INSIDE NIH IMAGE

(NIH Image 1.60)

GENERAL INFORMATION

About this Document

This document is intended for beginners, who never formally studied computer science towards a degree, but need to use software and get the most out of it. For those who would like to correct errors in this guide please contact Mark Vivino in the NIH Division of Computer Research and Technology. My email is mvivino@helix.nih.gov. If you are like me, spending your whole day writing a graphical user interface is not your way of make a contribution to the world. You might 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). 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). Having freely available and modifiable source code is not seen often with most commercial packages. This guide updated 4/10/96, current to Image version 1.60

Macro Examples, Techniques & Operations

What is a macro and why write one?

A macro is text containing a sequence of calls or routines which NIH Image interprets and executes. To write a macro, you can choose "New" then "text window" to create a text window within NIH Image. You load the macro using "Load Macro". A rich set of example macro routines is distributed with the NIH Image program. You can try some of these out and borrow code from them in order to write your own macro.

Simple macros, such as the one below, are useful utilities to save time and effort. This macro is an example of a macro which follows the same operations that could be manually performed by you from the NIH Image menus. It clears anything outside of the Region of Interest (ROI) which you draw. Macros can, of course, be much larger and can include looping, calculations and basically an entire imaging application.

```
MACRO 'Clear Outside [C]';
  {Erase region outside ROI.}
BEGIN
  Copy;
   SelectAll;
  Clear;
   RestoreRoi;
   Paste;
   KillRoi;
END;
```

As a general guidline, if you have a highly iterative operation, prolonged calculation, derivation, modification or anything else 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.

Before you begin

It should not be hard for you to start writing a macro. You will want to do several things before you begin. First, go to the "NIH Image 1.xx Manual" file and print the section “Macro Programming Language “. This provides you with a complete list of all macro calls. The list is organized by the NIH Image menus, or the call is categorized as a miscellaneous call or as a miscellaneous function. After you print this section, be sure you understand the organization of the manual by looking at the NIH Image menus and examining the list of calls in the printout. Finally, locate the macros folder distributed with NIH Image. Open, load and examine some of the macros. Try using "Find" from the "Edit" menu on one of the open macros. "Find" is fairly useful in helping you debug a macro. It allows you to go to sources of error when you get error messages during the load or execution of a macro.

For the programming beginner

You probably don't need to study programming to write a macro. Depending on the complexity of your application, you might be able to pick up everything you need by examining some of the macros in the macros folder. To some, a confusing aspect of writing macros is understanding what a function is and how it is used. A function returns a value or a boolean (true/false). In the example below, nPics is a function which will return an integer number of pictures open. KeyDown('option') returns a true or false depending on whether you hold the option key down.

```
Macro 'Function demo';
begin
 {Here is an example use of the nPics function returning a value}
    showmessage('Number of images open: ',nPics);
 {Here is an example use of keydown function returning a boolean}
   If KeyDown('option') then putmessage('Number of images open: ',nPics);
end;
```

One text I recomend skimming through is Pascal Programming and Problem Solving, by Leestma and Nyhoff, other texts are listed in the "Macro Programming Section" of the NIH Image manual.

Macro global vs. local vars

Most programming languages like pascal, C, etc, 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; {global}

Macro 'Add numbers';
Var
   A,B: real; {local}
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;
```

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.

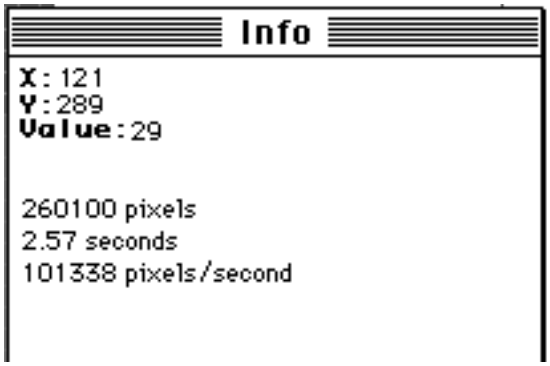
```
PutMessage('This macro requires a line selection');
```

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

```
PutMessage('There are ', nPics, '  open');
```

ShowMessage

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



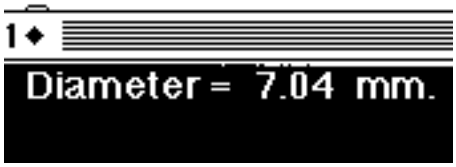
Here is a simple example of output to the Info window:

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

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

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

Write



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.');
```

Number of digits

There are several ways to set the number of digits you use for output with Showmessage, Write, Writeln, putmessage or displaying rUser arrays. One way is to use the SetPrecison macro call.

```
Macro 'Digits example one';
Var
   A,B, Answer: real;
begin
    SetPrecision(4);
   A := 3.1415962;
  B := 10.0;
  Answer := A*B;
    ShowMessage('The result is: ', Answer);
end;
```

The answer shown in the Info window will have 4 digits after the decimal.

The other method is to use the form e:f1:f2 where f1 is the field width, and f2 specifies the number of digits to the right of the decimal point.

```
Macro 'Digits example two';
Var
   A,B, Answer: real;
begin
   A := 3.1415962;
  B := 10.0;
  Answer := A*B;
    ShowMessage('The result is: ', Answer:4:2);
end;
```

The answer shown in the Info window will have 2 digits after the decimal.

Switching and choosing windows

There are a number of ways to switch between windows in a macro. One of the best ways is to use the PidNumber function to identify a unique ID for that window you will be switching to at a later time during your macro execution. Pidnumber is a function which returns a value. For example you might have:

```
var
  MyPicID:integer;
begin
   MyPicID := PidNumber;
    Duplicate('Duplicate image');
{some process}
   SelectPic(MyPicID); {To go back to the original}
```

Here the returned value from the PidNumber function was assigned to a variable called MyPicID. The variable MyPicID was then used later on in the macro to select the picture.

As an alternative to SelectPic, you could have used ChoosePic(MyPicID). This would have selected the picture but would not have made it the active front window. This is useful when you flip between many windows, but do not need to activate the window.

As a second alternative, you could can use SelectWindow('Window name') to select the window by its title. But if the named window is closed or non-existent, your macro will end with the ensuing error.

How to input a number or string

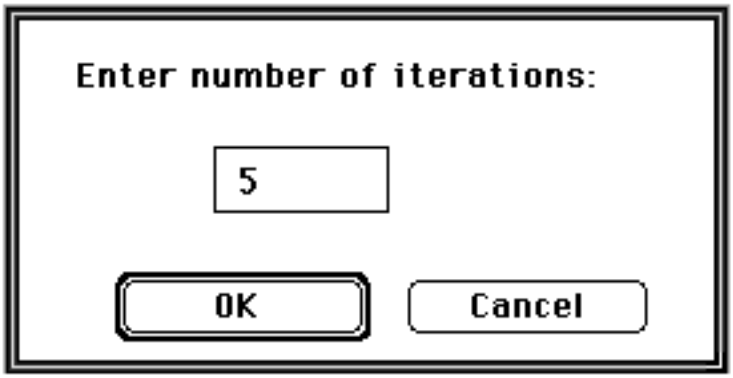
Making a call to getnumber will allow you to enter a number into your macro. The GetNumber macro will return a real number, or, if assigned to an integer variable it will not pass the decimal digits should they exist

GetNumber('Prompt', default, d)

Displays a dialog box and returns with the value entered. *Prompt* is the prompting string. *Default* is the default value. *d* (optional) is the number of digits to the right of the decimal point (default is 2). Set *d* to zero to display an integer default value.

This example displays, then reads in an integer

```
macro 'Number input example';
var
   MyNumber:integer;
begin
      MyNumber:=GetNumber('Enter number of iterations:',5,0);
    {some process}
end;
```



The idea is the same for entering a string

```
macro 'String input';
var
   MyString:string;
begin
      MyString:=GetString('What name?','Data');
end;
```

Looping

The NIH Image macro language has the standard set of pascal loops. This includes "for" loops and "while" loops, etc.

For loop:

```
Macro 'For loop example';
Var
    i:integer;
begin
   for i := 1 to 10 do begin
       ShowMessage('This iteration is: ', i);
       wait(0.5); {just a delay to see the answer in the Info window}
     {some process}
  end;
end;
```

While loop:

```
Macro 'While loop example';
Var
    i,MyNumber:integer;
begin
    i:=1; {start the loop at 1}
     MyNumber:=GetNumber('Enter end of the loop:',10,0);
    while i<=MyNumber do begin
       {some process}
         ShowMessage('This iteration is: ', i);
         wait(0.5); {just a delay to see the answer in the Info window}
       i:=i+1;
   end;
end;
```

Loop with step:

The NIH Image macro language is (almost) a subset of Pascal. The macro FOR statement does not have a BY option. Instead, use a WHILE loop with appropriate increment.

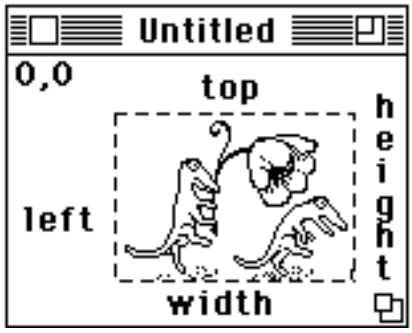
```
Macro 'Loop with step';
Var
    i,MyNumber, step:integer;
begin
    i:=0; {start the loop at 0}
      MyNumber:=GetNumber('Enter ending value:',100,0);
      Step:=GetNumber('Enter step:',10,0);
    while i<=MyNumber do begin
      {some process}
        ShowMessage('i value: ', i);
         wait(0.5); {just a delay to see the answer in the Info window}
       i:=i+step;
   end;
end;
```

Repeat Loop:

```
Macro 'Repeat loop';
Var
   i:integer;
begin
  i := 0;
   Repeat
      {some process}
        ShowMessage('This iteration is: ', i);
        wait(0.5); {just a delay to see the answer in the Info window}
      i := i +1;
     until Button;
end;
```

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. Coordinates of an image are x,y with 0,0 in the top left of the image. “top” will be the distance from the top in y. “left” will be the distance in x from the left edge. “height” is in y, and “width” in x.



This macro will list your ROI’s coordinates.

```
Macro 'Coordinates';
Var
    left, top,width,height:integer;
begin
    GetRoi(left,top,width,height);
  If width = 0 then begin
      putmessage('no ROI exists');
     exit;
  end;
  ShowMessage('ROI is ',left,' x units from the left','\',
                    width, ' x units in width','\',
                    top, ' y units below the top edge','\',
                   height, ' y units in height');
end;
```

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.

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 press of option, shift and control keys

The macro "KeyDown(key)" (Key = 'option', 'shift', or 'control') returns a boolean true or false. It returns TRUE if the specified key is down. The example macro below can be run on any stack, using shift to delay more or control to delay less.

```
macro 'Animate Stack';
var
    i,delay:integer;
begin
    RequiresVersion(1.56);
  i:=0;
   delay:=0.1;
   repeat
    i:=i+1;
     if i>nSlices then i:=1;
     Wait(delay);
     SelectSlice(i);
   if KeyDown('shift') then delay:=1.5*delay;
     if delay>1 then delay:=1;
   if KeyDown('control') then delay:=0.66*delay;
   if KeyDown('option') then beep;
      ShowMessage('delay=',delay:4:2);
   until button;
end;
```

Measurement and rUser Arrays

Currently you can not declare your own arrays but you can store macro data and results in what is called the rUser arrays.There are actually many arrays available to the macros, but of only several types:

1) Measurement, but user configuarable, rUser arrays.
2) NIH Image measurement results arrays
3) Built in NIH Image arrays
4) LineBuffer array (image data array)

Here are the specific examples of these

1) rUser1, rUser2
2) rArea, rMean, rStdDev, rX, rY, rMin, rMax, rLength, rMajor, rMinor, and rAngle.
3) Histogram, RedLUT, GreenLUT, BlueLUT, xCoordinates, yCoordinates, Scion, PlotData
4) LineBuffer

You may write to the rUser arrays simply as:

```
rUser1[1]:=SomeNumber;
rUser2[1]:=SomeOtherNumber;
```

And similarly retrieve as:

```
SomeNumber := rUser1[1];
SomeOtherNumber := rUser2[1];
```

If you have more than two sets of data which you'd like to keep, and because there are only two rUser arrays, then you can access other macro arrays. However you will need to be careful because these arrays are affected by the Measurement command and the index of the counter (rCount). You could easily write over your data without knowing. An example use of measurement arrays outside the intended use 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.

rCount, SetCounter, ResetCounter

The index of the measurement counter is stored in rCount. The index could also be described as the last value seen in the index column of the results window (Show Results). Run this macro to see measurements and the counter (rCount) value.

```
Macro 'rCount explaned';
Var
    i:integer;
begin
   ResetCounter;
    SetOptions('Mean');
   MakeRoi(0,0,5,5);
   for i := 1 to 10 do begin
     Measure;
     MoveRoi(5,0);
  end;
   ShowResults;
   ShowMessage('The final index in the results window is the value of rCount','\\',
     'rCount value is: ', rCount);
end;
```

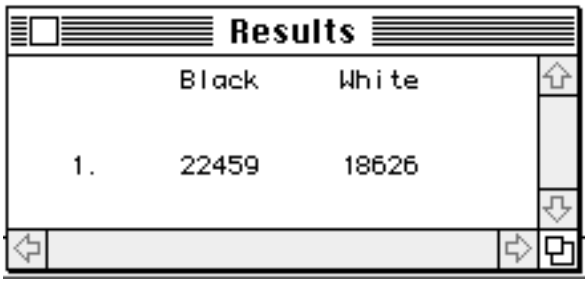
Issuing the ResetCounter command will allow overwriting of all previous measurement data. You can also use the SetCounter command when you want to display a set of your own data which is not dependant upon issuing the measure command. An example would be:

```
Macro 'SetCounter example';
Var
     left,top,width,height:integer;
begin
   ResetCounter;
    SetOptions('rUser1');
    SetPrecision(3);
   SelectAll;
    GetROI(left,top,width, height);
  rUser1[1] := width;
  rUser1[2] := height;
   rUser1[3] := height/width;
   SetCounter(3);
  ShowResults;
end;
```

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
    SetUser1Label('Black');
    SetUser2Label('White');
  Measure;
    rUser1[rCount]:=histogram[255];
    rUser2[rCount]:=histogram[0];
  ShowResults;
end;
```



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');
```

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.

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;
```

Batch Processing

It's easy to write a macro to process a series of images in a folder as long as the file names contain a numerical sequence such as 'file01.pic', 'file02.pic', 'file03.pic', or image 001, image 002, ...

```
macro 'Batch Processing Example';
{
Reads from disk and processes a set of images too large to
simultaneously fit in memory. The image names names must be
in the form 'image001', 'image002', ..., but this can be changed.
}
var
  i:integer;
begin
  for i:=1 to 1000 do begin
      open('image',i:3);
     {process;}
      save;
      close;
   end;
end;
```

Avoiding a macro dialog box

From wayne@helix.nih.gov (Wayne Rasband) reply on nih-image@soils.umn.edu

You should be able to process many files and only have to see one dialog box. For example, only one dialog appears when you run the following macro as long as 'A', 'B' and 'C' are in the same folder.

```
macro 'test';
begin
    Open('A');
   Invert;
   Save;
   Close;
  Open('B');
```

Another way to avoid the dialog box is to use full directory paths as in the following example.

```
macro 'test';
begin
  Open('hd400:images:A');
  Invert;
  Save;
  Close;
  Open('hd400:images:B');
  Invert;
  Save;
```

```
  Close;
  Open('hd400:images:C');
  Invert;
  Save;
  Close;
end;
```

In V1.55, you can use a full folder path [.e.g., SaveAs('HD400:My Images:mage001')] and the dialog box will not be displayed.

TickCount

From wayne@helix.nih.gov (Wayne Rasband) reply on nih-image@soils.umn.edu

According to "Inside Macintosh", ticks are counted at the rate of 60 per second. You can varify this by running the enclosed macro and timing the interval between beeps.

```
macro 'TickCount Test';
{"Beeps" every 10 seconds}
  var
       interval,ticks:integer;
begin
   interval:=600;
    ticks:=TickCount+interval;
   repeat
     if TickCount>=ticks then begin
      beep;
        ticks:=ticks+interval;
    end;
   until button;
end;
```

Placing time and date into your data

If you desire date/time in your results you can create a separate text window which will include date and time. You can copy other results (from Show Results) to this window afterwards.

```
macro 'Date and time to window';
var
  year,month,day,hour,minute,second,DayOfWeek:integer;
begin
  GetTime(year,month,day,hour,minute,second,DayOfWeek);
  NewTextWindow('Data measurements',500,600);
   SetFont('Monaco');
  SetFontSize(12);
   Writeln('Data Analysed - ',month:2,'/',day:2,'/'year-1900:2,' at '
       ,hour:2,':',minute:2);
  Writeln('');
end;
```

PlotData notes

From reply of jy@nhm.ic.ac.uk on nih-image@soils.umn.edu

>Does anyone know of an easy way to get the actual points in x,y coordinates and
>the values at each point from the profile plot data using macros?

Image 1.54 introduced a new command to +/- allow this:

"A command was added to the macro language for making profile plot data available to macro routines. It has the form "GetPlotData(count,ppv,min,max)", where count is the number of values, ppv is the number of pixels averaged for each value, and min and max are the minimum and maximum values. The plot data values are returned in a built-in real array named PlotData, which uses indexes in the range 0-4095. The macro "Plot Profile" in "Plotting Macros" illustrates how to use GetPlotData and PlotData."

[from the changes file]

To help answer your question further....

1. For a count value of n the PlotData array will have meaningful values from 0 to n-1 (higher array values are accessible but will contain old/meaningless results).

2. Count is equal to the line length, in pixels, rounded to the nearest integer value. But...

3. Substantially more pixels are usually highlighted by a line selection, and this seems to have only an approximate corelation with the pixels used by PlotData.

4 The PlotData array contains real-numbers (not integers) which presumably are derived from a weighted average of pixels rather than being the values of single pixels - even when ppv is 1. Because of this it is not possible to relate PlotData values to single locations.

5. My conclusion after some experimentation is that;

```
after   GetLine(x1,y1,x2,y2,lw);
and     GetPlotData(count,ppv,min,max);
```

The following function will probably return the centre of the location used to derive PlotData[c]:

```
ypos:=y1+(c+0.5)/(count)*(y2-y1);
xpos:=x1+(c+0.5)/(count)*(x2-x1);
```

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 **Info** 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;
```

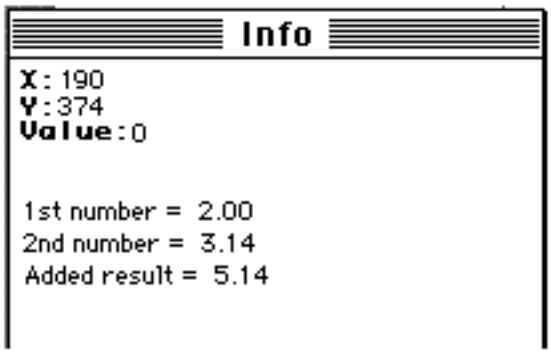
Step 3:

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.



Pascal Examples, Techniques & Operations

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 recommend 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('Number of Objects:  ', Long2Str(ObjectCount)));
```

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 cast as a string into the Info window.

```
ShowMessage('No  QuickTime');
```

or when more involved you must convert reals or integers into strings before issuing the command:

```
str1 := concat('min=', long2str(CurrentMin), ' (', long2str(AbsoluteMin), ')', crStr,
'max=', long2str(CurrentMax), ' (', long2str(AbsoluteMax), ')');

ScaleFactor := 253.0 / (CurrentMax - CurrentMin);

RealToString(ScaleFactor, 1, 4, str2);

ShowMessage(concat(str1, crStr, 'scale factor= ', str2));
```

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:',5,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. You have to add File2 (File2.p contains GetTextFile) to the Uses clause at the beginning of User.p. Note that this routine has been changed for version of Image 1.54 and above.

```
procedure OpenData;
 var
   fname: str255;
   RefNum, nValues, i: integer;
   rLine: RealLine;
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^);
```

4D dataset

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);
```

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.

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;
```