INSIDE NIH IMAGE 1.54

General Information

About this Document

Presumably, you are going over this manual because you would like to extend the functionality of the NIH Image application to match your needs, or, you would like to write a macro. You should note that this guide was NOT written by Wayne Rasband, who is the author of the NIH Image program. This manual has been organized by Mark Vivino of NIH's Division of Computer Research and Technology. If you find errors in the manual you can blame me. You can reach me via email at mvivino@helix.nih.gov or voice at 301-496-9344 (old) or 301-xxx-xxxx (new). If you are like me, the idea of spending your whole day writing a graphical user interface is just a whole lotta wasted time. Chances are you have an image processing application that needs doing and don't want to figure out all the aspects of the Mac Toolbox (or Windows or Motif for that matter). You can build your image processing application into the NIH Image program and save yourself from a lot of wasted effort. Hopefully, this manual may help you on your route whether simple (macro) or complex (pascal). I won't claim to be an expert programmer, Mac programmer, Pascal programmer or any other language expert. In fact I'm an engineer who works primarily on clinical applications at NIH. I have simply found the NIH Image program as the best tool for any number of projects which I have been on. To say the least, having freely available and modifiable source code is not seen often with most commercial packages. This guide updated 12/22/93, current to Image version 1.53. If the default font for this document is not Times 12 point then change it to that, that is how the page layout was set.

To macro or not to macro, this is the question

It would be difficult to make a broad recommendation that your application or extension could or couldn't be developed with a macro routine. Perhaps as a general guideline I would say that if you use the same set of menu selections on a repeated basis, a macro is the best thing for you. On the other hand, if you have an iterative and constantly looping calculation, derivation, prolonged modification or anything else fairly complex you should consider using a pascal routine for that portion of your coding. The ease of the macro interface with your code executing at compiled pascal execution rates can be done with calls to UserCode in your macro. A rich set of example macro routines is distributed with the NIH Image program.

```
MACRO 'Clear Outside [C]';
{Erase region outside current selection to background color.}
BEGIN
 Copy;
 SelectAll;
 Clear;
 RestoreRoi;
 Paste;
 KillRoi;
END;
```

Simple macros, such as the one above, can save time and effort. Macros can be much larger and can be composed of your specific imaging application.

Macro Examples, Techniques & Operations

Basic macro

If you have not done so already, go to the "About Image" file and print the section on macro programming. This provides you with a complete list of all macro calls. The list is organized by the Image menus or is categorized as miscellaneous. You can write a macro with the built in Image text editor or any text editor (word processor) which can save as text. If you don't know pascal, chances are you can pick up what you need to know for macro writing fairly easily. Take a look at the numerous macro examples included with Image and those in this manual to get you started.

Macro global vs. local vars

Just as in pascal, C, or other programming languages, you can have a local or global variable. A global variable is declared at the top of the macro file and can be utilized by any procedure or macro in the file. A local variable is declared in the procedure or macro in which it is used. For the example macro set below, "A" and "B" are local to the 'Add numbers' macro. "Answer" is globally declared and used by both macros.

```
VAR
 Answer:real;

Macro 'Add numbers';
Var
 A,B: real;
begin
 A := Getnumber('Enter the first number',2.0);
 B := Getnumber('Enter the second number',3.14);
 Answer := A+B;
end;

Macro 'Show Answer';
begin
 ShowMessage(' The added result is: ', Answer:4:2);
end;
```

Operating on each image in a stack (SelectSlice)

By using a loop (for i:= 1 to nSlices) you can operate on a series of 2D images. The nSlices function returns the number of slices in the stack.

```
macro 'Reduce Noise';
var
 i:integer;
begin
  if nSlices=0 then begin
  PutMessage('This window is not a stack');
  exit;
 end;
 for i:= 1 to nSlices do begin
  SelectSlice(i);
  ReduceNoise;   {Call any routine you want, including UserCode}
 end;
end;
```

See the series of stack macros distributed with the Image program for more examples.

Using SelectWindow & SelectPic

You can use SelectPic or SelectWindow to choose an image or text window before you do any operations on it or to it. This example shows how both selectpic and selectwindow can be used. SelectWindow is somewhat easier since you do not need to know the id, or PicNumber, of the image. You need to only know the name. However, selectpic lets you operate a little more generically by not having to know the name of a window. If you are going to use SelectPic, you do need to retrieve the id by using the PicNumber function.

```
Macro 'Copy results to Text window';
VAR
 OriginalPic:integer;
 year,month,day,hour,minute,second,dayofweek:integer;
BEGIN
 OriginalPic := PicNumber;
 NewTextWindow('Image Analysis results');
 GetTime(year,month,day,hour,minute,second,dayofweek);
 Writeln('Image Analysis Laboratory');
```

```
    Writeln(Month,'/',day,'/',year);
    SelectPic(OriginalPic);
    Measure;
    ShowResults;
    Copy;
    SelectWindow('Image Analysis results');
    Paste;
END;
```

Putmessage, ShowMessage & Write

PutMessage

      PutMessage is perhaps one of the easiest ways to provide feedback to users. To use putmessage you simply call the routine with the message or string you wish to give to the user.

```
PutMessage('This macro requires a line selection');
```

You can pass multiple arguments with PutMessage if you needed to.

```
PutMessage('Have a ', 'Nice day');
```

ShowMessage

      ShowMessage allows display of calculations, data, variables or whatever you caste as a string into the Values window.
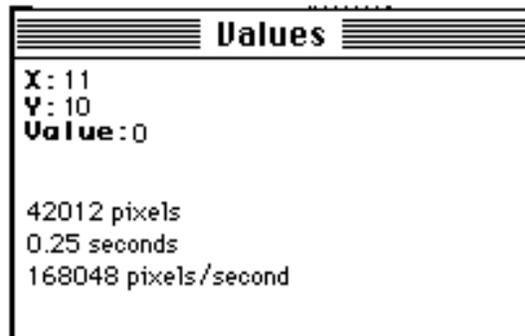
Here is a simple example of output to the Values window:

```
ShowMessage('x1 = ',x1);
```

You can use the backslash ('\') character to do a carriage return for macros:

```
ShowMessage('Average Size=',AverageSize:1:2,'\TotalCount=',TotalCount);
```

Write


```
1♦ ▤▤▤▤▤▤▤▤▤▤▤▤▤▤▤
Diameter = 7.04 mm.
```
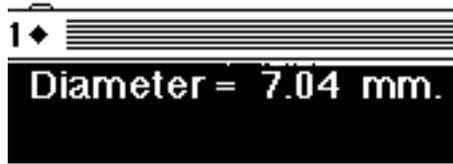
You can also write data or info onto the image window with a macro call to Write or Writeln.

```
Diameter := Width / PixelsPerMM; {in MM.}
MoveTo(300,10);
Write('Diameter = ', Diameter:5:2,'  mm.');
```

How to input a number

Making a call to getnumber will allow you to enter a number into your macro. The GetNumber macro will return a real number, or if assigned to an integer variable, such as in this example, it will not pass the decimal digits.

```
var
 MyGlobalNumber:integer;

macro 'Number input';
begin
  myGlobalNumber:=GetNumber('Enter number of iterations:',0);
end;
```



```
Enter number of iterations:

    0

   OK        Cancel
```
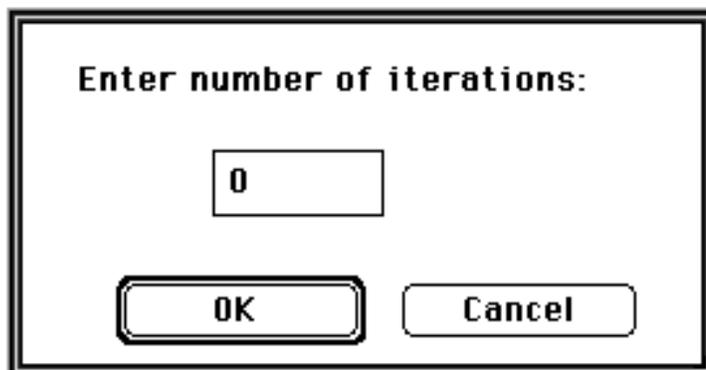
Placing macro data in the "Results" window

If you have particular information, data, calculated results, or any type of numeric data which you want to keep, you can redirect it into the Results window. Use the SetUser label commands to title your field name. The rCount function keeps the current index of the measurement counter. Since rUser1 and rUser2 are arrays, you specify the index of the array with the rCount value. See below.

```
macro 'Count Black and White Pixels [B]';
{
Counts the number of black and white pixels in the current
selection and stores the counts in the User1 and User2 columns.
}
begin
 RequiresVersion(1.44);
 SetUser1Label('Black');
 SetUser2Label('White');
 Measure;
 rUser1[rCount]:=histogram[255];
 rUser2[rCount]:=histogram[0];
 UpdateResults;
end;
```
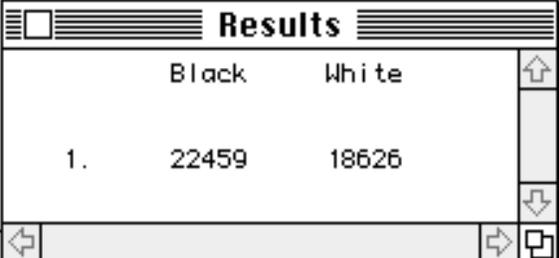

If you have more than two sets of data which you'd like to keep, then you can access other macro arrays. This includes rArea, rMean, rStdDev, rX, rY, rMin, rMax, rLength, rMajor, rMinor, and rAngle. An example of this is a snipet of code from the Export look up table macro:

```
for i:=0 to 255 do begin
  rArea[i+1]:=RedLut[i];
  rMean[i+1]:=GreenLut[i];
  rLength[i+1]:=BlueLut[i];
 end;
```

Here rArea, rMean and rLength are used for Red, Green and Blue instead of area, mean and length.

Saving results data to a tab delimeted file

You can also save data from the macro, to a tab delimeted text file by adding several commands in your macro:

```
SetExport('Measurements');
Export('YourFileName');
```

Regions of Interest (ROI)

Before you start looking at macro ROI's an introduction to coordinates is worthwhile. See the picture below for a general guideline. Regions of interest are characterized by 'marching ants' which surround a selection.
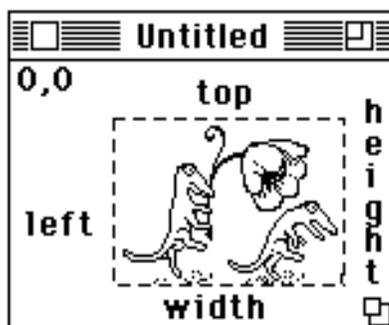
Getting ROI information

GetRoi(left,top,width,height)

You will want to call this macro routine if you need any information about the current ROI. The routine returns a width of zero if no ROI exists.

ROI creation

SelectAll

The Selectall macro command is equivalent to the Pascal SelectAll(true), which selects all of the image and shows the ROI's 'marching ants'. See the above paragraph for pascal code relating to Selectall.

MakeRoi(left,top,width,height)

This is as straight forward as the name implies.

MakeOvalRoi(left,top,width,height)

Not terribly differing to implement from MakeROI. If you want a circular ROI set width and height to the same value. See the example below.

Altering an existing ROI

MoveRoi(dx,dy)

Use to move right dx and down dy.

InsetRoi(delta)

Expands the ROI if delta is negative, Shrinks the ROI if delta is positive.

Other routines involving ROI's

RestoreROI,KillRoi

These are opposities.

Copy,Paste,Clear,Fill,Invert,DrawBoundary

Detecting the press of the mouse button

The example below shows a macro which operates until the mouse button is pressed. Button is your basic true or false boolean and becomes true when the button is pressed.

```
macro 'Show RGB Values [S]';
var
 x,y,v,savex,savey:integer;
begin
 repeat
  savex:=x; savey:=y;
  GetMouse(x,y);
  if (x<>savex) or (y<>savey) then begin
   v:=GetPixel(x,y);
   ShowMessage('loc=',x:1,', ',y:1,
     '\value=',v:1,
     '\RGB=',RedLUT[v]:1,', ',GreenLUT[v]:1,', ',BlueLUT[v]:1);
   wait(.5);
  end;
 until button;
end;
```

Accessing bytes of an image

The macro commands GetRow, GetColumn, PutRow and PutColumn can be used for accessing the image on a line by line basis. These macro routines use what is know as the LineBuffer array. This array is of the internally defined type known as LineType. Pascal routines such as GetLine use the LineType. If you plan on accessing 'lines' of the image within your macro, it would might be worth your while to examine the pascal examples in the pascal section. After looking at these, you probably will see how to use the LineBuffer array in a macro.

First look at the definition of LineType. LineType is globally declared as:

```
LineType = packed array[0..MaxLine] of UnsignedByte;
```

Naturally, UnsignedByte has been type defined as:

```
UnsignedByte = 0..255;
```

The example below is a macro which uses the linebuffer array. If you are interested in using a macro to get at image data, this example should be fairly clear.

```
Macro 'Invert lines of image'
var
 i,j,width,height:integer;
begin
 GetPicSize(width,height);
 for i:=1 to height do begin
   GetRow(0,i,width);
   for j:=1 to width do begin
    LineBuffer[j] := 255-LineBuffer[j];
    end;
    PutRow(0,i,width);
  end;
```

Reading from disk

One simple way to load data from disk is to create a window and dump information to it. An example of this is a macro which imports files created by the IPLab program. The macro reads the first 100 bytes from the file into a temporary window. It erases the window when it is through finding useful header information.

```
macro 'Import IPLab File';
var
  width,height,offset:integer;
begin
  width:=100;
  height:=1;
  offset:=0;
  SetImport('8-bit');
  SetCustom(width,height,offset);
  Import('');  {Read in header as an image, prompting for file
name.}
  width := (GetPixel(8,0)*256) + GetPixel(9,0);
  height := (GetPixel(12,0)*256) + GetPixel(13,0);
  Dispose;
  offset:=2120;  {The IPLab offset}
  SetImport('16-bit Signed; Calibrate; Autoscale');
  SetCustom(width,height,offset);
  Import('');  {No prompt this time; Import remembers the name.}
end;
```

See the pascal section for examples of reading from disk to User arrays.

Accessing an image Look Up Table (LUT)

You can modify the way an image appears by altering the RedLUT, GreenLUT and BlueLUT. This is simple and straightforward enough. You can access the RedLUT, GreenLUT and BlueLUT arrays from both macros and from Pascal.

The pascal definitions are:

```
LutArray = packed array[0..255] of byte;

RedLUT, GreenLUT, BlueLUT: LutArray;
```

Here is an example macro which finds any gray or black components in a color image and sets them to white. It's useful for seperating certain kinds of medical data.

```
macro 'Remove Equal RGB [V]';
{Changes only the LUT, removes gray component from an image}
var
 i,Value:integer;
begin
  for i:=1 to 254 do begin
    If ((RedLUT[i] = BlueLUT[i]) and (RedLUT[i] = GreenLUT[i]))
    then begin
      RedLut[i] :=255;
      BlueLut[i] := 255;
      GreenLut[i] :=255;
    end;
  end;
ChangeValues(255,255,0); {remove black}
UpdateLUT;
end;
```

Calling user written pascal from a macro

Image allows you to call by name user developed pascal routines from a macro which you write. Outlined below are example steps you can take to achieve this. You can pass into your pascal procedure up to three extended values. If you don't have any values to pass than pass a zero or any other value.

Step 1:

Write a macro or macro procedure which calls UserCode(n,p1,p2,p3). Be sure to pass values for n, p1, p2 and p3. The example below will call a routine in User.p to add and display two numbers. Note that n equals 1 in this call, because the routine calls the 1st UserMacroCode. This is further explained in step 3.

```
macro 'Add two values'
var
    NoValue:integer;
    ValueOne,ValueTwo:Real;
begin
    NoValue := 0;
    ValueOne := 2.0;
    ValueTwo := 3.14
    UserCode('AddTwoNumbers',ValueOne,ValueTwo,NoValue);
end;
```

Step 2:

Write a pascal routine in the User.p module. Again, this example simply adds two numbers and shows the result in the **Values** Window.

```
procedure AddTwoNumbers (Value1, Value2: extended);
 var
  str1, str2, str3: str255;
  Result: extended;
begin
 Result := Value1 + Value2;
 RealToString(Value1, 5, 2, str1);
 RealToString(Value2, 5, 2, str2);
 RealToString(Result, 5, 2, str3);
 ShowMessage(Concat('1st number = ', str1, cr, '2nd number = ', str2, cr, 'Added result = ', str3));
end;
```

Step 3:

Modify the UserMacroCode procedure to call your pascal procedure. The UserMacroCode procedure is found at the bottom of the User.p module. Because you could call differing UserCode routines, the string you pass into UserCode selects which routine you would like to call. This example checks to see if you have passed the string 'AddTwoNumbers'.

```
 procedure UserMacroCode (str: str255; Param1, Param2, Param3: extended);
begin
 MakeLowerCase(str);
 if pos('addtwonumbers', str) <> 0 then begin
  AddTwoNumbers(Param1, Param2);
  exit(UserMacroCode);
 end;
 ShowNoCodeMessage;
end;
```

Step 4:
      Compile your modified version of Image. Load your macro and execute away. Shown below is the result of the entire example.

```
╔═══════════ Values ═══════════╗
║ X: 190
║ Y: 374
║ Value: 0
║
║
║ 1st number =  2.00
║ 2nd number =  3.14
║ Added result =  5.14
║
```
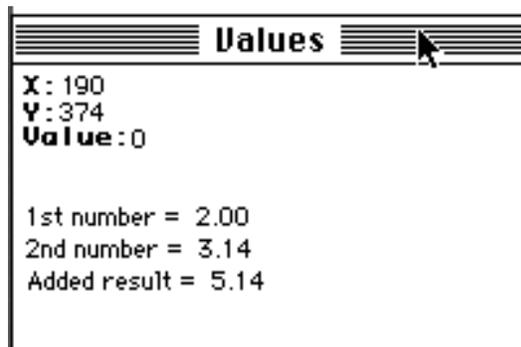
Using the Older UserCode routine

Step 1:
      For the older style Usercode call you pass a number into UserCode(n,p1,p2,p3). In the example below n equals 1, because the routine calls the 1st OldUserMacroCode. This is further explained in step 3.

```
macro 'Add two values'
{....deleted, see above}
    UserCode(1,ValueOne,ValueTwo,NoValue);
end;
```

Step 3:
      Modify the OldUserMacroCode procedure to call your pascal procedure.

```
procedure OldUserMacroCode (CodeNumber: integer; Param1, Param2, Param3: extended);
begin
 case CodeNumber of
 1:
  AddTwoNumbers(Param1, Param2);   {<====== This is the line added to do the calling}
 2:
  ShowNoCodeMessage;
```

Pascal Examples, Techniques & Operations

Getting around the Pascal project

It hardly needs stating about the usefulness of the search utilities in Think Pascal (see Find... under Search in Think Pascal). Another useful feature in Think Pascal is holding command while clicking in the top window frame. This allows easy access to procedures in a pascal unit. If you just can't seem to find the procedure you are looking for, it's not terribly hard to go to Image.p, move down to DoMenuEvent and find a reasonable starting point in your search.

The Image project as seen in Think Pascal.
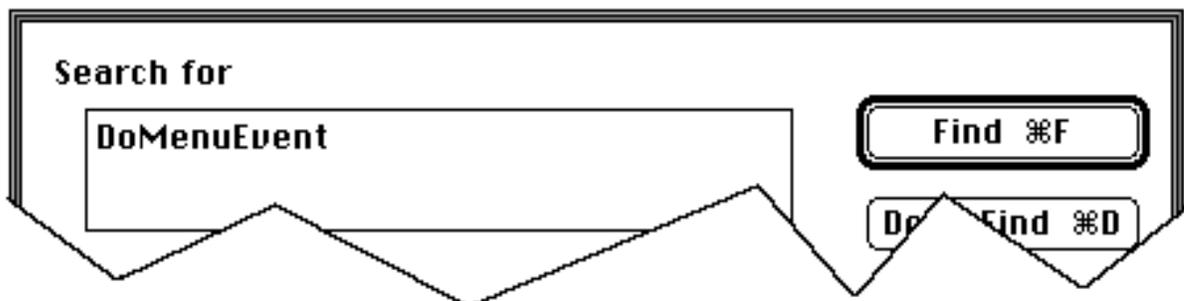
The Find... utility dialog box in Think Pascal.

```
{Each menu selection in Image has a starting point in DoMenuEvent}
procedure DoMenuEvent (MenuChoice: LongInt);
 var
```

```
MenuID, MenuItem, i, ignore: integer;
   •
FileMenu:  begin
   •
case MenuItem of
 NewItem:
```

Users can use User.p

The User.p module is a good candidate for the placement of pascal source code which you develop. Since the User.p module is strategically placed in the build order below other modules you can call just about any routine in the rest of the project. Be sure to add the module name which contains the routine you are calling to the uses command in User.p

```
uses
 QuickDraw, Palettes, PrintTraps, globals, Utilities, Graphics; <=== add module name here if you need to. Example
would be File1, File2 or any other unit.
```

Recommended addition when adding to pascal

If you plan on modifying any of the pascal units, I would personally recomend that you add two comment lines to each and every pascal modification that you do. These are:

```
{Begin Modification}
YourModification;
{End Modification}
```

You won't regret it later when you go through code you wrote a year or two ago, or if you try and read somebody else's code. It is easy to use the find utility to find your old or other peoples modifications by searching on "begin modifications'.

Returning a value from pascal to a macro

One method for returning a calculated value from a pascal routine back into a macro is to use the rUser1 or rUser2 arrays. You can return real numbers and many of them if you need too.

In Pascal have:

User1^[1] := MyReturnValue;

In the macro have:

ReturnedValue := rUser1[1];

Or if you desire seeing the output in the results window you could have a macro like this:

```
Macro 'Show table';
begin
 SetOptions('User1');
 SetPrecision(3);
 SetCounter(5);
 SetUser1Label('My 5 calc values');
 ShowResults;
end;
```

Pascal versions of SelectSlice & SelectPic

SelectSlice is available directly in pascal. You might set something up like the following:

```
if Info^.StackInfo <> nil then
 SliceCount := Info^.StackInfo^.nSlices
else
 SliceCount := 1;
for SliceNumber := 1 to SliceCount do begin
 SelectSlice(SliceNumber);
```

For SelectPic you might copy this code (taken from macros source file) and pass the PictureNumber to the routine (i.e. for PictureNumber:=1 to nPics):

```
procedure SelectImage (id: integer);
begin
 StopDigitizing;
 SaveRoi;
 DisableDensitySlice;
 SelectWindow(PicWindow[id]);
 Info := pointer(WindowPeek(PicWindow[id])^.RefCon);
 ActivateWindow;
 GenerateValues;
 LoadLUT(info^.cTable);
 UpdatePicWindow;
end;
```

Putmessage, showmessage & PutmessageWithCancel

PutMessage

PutMessage is perhaps one of the easiest ways to provide feedback to users. To use putmessage you simply call the routine with the message or string you wish to give to the user.

```
PutMessage('Capturing requires a Data Translation or SCION frame grabber card.');
```

You can pass multiple arguments with PutMessage. Doing this is a bit different is Pascal and macros.

```
PutMessage(concat('Have a ', 'Nice day'));
```

or even something like:

```
PutMessage(concat('A disastrous bug occurred at: ', Long2Str(BigBadWolf)));
```

PutMessageWithCancel

PutMessageWithCancel allows you to choose the path you might want to take in your code. Unlike putmessage, it allows you to press a cancel button. This might indicate that you should exit your procedure, such as in this example:

```
var
  item: integer;
begin
item := PutMessageWithCancel('Do you really want to do this operation?');
if item = cancel then
 exit(YourProcedure);
```

ShowMessage

ShowMessage allows display of calculations, data, variables or whatever you caste as a string into the Values window.

```
ShowMessage(CmdPeriodToStop);
```

or more involved:

```
ShowMessage(concat(str1, ' pixels ', cr, str2, ' seconds', cr, str3, ' pixels/second', cr, str));
```

How to input a number

```
function GetInt (message: str255; default: integer; var Canceled: boolean): integer;
function GetReal (message: str255; default: extended; var Canceled: boolean): extended;
```

You probably don't want to develop an entire dialog routine just to pass a number into your procedure from the keyboard. Fortunately, you don't have to. A default dialog exists for getting integers and real numbers.

```
var
  EndLoopCount:integer;
  WasCanceled:boolean;
begin
....{rest of code}
  EndLoopCount :=0;   {a default}
  EndLoopCount := GetInt('Enter number of iterations:',0,WasCanceled);
  if WasCanceled then
    exit(YourProcedureName);
```

Reading from disk

From disk to macro user arrays:

If you have tab delimited data which you want loaded into the macro User arrays, you can easily open the data with this routine. If you have more than two columns of data then use one or more of the other macro arrays. To use this routine copy it into User.p, set it up as a UserCode call and recompile Image. Note that this routine has been changed for version of Image 1.50 and above.

```
procedure OpenData;
var
 fname: str255;
 RefNum, nValues, i: integer;
 rLine: rLineType;
begin
if not GetTextFile(fname, RefNum) then
 exit(OpenData);
InitTextInput(fname, RefNum);
i := 1;
while not TextEOF do
 begin
  GetLineFromText(rLine, nValues);
  User1^[i] := rLine[1];
  User2^[i] := rLine[2];
  i := i + 1;
 end;
end;
```

If you want to see the data, take a look at the macro above in the section on returning a value from pascal to a macro.

To your own arrays:

The routine is just as applicable to those who wish to read data from disk into arrays of their own, and not the user arrays. If you have your own large arrays, you will need to allocate memory for the pointers. An example of this is shown in the section "Memory". You can open data to as many arrays as you allocate by replacing User1^[i]. Example:

```
while not TextEOF do begin
  GetLineFromText(rLine, nValues);
  xCoordinate^[i] := rLine[1];
  yCoordinate^[i] := rLine[2];
  zCoordinate^[i] := rLine[3];
```

Memory and pointer allocation

Show below is an example of dynamic memory allocation. If you plan on using a large array then you need to allocate memory for the task. You should free the memory when done.

Here is an example of allocating memory for pointer arrays in User.p:

```
{User global variables go here.}
const
 MyMaxCoordinates = 5000;

type
 CoordType = packed array[1..MyMaxCoordinates] of real;
 CoordPtr = ^CoordType;

var
 xCoordinate, yCoordinate, zCoordinate: CoordPtr;

procedure YourAllocationCode;
begin
 xCoordinate := CoordPtr(NewPtr(SizeOf(CoordType)));
 yCoordinate := CoordPtr(NewPtr(SizeOf(CoordType)));
 zCoordinate := CoordPtr(NewPtr(SizeOf(CoordType)));
 if (XCoordinate = nil) or (yCoordinate = nil) or (zCoordinate = nil) then begin
  DisposPtr(ptr(xCoordinate));
  DisposPtr(ptr(yCoordinate));
  DisposPtr(ptr(zCoordinate));
  PutMessage('Insufficient memory. Use get info and allocate more memory to Image');
 end;
end;
```

If you don't need the pointer anymore you can free memory using the DisposPtr call.

Operating on an Image

The global variables below relate directly to handling of images. The entire PicInfo record is not displayed. The actual record contains a number of other useful image parameters and can be seen in the globals.p file of the image project. Familiarity with the data structure is advisable to those who plan on modifying or operating on the image in any manner.

```
type
  PicInfo = record
    nlines, PixelsPerLine: integer;
    ImageSize: LongInt;
    BytesPerRow: integer;
    PicBaseAddr: ptr;
    PicBaseHandle: handle;
    ...... {many others covered, in part, in other sections}
  end;

  InfoPtr = ^PicInfo;

var
  Info: InfoPtr;
```

Using this global structure allows for the simple use of

```
with Info^ do begin
  DoSomethingWithImage;
end;
```

Getting at the bytes of an image

Any number of techniques can be used to access the image for use or modification purposes. Several techniques and examples are listed below. The choice for which to use largely depends upon the application at hand.

Pascal routines such as GetLine use the LineType. First look at the definition of LineType. LineType is globally declared as:

```
LineType = packed array[0..MaxLine] of UnsignedByte;
```

Naturally, UnsignedByte has been type defined as:

UnsignedByte = 0..255;

Pascal Technique one: Use Apple's "CopyBits" to wholesale copy a ROI, memory locations, or an entire image. Example's of CopyBits can be seen in the Image source code Paste procedure, some of the video capture routines and many others.

Pascal Technique two: Use ApplyTable to change pixels from their current value to pixels of another value. You fill the table with your function. The simple example below, which is extracted from DoArithmetic, would add a constant value to the image. The index of the table is the old pixel value and tmp is the new pixel value. With ApplyTable you don't have to work with a linear function like adding a constant. You basically can apply any function you like. Of course, you would want to always check and see if you are above 255 or below zero and truncate as needed. The actual ApplyTable procedure calls assembly coded routines in applying the function to the image.

Technique 2 example

```
procedure SimpleUseOfApplyTable;
  var
   table: LookupTable;
   i: integer;
   tmp: LongInt;
   Canceled: boolean;
begin
 constant := GetReal('Constant to add:', 25, Canceled);
   for i := 0 to 255 do begin
     tmp := round(i + constant);
      if tmp < 0 then
        tmp := 0;
      if tmp > 255 then
        tmp := 255;
     table[i] := tmp;
   end;
 ApplyTable(table);
end;
```

Aside from "doing arithmetic" such as adding and subtracting, the AppyTable routine is used by Image to apply the Look Up Table (LUT) to the image. Changing the LUT, such as by contrast enhancement or using the LUT tool, doesn't change the bytes of the image until the menu selection "Apply LUT" is selected from the Enhance menu.

Technique Three:

A: Use a procedure such as GetLine to move sequentially down lines of the image. You can access each line as an array. Compiled pascal is obviously much than a macro at doing this. In addition, your macro can call the faster compiled pascal code.

B: Use the Picture base address, offset to current location, and Apple's Blockmove to access individual lines of the image. Again, each line can be treated as an array allowing access to individual picture elements. Examples below.

First look at the definition of LineType. LineType is globally declared as:
        LineType = **packed array**[0..MaxLine] **of** UnsignedByte;

Naturally, UnsignedByte has been type defined as:
        UnsignedByte = 0..255;

For the technique 3 examples you can either:

1) Deal with the entire image and find it's width and height as:

```
with info^.PicRect do begin
  width := right - left;
  height := bottom - top;
  vstart := top;
  hstart := left;
 end;
```

2) Deal with just the ROI that you have created and use:

```
with Info^.RoiRect do begin
  width := right - left;
  RoiTop := top;
  RoiBottom := bottom;
  RoiLeft := left;
  RoiRight := right;
 end;
```

It is often useful to have your routine automatically define the entire image as the area which you will operate on. To automatically select the image you might do the following:

```
var
 AutoSelectAll: boolean;
begin
AutoSelectAll := not info^.RoiShowing;
if AutoSelectAll then
 SelectAll(false);
```

The false parameter is used to make an invisible ROI rather than the visible 'marching ants' typified by ROI selections. By first checking if an ROI exists, this code prevents overwrite of your specific ROI.

Technique 3A example

See specific examples in the procedure ExportAsText , DoInterpolatedScaling and others. See also the procedure GetLine.

```
procedure AnyOldProcedure;
 var
  width, hloc, vloc: integer;
  theLine: LineType;
begin
  with info^.RoiRect do begin
   width := right - left;
   for vloc := top to bottom - 1 do begin
    GetLine(left, vloc, width, theLine);
    for hloc := 0 to width - 1 do begin
     DoSomethingWithinTheLine   i.e. TheLine[hloc]
    end;
   end;
 end;
end;
```

Technique 3B example

This prolonged example will perform the same function as the 3a. It may or may not be easier for you to see how it functions, but should let you see how GetLine can do the job with a lot less programming. As usual some of the variables are seen in the globally declared PicInfo record.

```
procedure AnotherOldProcedure;
var
 OldLine,NewLine: LineType;
 SaveInfo: InfoPtr;
 p, dst: ptr;
 offset: LongInt;
 c,i: Integer;
begin
SaveInfo := Info;
with info^.PicRect do begin
  width := right - left;
  height := bottom - top;
  vstart := top;
  hstart := left;
 end;
if NewPicWindow('new window', width, height) then
 with SaveInfo^ do begin
  offset := LongInt(vstart) * BytesPerRow + hstart;
  p := ptr(ord4(PicBaseAddr) + offset);
  dst := Info^.PicBaseAddr;
  while i <= height do begin
     BlockMove(p, @OldLine, width);
     p := ptr(ord4(p) + BytesPerRow);
     while c <= Saveinfo^.pixelsperline do begin
       NewLine[c] := OldLine[c] {+ or -??-find a pixel and do what you want}
     end;
    BlockMove(@NewLine, dst, width);
    dst := ptr(ord4(dst) + width);
   end; {while i <= height}
 end; { with SaveInfo^}
end;
```

The 3b example is an oversimplification of the function duplicate in the image project. It usually is a good idea to first create a new window to move your information to. The NewPicWindow procedure can do this. The dst pointer can point into the new windows memory.

Working with two images

If you want to work with two images in pascal, using the data from one to effect the other image, you could set up something like the following code. You can easily work with two InfoPtr's to do the job. You might pass the picture number from a macro for convenience

```
SrcInfo := Info;
DestPic := Trunc(FinalImage);
Info := pointer(WindowPeek(PicWindow[DestPic])^.RefCon);
DstInfo := Info;{assign it to DstInfo}
 for vloc := RoiTop to RoiBottom - 1 do begin
   Info := SrcInfo;
   GetLine(RoiLeft, vloc, width, CurLinePtr^);

{Do something with the data and put the data to the other window}
   NewLinePtr^[hloc] := CurLinePtr^[hloc]*myfactor

   Info := DstInfo;
   PutLine(RoiLeft, vloc, width, NewLinePtr^);
```

Touching the 4th dimension

If you have multiple stacks of images which all relate to each other in some manner, you can load them all into memory for calculations. A program such as SpyGlass is useful for viewing this type of data, but it may not provide you with the means for calculating terribly much. If you wish to have a unique calculated value, or any type of value, for each point in each stack you could use Image and set something up like the below. Make sure you use Long integers for just about everything of the integer type. This routine should work with stacks of differing sizes loaded (i.e. one stack could be 200x200x5 and others might be 256x256x10 and so on).

```
{Set up multiple for loops for nPics and each SliceCount}
for PictureNumber := 1 to npics...
{You must find the previous data offset for the final array}
CurrentInfo := Info;
PreviousEndOfData := 0;
for i := 1 to PictureNumber - 1 do begin
 TempInfo := pointer(WindowPeek(PicWindow[i])^.RefCon);
 Info := TempInfo;
with Info^.PicRect do begin
   Previouswidth := right - left;
   Previousheight := bottom - top;
  end;
 if Info^.StackInfo <> nil then
   PreviousSliceCount := Info^.StackInfo^.nSlices
else
   PreviousSliceCount := 1;
 BytesUsed := PreviousSliceCount * PreviousWidth * PreviousHeight;
 PreviousEndOfData := PreviousEndOfData + BytesUsed;
end;
Info := CurrentInfo;
{Find how many slices in the current pic}
if Info^.StackInfo <> nil then
 SliceCount := Info^.StackInfo^.nSlices
else
 SliceCount := 1;
For SliceNumber := 1 to SliceCount ....
{Set up rest of the for loops here. The usual, up to hloc & vloc}
{put those here}
{Now compute a unique array offset}
ArrayOffset := PreviousEndOfData + (SliceNumber - 1) * LongInt(width) * height + LongInt(width) * longInt(vloc) +
LongInt(hloc);
```

```
{Finally store your calculation into a unique location}
MyHugeArray^[ArrayOffset] := SomeCalculatedValue;
```

Creating a dialog box

Get

function GetDNum (TheDialog: DialogPtr; item: integer): LongInt;
function GetDString (TheDialog: DialogPtr; item: integer): str255;
function GetDReal (TheDialog: DialogPtr; item: integer): extended;

Set

procedure SetDNum (TheDialog: DialogPtr; item: integer; n: LongInt);
procedure SetDReal (TheDialog: DialogPtr; item: integer; n: extended; fwidth: integer);
procedure SetDString (TheDialog: DialogPtr; item: integer; str: str255);
procedure SetDialogItem (TheDialog: DialogPtr; item, value: integer);

Dialogs are a good way to handle user I/O. If you can't get by with the set of dialogs in Image you could add one of your own. They can be used to set parameters or give options to the user. Several example dialogs in Image are the preferences dialog box and the SaveAs dialog. The template for dialog boxes are in the Image.rsrc file under DLOG and DITL. The DITL resource is for creation of each dialog item in the DLOG. Naturally, each item in the dialog template has a reference integer value associated with it. This allows you to keep track of what you are pressing or which box you are entering information into.

To handle the dialog to user I/O, you need to have a tight loop checking what is being pressed or entered. If the user is entering a number or string you need to retrieve it with one of the GET dialog functions. Likewise, you can pass information or turn off a button with the SET procedures. The basic form for a dialog loop appears below:

```
mylog := GetNewDialog(130, nil, pointer(-1)); {retrieve the dialog box}
Do default SET's here
OutlineButton(MyLog, ok, 16);
repeat
  ModalDialog(nil, item);
  if item = SomeDialogItemID then begin
    Get or Set something
... lots of if statements to check which item is pressed
until (item = ok) or (item = cancel);
DisposDialog(mylog);
```

Setting up alternative menu list from pascal

If you choose to not use the macro UserCode call but still don't plan on figuring out how events like MouseDown's and the rest of Mac programming works, the User.p module provides a method to add a routine and menu item fairly fast and simply to the Image program. By uncommenting the InitUser command, which is in **Image.p** at the very very bottom, you can add the User menu to the recompiled Image.

```
begin
Init;
...
LoadDefaultMacros;
UnloadSeg(@Init);
{InitUser;}              <=======Uncomment this line for User.p
```

The simple user menu is added to the right of the other Image menus

If you wish to, and with a little work, you can change this menu to reflect your command names by using ResEdit. Caution is always advised for a new ResEdit user. The Image project comes with an Image.rsrc file. This file contains the 'menu' definitions which you can change to your liking.

To execute your routine from the menu selection you will need to either replace the demonstration UserCommand procedures or simply change the menu selection code to the name of your procedure and recompile.

```
procedure DoUserMenuEvent (MenuItem: integer);
begin
 case MenuItem of
 1:
  DoUserCommand1;   <====Change to your procedure name
 2:
  DoUserCommand1;
 end;
end;
```

If you plan on more than two menu items, you will need to use ResEdit and change the menu as well as the add another MenuItem case in the above procedure.

It isn't terribly difficult to write routines into any of the available units, or even to use a menu other than the user menu. Some of the units are fairly large and it isn't terribly practical for debugging purposes to add much more to them. You will, of course, have to use other units if you are developing routines that are essentially local to that portion of the code. For example you would probably want to keep additional video routines in the camera.p unit.

Key & mouse

```
function OptionKeyDown: boolean;
function ShiftKeyDown: boolean;
function ControlKeyDown: boolean;
function SpaceBarDown: boolean;
```

It is fairly common for a menu selection to have several possible paths to follow. The selection process can be dictated via use of simple boolean functions. For the most part they are self explanatory. Holding the option key down when selecting a menu item is the most common way to select a divergent path. Your routine need only execute the function to test the key status.

```
if OptionKeyDown then begin
  DoSomething;
 end
else begin
  DoSomeThingElse;
 end;
```

CommandPeriod

```
function CommandPeriod: boolean;
```

The CommandPeriod function is used when you want to interrupt execution of a procedure. For example you might include the following bit of code in a prolonged looping routine that you write:

```
if CommandPeriod then begin
   beep;
   exit(YourLoopingProcedure)
 end;
```

Mouse button

Apple has supplied several mouse button routines such as the true or false button boolean. It's functionality is the same as in the macro language.

```
Function Button:boolean;
```

The button functions are explained in Inside Mac

Image and text

There are a number of ways to handle text with Image. If you are working in the context of macros, then a text window should handle most of what you want to do. Copy and paste functions work with the text window. Sample macros, such as the example under SelectPic and Selectwindow in the macros section above, show how to handle the majority of text data handling needs.

If your needs are larger, or if you are considering extensive data to disk handling, then you should consider using the textbuffer pascal routines described below. You can use these routines to export as text all the data you can possibly fill memory with. These are NOT connected with the text window routines, which are seperately seen in the Text.p file.

Global declarations

```
const
 MaxTextBufSize = 32700;
type
 TextBufType = packed array[1..MaxTextBufSize] of char;
 TextBufPtr = ^TextBufType;
var
 TextBufP: TextBufPtr;
 TextBufSize, TextBufColumn, TextBufLineCount: integer;
```

Other useful definitions include:

```
 cr := chr(13);
 tab := chr(9);
 BackSpace := chr(8);
 eof := chr(4);
```

Dynamic memory allocation for the textbuffer (under Init.p) sets up a non-relocatable block of memory.

```
 TextBufP := TextBufPtr(NewPtr(Sizeof(TextBufType)));
```

To clear the buffer set TextBufSize equal to zero. Use TextBufSize to keep track of what data within the textbuffer is valid. Anything beyond the length of TextBufSize is not useful. Many Apple routines, such as FSWrite, require the number of bytes be passed as a parameter.

Text buffer utilities

Some of the utilities associated with the textbuffer include:

```
procedure PutChar (c: char);
procedure PutTab;
procedure PutString (str: str255);
procedure PutReal (n: extended; width, fwidth: integer);
procedure PutLong (n: LongInt; FieldWidth: integer);
```

Expansion of PutString may help in the understanding of the functionality involved:

```
procedure PutString (str: str255);
 var
  i: integer;
 begin
 for i := 1 to length(str) do begin
   if TextBufSize < MaxTextBufSize then
    TextBufSize := TextBufSize + 1;
    TextBufP^[TextBufSize] := str[i];
    TextBufColumn := TextBufColumn + 1;
  end;
 end;
```

An example call sequence which places text into textbuffer might look something like:

```
PutSting('Number of Pixels');
PutTab;
PutString('Area');
putChar(cr);
```

To Save the textbuffer, the procedure SaveAsText can be used after a SFPPutfile to FSWrite data to the disk or other output.

Saving a text buffer

To Save the textbuffer, the procedure SaveAsText can be used after a SFPutfile. SaveAsText will FSWrite data to the disk. SFPutfile shows the standard file dialog box and FSWrite (within SaveAsText) does the actually saving to disk.

```
procedure SampleSaveBuffer;
 var
  Where: point;
  reply: SFReply;
 begin
 SFPutFile(Where, 'Save as?', 'Buffer data', nil, reply);
 if not reply.good then
  exit(SampleSaveBuffer);
 with reply do
  SaveAsText(fname, vRefNum);    {this will handle the FSWriting}
 end;
```

Image Engineering

Fundamentals of densitometry

It is possible to use a scaling system for pixels which has a one to one correspondence to the concentration of what you are studying. Sample concentrations can be determined using optical, electronic, and most importantly for our purposes, a computer based imaging technique. Densitometric science was described originally by Bouguer and Lambert who described loss of radiation (or light) in passing through a medium. Later, Beer found that the radiation loss in a media was a function of the substance's molarity or concentration. According to Beer's law, concentration is proportional to optical density (OD). The logarithmic optical density scale, and net integral of density values for an object in an image is the proper measure for use in quantitation. By Beer's law, the density of a point is the log ratio of incident light upon it and transmitted light through it.

$$OD = Log_{10}(\frac{I_o}{T})$$

There are several standard methods used to find the density of an object or a point on an image. Scanning densitometers have controlled or known illumination levels, then measure transmitted light through an object such as a photographic negative. Since both the incident and transmitted light are known quantities, the device can then compute this ratio directly. This is also the case of those who use a flat field imaging technique and capture two seperate images. The first image is of an empty light box and the second is of the specimen to be evaluated. These two can then be used in computing a log ratio.

In the case of a video camera/frame grabber combination,

using a non flat field technique, several things are of note. With a camera, you do not measure OD values directly. The camera and frame grabber pixel values are linear with respect to Transmission (T), which is the anti-log of the negative of OD:

$$T = 10^{-O.D.}$$

or:

$$O.D. = -\log_{10}(T) = \log_{10}(\tfrac{1}{T})$$

Since this is often a source of confusion among those designing systems for densitometry you should again note that the camera does not measures T, nor does it measure OD. Camera systems, CCD's, and any frame grabber conversion values (pixel values) have been designed so that they are linear with respect to T. It isn't meaningful to take the minus log of the pixel value since these are not T values.

Nevertheless, you want to do densitometry and need a scale (not pixel values) which correlates to concentration or OD. Further, it may not be convienient to measure the incident light and do a log ratio. Fortunately, you can use an external standard, such as an OD step tablet or a set of protein standards on a gel. NIH Image has the built in "Calibrate..." command to allow you to transform pixel values directly from a scale which is linear with respect to T and into a scale which correlates to OD or concentration. The calibrate command, used with standards, is best done with an exponential, rodbard or other fit since the relationship of OD to T is not a simple linear (y=mx+b) relationship (see equation above) and because the camera may not be perfectly linear with respect to T over the range of density

values you use as standards. In other words you have both created a LUT of OD values for each linear to T pixel value and you help compensate for slight nonlinearities of the camera.

A sample calibration curve fit:

$$y=c*((a-x)/(x-d))^{(1/b)}$$

a=4.31879912
b=1.10090456
C=0.33031497
d=302.0483045

S.D.=0.5033
R^2=0.9999

(1.51 ... 0.00, 0 ... 255)
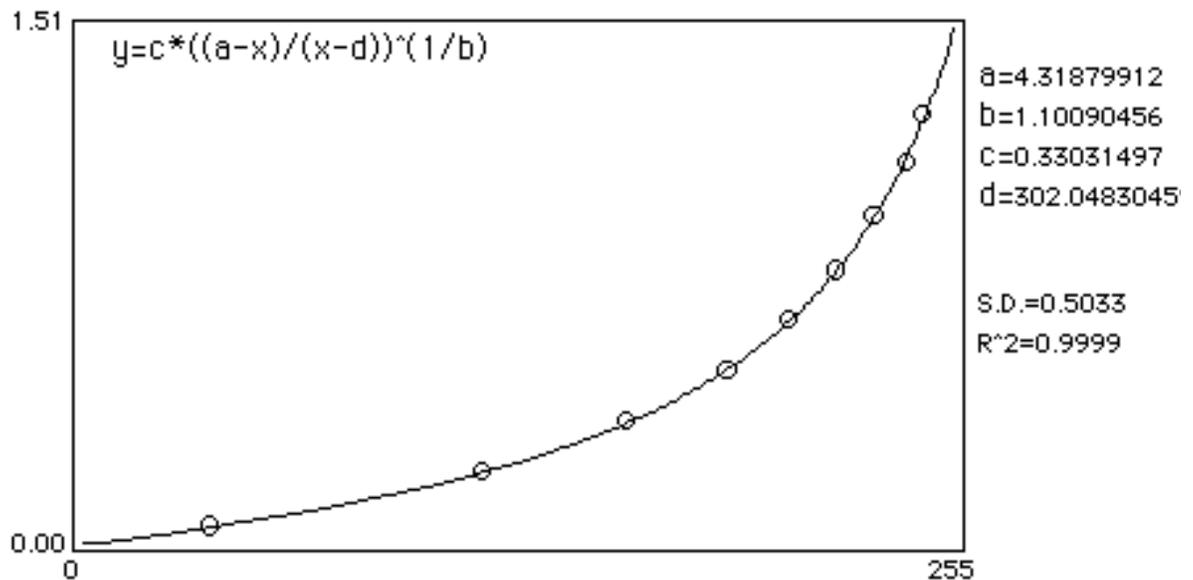
There are several other points of note which you should adhere to in performing your densitometric analysis. Your standards should always exceend the range of data which you want to image and perform density measurements on. You should not use curve fit data (or the pixel values) which extend beyond the last, or before the first calibration point. Additionally, there is a point at which the camera can no longer produce meaningful output when additional light is input (saturation). You could also have a low light level condition where the camera or CCD can not produce a measureable output (under-exposure). You will notice that these data points do not fit well into an end of the curve, or could basically ruin the fit of the data. You should remove these points from your calibration data and not use the density values for these pixels in your measurements.

References:

Kodak Corporation, KODAK Neutral Density Attenuators, Kodak publication no P-114, Photographic Products Group, 1982

Kodak Corporation, Scientific Imaging with KODAK Films and Plates, Eastman Kodak publication no. P-315, 1987

Webster JG, Medical Instrumentation, Application and Design, Boston, Houghton Mifflin Company, 1978:95,518-9.

Textbooks of Quantitative Chemistry

Camera SNR and significant bits

Camera signal to noise ratio is defined as the peak to peak signal output current to root mean square noise in the output current. Although this sounds confusing, the value describes whether the camera is one which can give you 256 significant gray levels or not. A formula to convert camera SNR into number of significant digits is very useful. The formula to do this is defined as :

$$SNR\ (dB) = 6.02n + 1.8$$

Where n will be the number of significant bits. You will need 50 or higher dB for the potential to convert a signal to 8 bits. In practice you will need a better camera than this to get 8 bits since it is likely most conditions for imaging are not ideal.

In order to get a true 8 bits, you will also need to capture the video using a frame grabber that does not introduce error. A frame grabber which has an analog to digital converter (ADC) with less than one half bit of differential non-linearity, <u>specified at the video bandwidth</u>, is needed to capture 8 significant bits.

References:

IEEE Standard Dictionary of Electrical and Electronic Terms

Analog-digital Conversion Handbook, Staff of Analog Devices, Edited by Daniel Sheingold, Prentice Hall, 1986

Designing your own optimal threshold

If you do not like the automatic threshold which NIH Image picks for your data, you can implement alternative thresholding with a macro. If you want a consistently picked threshold point based on specific image parameters you might try the following technique. Find an appropriate threshold by trying a multiple of the number of standard of deviations past the mean. Running the macro below will help you do this. Once you are satisfied with the right multiple (can be decimal), code the number into your macro and execute it on all the images you want to analyze. If you have a border or other artifact which is not in all images, be sure to exclude this by using a ROI.

```
Macro 'Std Dev Threshold [T]';
VAR
  Count, Threshold:integer;
  StdDev,TheMean,MultFactor:real
BEGIN
 ResetGrayMap;
{Pick a multiplication that works for your dataset}
 MultFactor := GetNumber('StdDevs past the mean to threshold?',1.0);
{
Hardcode the number in the macro when you find a good value
that seems to work well with your dataset
}
{MultFactor := 1.0;}
 ResetCounter;
 Measure;
 StdDev := rStdDev[rCount];
 TheMean := rMean[rCount];
{Apply the multiplication and threshold}
 Threshold := TheMean+ round(MultFactor*StdDev);
 SetThreshold(Threshold);
 Showmessage('Threshold level =',Threshold,'\\Using ',MultFactor:4:2, ' standard of deviations','\from the mean');
END;
```

IR & Coomassie blue filters

There are filters you can place in front of a camera lens to give complete rejection of the IR (>700nm) wavelengths. This is often crucial in video acquisition using CCD cameras. Filters which show 0% transmittance above 700 to 800 nm are available and recommended over those which show 10% or more transmission at these wavelengths. There are also several filters that compensate for the poor video response in the wavelengths associated with Coomassie blue stains. The graph below represents a pass of Coomassie brilliant blue in typical solution through a spectrophotometer. The peak absorbance is about 585 nm.
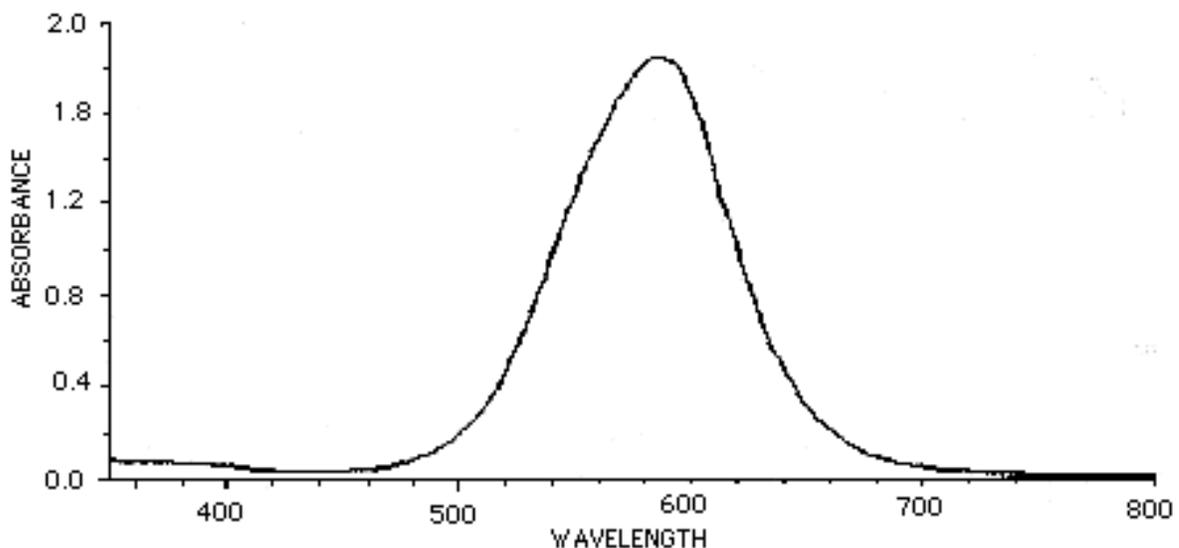
Corian Corporation sells filters filters at 630 nm and 560 nm, both of which produce excellent results when imaging Coomassie blue. This is true when sufficient lighting exists. Corian also sells a complete IR suppressing filter for about $200. Filter holders can be purchased at photography stores or fabricated. Tiffen sells 52 and 62 mm ring attachments which fit on typical camera lenses. These are also available at photography stores.

Corian Corp

1-508-429-5065
73 Jeffrey Ave.
Holliston, MA 01746-2082

FR-400-S Complete IR surpressing filter
P10-630-R and P10-560-R, 630nm and 560nm filters

Device drivers (framegrabbers)

The Following global variables relate to framegrabber support.

```
var
 FrameGrabber: (QuickCapture, Scion, ScionLG3, NoFrameGrabber);
 fgSlotBase: LongInt;
 ControlReg, ChannelReg, BufferReg, DacHighReg, DacLowReg: ptr;
 Digitizing: boolean;
 InvertVideo, HighlightSaturatedPixels: boolean;
 VideoChannel: integer;

 FramesToAverage: integer;
```

When you run Image, the software executes the LookForFrameGrabbers procedure (seen in Init.p). The LookForFrameGrabbers routine executes GetSlotBase. GetSlotBase will read the vendor id's from boards residing in the NuBus of your Mac. If any board matches the Data Translation or Scion id, then GetSlotBase will return the base memory mapped address for the board.

function GetSlotBase (id: integer): LongInt;
 {Returns the slot base address of the NuBus card with the specified id. The address}
 {returned is in the form $Fss00000, which is valid in both 24 and 32-bit modes.}
 {Returns 0 if a card with the given id is not found.}

Within the QuickCapture board are several registers to control operations. These are offset from the boards base address by $80000 and $80004.

The highest priority loop in image controls acquisition from the QuickCapture board. The Digitizing boolean becomes true when you start capturing from the menu item.

```
        if Digitizing then begin
          CaptureAndDisplayFrame;
```

To set the QuickCapture going, you need to set bit 7 on the control register ($Fss80000)

```
        procedure GetFrame;

          ControlReg^ := BitAnd($80, 255); {Start frame capture}
```

and then use CopyBits to copy to the video memory.

Useful handheld debounced pushbutton

The circuit below is a useful handheld external trigger for frame grabbers or other devices. It can be adapted in a number of ways to debounce the signal from an instrument to the quickcapture or other frame grabber. The top half is for battery power, the bottom half does the debouncing.

Index