INSIDE NIH IMAGE

(NIH Image 1.55)

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 301-402-2633. 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'm NOT a programmer but have 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 4/21/94, current to Image version 1.55. 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. I have tried to list the contributor of text from the NIH Image mailing list where appropriate in this manual.

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.

Reference card

In the macros folder, distributed with NIH Image, is a file called "reference card". This text file is very useful to have open while programming a macro.

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 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.

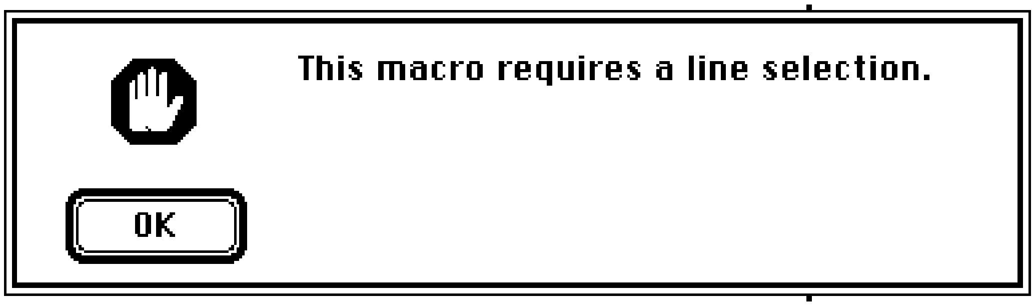
Using SelectWindow & SelectPic

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


This macro requires a line selection.

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.

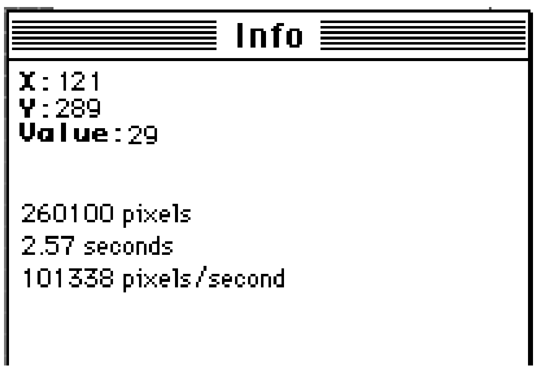
```
PutMessage('This macro requires a line selection');
```

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 Info window.



```
X: 121
Y: 289
Value: 29


260100 pixels
2.57 seconds
101338 pixels/second
```

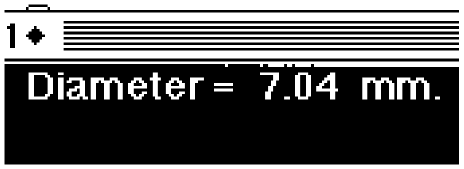
Here is a simple example of output to the Info window:

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

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

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

Write



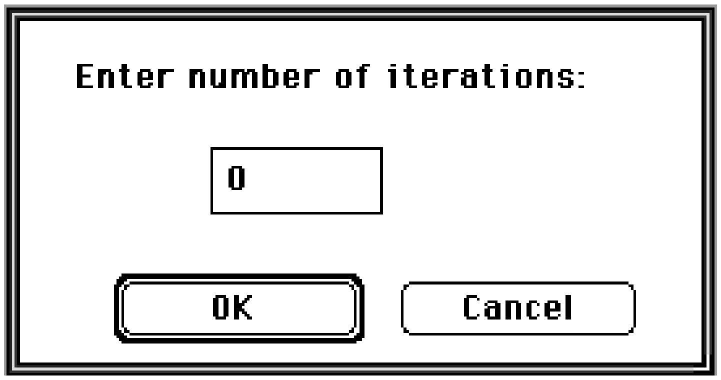
You can also write data or info onto the image window with a macro call to Write or Writeln.

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

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, 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;
```



The idea is the same for entering a string

```
var
 MyString:string;

macro 'String input';
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. Although close to pascal, the macro language doesn't have everything as this email shows:

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

>in the "for" command (as in for i= 1 to fred Do) is there a skip command.
>For example, can I choose to do:
>   for i = 1 to fred by 10 DO

The NIH Image macro language is (almost) a subset of Pascal and the Pascal
FOR statement does not have a BY option. Instead, use a WHILE loop. For
example:

```
  i:=1;
  while i<=fred do begin
     {process}
     i:=i+10;
  end;
```

Measurement and rUser Arrays

There are a number of arrays in macros, but there are two varieties; the measurement arrays and the rUser arrays. You can store macro data and results in the rUser arrays. These arrays are not affected by the Measurement counter (rCount) which works with measurements arrays such as rMean[rCount], rArea, etc. The current rCount for these is changed by doing a measurement or calling SetCounter.

Example of storing data to the rUser arrays:

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

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. This includes rArea, rMean, rStdDev, rX, rY, rMin, rMax, rLength, rMajor, rMinor, and rAngle. However you will need to be careful because these arrays are affected by the rCount value and you could write over your data. An example use of measurement arrays outside the intended use is a snipet of code from the Export look up table macro:

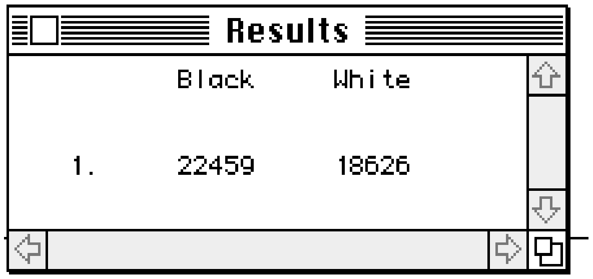
```
for i:=0 to 255 do begin
   rArea[i+1]:=RedLut[i];
   rMean[i+1]:=GreenLut[i];
   rLength[i+1]:=BlueLut[i];
 end;
```

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

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;
```



```
        Black     White
1.      22459     18626
```

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');
```

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);
```

Zero fill and Blank Fill

From reply of chien@jeeves.ucsd.edu (Chi-Bin Chien) on nih-image@soils.umn.edu

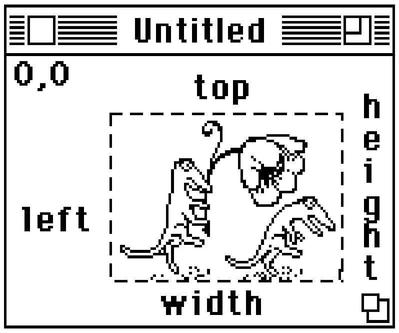
> is there a general rule to understand the arguments description for
> certain commands in the Macro Reference Card ('accepts multiple arguments
> zero fill' and 'accepts mult. arguments blank fill')? I could find them
> not in the 'About NIH-Image' document, only in the RefCard.

You may notice that the descriptions "zero fill" and "blank fill" only apply
to commands that accept strings as arguments, for instance SaveAs(), which is
zero fill, and ShowMessage(), which is blank fill. These commands take multiple
arguments, which will be concatenated together into one long string, then used
as a filename or printed or whatever. "zero fill" and "blank fill" refer to
what is to be done with numerical arguments such as 'n:3'. For text, you want
the number to be padded out to the specified length with leading spaces ("blank
fill"): ' 1', ' 2', ' 11', etc. For filenames, you usually don't want spaces,
so the numbers are padded with leading zeroes instead: '001', '002', '011'. So
in general, commands that take filenames will be zero fill, and

commands that
take strings for display, etc. will be blank fill.

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.

Batch Processing

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

>is there a possiblility to define 'open' access to the file contents of a
>folder (with, lets say, Images of 2.5 MB size each)? I want to do a Batch list
>for Background subtraction and contrast enhancement.

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', etc. I have included an example macro that does
this.

```
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;
```

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

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:

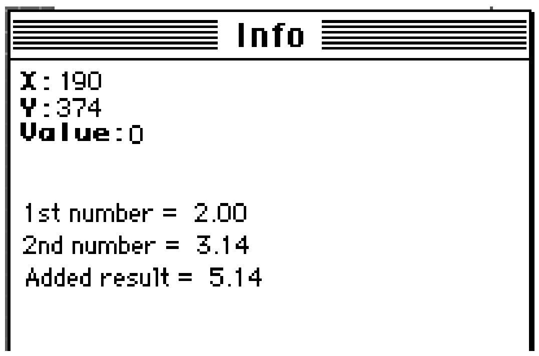
```
LutArray = packed array[0..255] of byte;

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 p**Info**



```
X: 190
Y: 374
Value: 0


1st number =  2.00
2nd number =  3.14
Added result =  5.14
```

recommend

l('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 Info 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 1.54 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 c f 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;
```

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.

Photoshop plug-ins and debugging

Plug-in Advantages

Can be written C
Work with other programs besides NIH Image
Work with the off-the-shelf versions of NIH Image

Plug-in Disadvantages

Has to be written in C
Hard to debug
Does not have access to Image's internal data structures and routines

Debugging:

From Scott Wurcer <Scott.Wurcer@analog.com> reply on nih-image@soils.umn.edu

For simple plug-ins you can create a dialog that puts information into user
items via SetIText in response to a button or whatever. You can also pause and
restart your code via a few buttons. This allows you to run your plug-in stop
and check on internal variables or other items, and then proceed. You can
usually converge on a simple problem in a few iterations. Then throw out the
unneeded dialog items when you are done.

From davilla@marimba.cellbio.duke.edu (Scott Davilla) reply on nih-image@soils.umn.edu

The best way to debug a plug-in is with MacsBug (yes assembly).
While I have hear the one can debug extension type code with SourceBug,
I've never tried. The problem with extension type code is that most
debuggers depend on symbolic definitions based on code that is compiled
and linked into an application (everything is very well defined). With
extension type code, transfer into it is based on a blind jump to the
address of the plugin code (that gets loaded "manually" ie see the NIH
Image modual 'plugins.p'). Global variables are a big no no unless you
are clever with setting up the plug-in (see the apple tech note on standalone code).

From Carl.Gustafson@cbis.ece.drexel.edu (Carl Gustafson) reply on nih-image@soils.umn.edu

TMON or MacsBug. (I hope you can read 68K :-))

Seriously, you may want to write a simple program that loads the code
resource from the plugin, creates a parameter block, and then jumps to the
plugin's entry point. None of this is for the faint of heart.

Compiler directives

> For some reason even through I place the cute little stop signs next
> to my source code the program does NOT stop at the line.

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

Debugging has probably been turned off for that part of the file using the
{$D-} compiler directive. Many of the source code files in NIH Image have
debugging selectively disabled to keep the generated code under 32K.

From Harald.Felgner@physik.tu-muenchen.de reply on nih-image@soils.umn.edu

several files in the Image project are so large that they cannot be compiled
with the debug option on. Otherwise the resulting code would exceed the 32K
limit. Wayne has included the compiler options {$PUSH} and {$D-} at the
beginning of some files and {$POP} somewhere else inside the same file. So just
checking the D in the project window does not have any effect. The Stop signs
don't work. Enclose the procedures or functions you want to stop in in
{$PUSH}
{$D+}

..
{$POP}.
And look into your THINK Pascal manual for a better explanation.

From Harald.Felgner@physik.tu-muenchen.de reply on nih-image@soils.umn.edu

> I'm not able to use the debug option.

Debugging is disabled in some files of the Image project via compiler
directives, because the resulting code would be too large (>32K).
If you want to debug a certain function or procedure, you have to enclose it in
a combination of the compiler directives {$POP}, {$D+} and {$PUSH}.
See your THINK Pascal manual for help.

Debugging in Think Pascal and Zone Size

From wayne@helix.nih.gov (Wayne Rasband) reply

> I am trying to develop an application where I open a stack of three
>images, take on stack at a time, copy it to a working image, and perform
>some opeartions on it. Normally things work fine, however when I debug the
>program using Think Pascal 4.0, it keeps giving me this error about not
>being able to open the stack because it does not have enough

memory. I
>tried increasing the amount of memory allocated to Think Pascal, without
>much luck. I was wondering as to how do you handle such situations?

You need to increase the Zone Size in Run Options....
--wayne

Index