
                }
                indx = (BOXSIZE*PIXBYTES) * box;  /*
left edge of box */
                indx        +=        (BOXSIZE-
(2*GRATLEN))*PIXBYTES;
                        i = BOXSIZE/2;
                        for( j = 0; j < GRATLEN; j++ )
                                memcpy(
&b[i][indx+(j*PIXBYTES)], gray75, PIXBYTES );

                        break;
                }
}

/*
** Write the DPX header to the file
*/
void
write_dpx_header( FILE *fp )
{
        int                     i;
        DPX_HEADER      hdr;
        FILM_SUBHDR     subhdr;
        time_t          t;
        long            *l;
        struct tm  *tm;

        /*
        ** Set up the dpx header
        */
        bzero( (void*)&hdr, (int)sizeof(DPX_HEADER) ); /*
clear the header */

        /*
        ** Fill in the file information
        */
        hdr.magic = DPX_BIGENDIAN;
        hdr.imgDataOffset = sizeof(DPX_HEADER) +
sizeof(FILM_SUBHDR);
        strcpy( hdr.version, "V1.0" );

        /*
        ** Be sure that if XSIZE changes, it's a multiple of
4 because
        ** each scanline is padded to the nearest word
(32 bit) boundary.  2400
        ** across is nice because that's 7200 bytes or
1800 words -wrb.
        */
        hdr.imageFileSize = sizeof(DPX_HEADER) +
sizeof(FILM_SUBHDR) + XSIZE*YSIZE;

        hdr.dittoKey = UNDEF_32BIT;
        hdr.genericHdrLen = sizeof(DPX_HEADER);
        hdr.industryHdrLen = sizeof(FILM_SUBHDR);
        hdr.userDefHdrLen = 0;
        strcpy( hdr.fileName, FILENAME );

        /
        ** Fill in time info
        */
        time( &t );
        tm = localtime( &t );
        /*
        ** YYYY:MM:DD:HH:MM:SS:LTZ
        */
        sprintf(                hdr.createDateTime,
"%04d:%02d:%02d:%02d:%02d:%02d:%s",
# ifdef sgi
                        tm->tm_year + 1900,
# else
```





```
                        tm->tm_year + 1990,   /* IBM PVS */
# endif
        tm->tm_mon+1, tm->tm_mday, tm->tm_hour, tm-
>tm_min, tm->tm_sec, tzname[daylight] );
        strcpy( hdr.creator, progName );      /*
originating program */
        strcpy( hdr.copyright, "" );
        hdr.encryptKey = UNDEF_32BIT;

        /*
        ** Fill in the Image information
        */
        hdr.imageOrient = LR_TB;
        hdr.numElements = 1;
        hdr.pixelsPerLine = XSIZE;
        hdr.linesPerImage = YSIZE;
        hdr.xOffset = 0;
        hdr.yOffset = 0;
        hdr.xCenter = UNDEF_32BIT;
        hdr.yCenter = UNDEF_32BIT;
        hdr.xOrigSize = XSIZE;
        hdr.yOrigSize = YSIZE;
        strcpy( hdr.sourceImageName, FILENAME );
        strcpy( hdr.sourceDateTime, hdr.createDateTime
);
        strcpy( hdr.inputDevice, "Software by
<wrb@wrb.com>" );
        for( i = 0; i < 4; i++ )
                hdr.borderValidity[i] = UNDEF_16BIT;
        hdr.aspectRatio[0] = UNDEF_32BIT;
        hdr.aspectRatio[1] = UNDEF_32BIT;

        /*
        ** Fill in channel data
        */
        for( i = 0; i < NUM_CHANS; i++ )
        {
                hdr.elements[i].dataSign          =
CDSF_UNSIGNED; /* unsigned */
                hdr.elements[i].minData = 0;
                hdr.elements[i].minQuantity = 0.;
                hdr.elements[i].maxData = 255;
                        /* 8-bit data */
                hdr.elements[i].maxQuantity       =
(float)255;
                hdr.elements[i].descriptor = IE_RGB;
        /* RGB only */
                hdr.elements[i].transferChar      =
IE_USER_DEFINED;
                hdr.elements[i].colorSpec         =
IE_USER_DEFINED;
                hdr.elements[i].bitSize = 8;
                /* 8-bit data */
                hdr.elements[i].packing           =
CDPM_FILLED;
                hdr.elements[i].encoding          =
CDEM_NONE;
                hdr.elements[i].dataOffset        =
UNDEF_32BIT;
                hdr.elements[i].eolPadding        =
UNDEF_32BIT;
                hdr.elements[i].eolPadding        =
UNDEF_32BIT;
        }

        /*
        ** Fill in unused elements
        */
        for( i = NUM_CHANS; i < 8; i++ )
        {
                long *lp;

                hdr.elements[i].dataSign          =
UNDEF_32BIT;
                hdr.elements[i].minData           =
UNDEF_32BIT;
                lp = (long*)&hdr.elements[i].minQuantity;
                *lp = UNDEF_32BIT;      /* setting a
floating pt number */
                hdr.elements[i].maxData           =
UNDEF_32BIT;
                lp                                =
(long*)&hdr.elements[i].maxQuantity;
                *lp = UNDEF_32BIT;      /* setting a
floating pt number */
                hdr.elements[i].descriptor        =
UNDEF_8BIT;
                hdr.elements[i].transferChar      =
UNDEF_8BIT;
                hdr.elements[i].colorSpec         =
UNDEF_8BIT;
                hdr.elements[i].bitSize           =
UNDEF_8BIT;
                hdr.elements[i].packing           =
UNDEF_16BIT;
                hdr.elements[i].encoding          =
UNDEF_16BIT;
                hdr.elements[i].dataOffset        =
UNDEF_32BIT;
                hdr.elements[i].eolPadding        =
UNDEF_32BIT;
                hdr.elements[i].eolPadding        =
UNDEF_32BIT;
        }

        /*
        ** Write the main header
        */
        if( fwrite( &hdr, sizeof(DPX_HEADER), 1, fp ) != 1 )
        {
                fprintf( stderr, "%s: can't write header to
'%s' because %s.",
                        progName, FILENAME, SE );
                fclose( fp );
                exit( -1 );
        }

        /*
        ** Fill in the film subheader
        */
        bzero( (void*)&subhdr, (int)sizeof(FILM_SUBHDR)
);

        subhdr.filmMfgID = UNDEF_8BIT;
        subhdr.filmType = UNDEF_8BIT;
        subhdr.offsetInPerfs = UNDEF_8BIT;
        subhdr.unused = UNDEF_8BIT;
        subhdr.prefix = UNDEF_32BIT;
        subhdr.count = UNDEF_32BIT;
        subhdr.fra mePosition = UNDEF_32BIT;
        subhdr.sequenceLength = UNDEF_32BIT;
        subhdr.heldCount = UNDEF_32BIT;
        l = (long*)&subhdr.frameRate;
        *l = UNDEF_32BIT;/* setting a floating pt number
```

```
*/
        l = (long*)&subhdr.shutterAngle;
        *l = UNDEF_32BIT;/* setting a floating pt number
*/

        /*
        ** Write the film subheader
        */
        if( fwrite( &subhdr, sizeof(FILM_SUBHDR), 1, fp )
!= 1 )
                {
                        fprintf( stderr, "%s: can't write subheader
to '%s' because %s.",
                                progName, FILENAME, SE );
                        fclose( fp );
                        exit( -1 );
                }
}
```

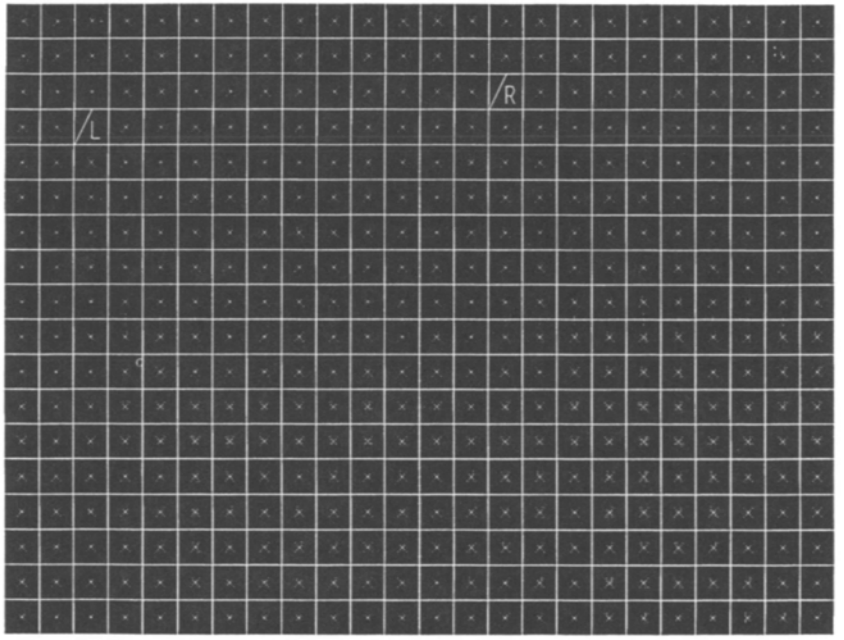



**Figure B.1 – Geometric/steadiness test output**





## Annex C (informative)
## Focus/resolution (zone plate) test pattern

```
#include <tiffio.h>
#include <math.h>
#include <stdio.h>
#include <sys/file.h>

unsigned char  *Pic;

int make_pattern (unsigned char *p, int w, int h);
int write_image (char *imagename, int w, int h);

/*-------------------------------------------------
** This was debugged on an SGI workstation running OS rev 5.2
** To make: cc -o TestPaternA smpteTestPatterna.c -ltiff -lm
-------------------------------------------------*/

main (int argc, char **argv)
{
   int  w, h;
   char  *imagename;

   if (argc < 2)  {
    fprintf (stderr, "Not enough arguments.\nusage: %s imagename
[4k]\n\n", argv[0]);
     exit (0);

   }  else  {
      imagename = argv[1];
   }

   if (argc > 2)  {
      w = 4096;
      h = 2458;
   }  else  {
      w = 2048;
      h = 1229;
   }

   if ((Pic = (unsigned char *)malloc(w*h)) == NULL)  {
      perror ("malloc");
      exit (0);
   }

   bzero (Pic, w*h);

   printf ("making %ik pattern\n", w/1024);
   make_pattern (Pic, w, h);
}

   printf ("writing %s image\n", imagename);
   write_image (imagename, w, h);
}

/*-------------------------------------------------*/

int write_image (char *imagename, int w, int h)
{
   register int  i, j, rows;
   unsigned short  nc = 3;
   unsigned char tline[16400];
   unsigned char *pp, *tp;
   TIFF  *tfo;

   /* open image */
   if ((tfo = TIFFOpen (imagename, "w")) == NULL)  {
      perror ("output file");
      exit (0);
   }

   TIFFSetField (tfo, TIFFTAG_SUBFILETYPE, 0L);
   TIFFSetField (tfo,
TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB);
   TIFFSetField (tfo, TIFFTAG_IMAGEWIDTH, w);
   TIFFSetField (tfo, TIFFTAG_IMAGELENGTH, h);
   TIFFSetField (tfo, TIFFTAG_BITSPERSAMPLE, 8);
   TIFFSetField (tfo,
TIFFTAG_COMPRESSION, COMPRESSION_LZW);
   TIFFSetField(tfo,TIFFTAG_ORIENTATION,
ORIENTATION_TOPLEFT);
   TIFFSetField (tfo, TIFFTAG_SAMPLESPERPIXEL, nc);
   TIFFSetField(tfo,TIFFTAG_PLANARCONFIG,
PLANARCONFIG_CONTIG);

   rows = 65536/(w*nc);
   TIFFSetField (tfo, TIFFTAG_ROWSPERSTRIP, rows);

   /* save image */
   pp = Pic;
   for (i = 0; i < h; i++)  {
      tp = tline;
      for (j = 0; j < w; j++, tp+=3)  {
         tp[0] = tp[1] = tp[2] = *pp++;
      }
      TIFFWriteScanline (tfo, tline, i, 0);
   }
   TIFFClose (tfo);
}

/*-------------------------------------------------*/

int make_pattern (unsigned char *p, int w, int h)
{
   int i, j, k;
   register float freq;
   register int period;

   period = w / 4;
   freq = 2*M_PI / period;

   for (i = 0; i < h; i++)  {
      k = i % period;
      for (j = 0; j < w; j++)  {
        *p++ = 255. * (.5 * (1. + cosf (M_PI + (j%period)*k*freq)));
      }
   }
}
```

An example of zone plate output is given in figure C.1.

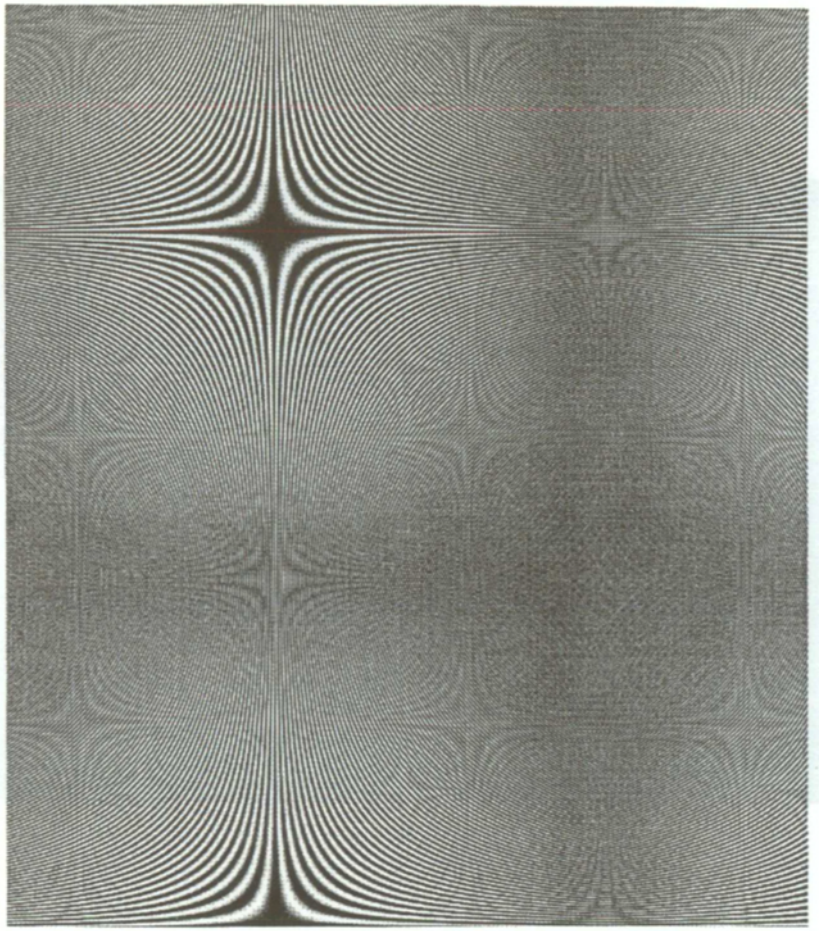


**Figure C.1 – Zone plate output example**





## Annex D (informative)
## Focus/resolution test pattern

```
#include <tiffio.h>
#include <math.h>
#include <stdio.h>
#include <sys/file.h>

#define ROOT .89089872

int make_pattern (unsigned char *p, int row, int col);
int pattern (unsigned char *p, int row, int col, int scale);
int draw_sign (unsigned char *p, float x, float y, float scale);
int fill_rect (unsigned char *p, float x0, float y0, float x1, float y1);
int hsliver (unsigned char *p, float x0, float y0, float x1, float y1);
int vsliver (unsigned char *p, float x0, float y0, float x1, float y1);

unsigned char  *Pic;

int Width;

/*-------------------------------------------------------*/

main (int argc, char **argv)
{
   register int  i, j, h, w;
   int  x, y, nv, nh;
   unsigned int  iw, ih, rows, pred;
   unsigned short  nc;
   unsigned char  *pp, *tp, tline[16400];
   TIFF  *tfo;

   if (argc > 2)  {
      w = 4096;
      h = 2458;
   }  else  {
      w = 2048;
      h = 1229;
   }
    Width = w;
    nc = 3;

   if ((Pic = (unsigned char *)malloc(w*h)) == NULL)  {
      perror ("malloc");
      exit (0);
   }

   if ((tfo = TIFFOpen (argv[1], "w")) == NULL)  {
      perror ("output file");
      exit (0);
   }

   TIFFSetField(tfo,TIFFTAG_PHOTOMETRIC,
PHOTOMETRIC_RGB);
   TIFFSetField (tfo, TIFFTAG_IMAGEWIDTH, w);
   TIFFSetField (tfo, TIFFTAG_IMAGELENGTH, h);
   TIFFSetField (tfo, TIFFTAG_BITSPERSAMPLE, 8);
   TIFFSetField(tfo,TIFFTAG_COMPRESSION,
COMPRESSION_LZW);
   TIFFSetField(tfo,TIFFTAG_ORIENTATION,
ORIENTATION_TOPLEFT);
   TIFFSetField (tfo, TIFFTAG_SAMPLESPERPIXEL, nc);
   TIFFSetField(tfo,TIFFTAG_PLANARCONFIG,
PLANARCONFIG_CONTIG);
   TIFFSetField (tfo, TIFFTAG_PREDICTOR, 2);
```



```
   rows = 65536/(w*nc);
   TIFFSetField (tfo, TIFFTAG_ROWSPERSTRIP, rows);

   bzero (Pic, w*h);
   nv = h/609;
   nh = w/992;

   printf ("nh %d, nv %d\n", nh, nv);
   for (i= 0; i < nv; i++)  {
      y = i*609 + (i*(h-nv*609))/(nv-1);
      for (j = 0; j < nh; j++)  {
         x = j*992 + (j*(w-nh*992))/(nh-1);
         printf ("make pattern xy %d %d\n", x, y);
         make_pattern (Pic, y, x);
      }
   }

   pp = Pic;
   for (i = 0; i < h; i++)  {
      tp = tline;
      for (j = 0; j < Width; j++, tp+=3)  {
         tp[0] = tp[1] = tp[2] = *pp++;
      }
       TIFFWriteScanline (tfo, tline, i, 0);
   }
    TIFFClose (tfo);
}

/*-------------------------------------------------------*/

int make_pattern (unsigned char *p, int row, int col)
{
   int r, c;

   printf ("make_pattern %i %i\n", row, col);

   r = row;
   c = col;
   pattern (p, r, c, 32);  r+= 32*12.5;
   pattern (p, r, c, 16);  r = row + 32*12.5; c = col + 32*19;
   pattern (p, r, c, 8);  r+= 8*14.5;
   pattern (p, r, c, 4);  r = row + 40*12.5; c = col + 40*19;
   pattern (p, r, c, 2);  r+= 2*17;
   pattern (p, r, c, 1);
}

/*-------------------------------------------------------*/

int pattern (unsigned char *p, int row, int col, int scale)
{
   float  x, y, f;
   int  i;

   printf ("pattern %i %i %i\n", row, col, scale);
   x = col;
   y = row;
   f = scale;
   for (i = 0; i < 6; i++)  {
      draw_sign (p, x, y, f);
      x += f*7 + 3./scale;
      f *= ROOT;
      y += scale*11*(1-ROOT)/2;
```

```
    }
}

/*-------------------------------------------------------*/

int draw_sign (unsigned char *p, float x, float y, float scale)
{

    float dy = 2./scale;

    /*printf ("draw_sign %f %f %f\n", x, y, scale);*/

    fill_rect (p, x,        y, x+scale,   y+scale*5);
    fill_rect (p, x+scale*2, y, x+scale*3, y+scale*5);
    fill_rect (p, x+scale*4, y, x+scale*5, y+scale*5);

    fill_rect (p, x, y+scale*6+dy,  x+scale*5, y+scale*7+dy);
    fill_rect (p, x, y+scale*8+dy,  x+scale*5, y+scale*9+dy);
    fill_rect (p, x, y+scale*10+dy, x+scale*5, y+scale*11+dy);
}

/*-------------------------------------------------------*/

int fill_rect (unsigned char *p, float x0, float y0, float x1, float y1)
{
    int row, col, lastrow, lastcol, i, j;
    unsigned char *q;

    /*printf ("fill_rect %f %f %f %f\n", x0, y0, x1, y1);*/
    row = y0;
    col = x0;
    lastrow = y1;
    lastcol = x1;

    if (row == lastrow) {
        hsliver (p, x0, y0, x1, y1);
    } else if (col == lastcol) {
        vsliver (p, x0, y0, x1, y1);
    } else {
        hsliver (p, x0, y0, x1, y1);
        for (i = row+1; i < lastrow; i++) {
            q = p + i*Width + col;
            *q++ += 255 * ((col+1) - x0);
            for (j = col+1; j < lastcol; j++) {
                *q++ = 255;
            }
            *q += 255 * (x1 - lastcol);
        }
        hsliver (p, x0, (float) lastrow, x1, y1);
    }
}

/*-------------------------------------------------------*/

int hsliver (unsigned char *p, float x0, float y0, float x1, float y1)
{
    int row, col, lastcol, i, j;
    float ya;
    unsigned char *q;

    /*printf ("hsliver %f %f %f %f\n", x0, y0, x1, y1);*/

    if (y0 >= y1) return;

    row = y0;
    col = x0;
    lastcol = x1;

    ya = 255 * (y1 - y0);
    q = p + row*Width + col;

    if (col == lastcol) {
        *q += ya * (x1-x0);
    } else {
        *q++ += ya * ((col+1) - x0);
        for (i = col+1; i < lastcol; i++) {
            *q++ += ya;
        }
        *q += ya * (x1 - lastcol);
    }
}

/*-------------------------------------------------------*/

int vsliver (unsigned char *p, float x0, float y0, float x1, float y1)
{

    int row, col, lastrow, i, j;
    float xa;
    unsigned char *q;

    /*printf ("vsliver %f %f %f %f\n", x0, y0, x1, y1);*/

    if (x0 >= x1) return;

    col = x0;
    row = y0;
    lastrow = y1;

    xa = 255 * (x1 - x0);
    q = p + row*Width + col;

    if (row == lastrow) {
        *q += xa * (y1-y0);
    } else {
        *q += xa * ((row+1) - y0);
        for (i = row+1; i < lastrow; i++, q+=Width) {
            *q++ += xa;
        }
        *q += xa * (y1 - lastrow);
    }
}
```





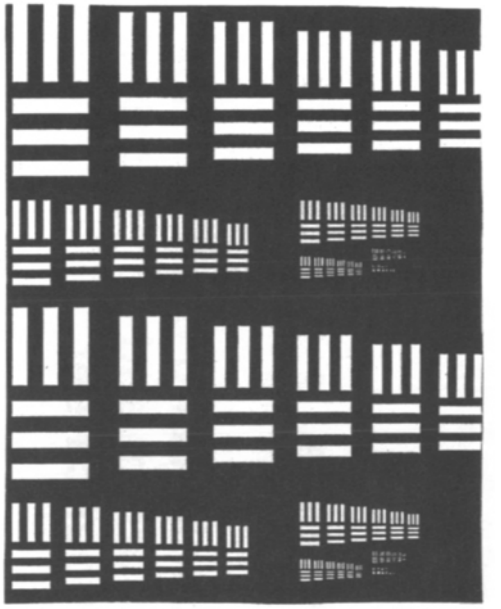


**Figure D.1 – Focus/resolution test pattern output example**

## Annex E (informative)
## Bibliography


ANSI/SMPTE 59-1991, Motion-Picture Film (35-mm) — Camera Aperture Images and Usage

ANSI/SMPTE 102-1991, Motion-Picture Film (35-mm) — Perforated CS-1870

ANSI/SMPTE 111-1996, Motion-Picture Film (35-mm) — Prints Made on Continuous Contact Printers — Exposed Areas for Picture and Audio

ANSI/SMPTE 139-1996, Motion-Picture Film (35-mm) — Perforated KS

ANSI/SMPTE 215-1995, Motion-Picture Film (65-mm) — Camera Aperture Image

ANSI/SMPTE 268M-1994, File Format for Digital Moving-Picture Exchange (DPX)

SMPTE 303M, Television — Color Reference Pattern

SMPTE RP 177-1993, Derivation of Basic Television Color Equations

SMPTE RP 187-1995, Center, Aspect Ratio and Blanking of Video Images

Military Standard, *USAF Resolving Power Test Target*, 150-A, paragraph 5.1.1.7, 1951

CIE Publication No. 15.2 (1986), Colorimetry, Second Edition

ITU-R BT.601-5, Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratio

ITU-R BT.709-2, Parameter Values for the HDTV Standard for Production and International Programme Exchange

McCamy, C.S., et al. A color-rendition chart. *Journal of Applied Photographic Engineering* Vol. 2, No. 3, 95; 1976

Pytlak, John P. and Fleischer, Alfred W. A simplified motion-picture laboratory control method for improved color duplication. *SMPTE Journal*, 85(10): 781-786; October 1976