# INSIDE IMAGE

# General Information

## About this Document

This 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 blame me. You can reach me via email at mvivino@helix.nih.gov or voice at 301-496-9344. 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 written 2/92, current to Image version 1.44b.

If anyone has delved into a topic which involves the internals of the NIH Image program, chances are that somebody else out there could benefit from reading about it. Perhaps you wouldn't mind writing it down and sending it my way?

## Getting around the project

It  hardly needs stating about the usefulness of the search utilities included as part of Think Pascal. Certainly another useful feature in Think Pascal is to hold command while clicking in the top window frame. This allows easy access to procedures in the 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.

## To macro or not to macro, this is the question

It would be difficult to make a broad recommendation that your application 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 in the world 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. If you have something somewhere in the middle, see if a macro can do it. The set of macro calls has grown fairly large and may support your functionality. In covering the internals of image I have more of an emphasis on the pascal routines and data structures of the program. However, I have tried including macro calls in appropriate areas. At the very least, your macro call will be using some of the routines covered in this document and you may benefit from going over what the routine actually does. A rich set of example macro routines is distributed with the NIH Image program. See "About Image" for the set of macro calls.

## Users can use User.p

The User.p module is a good candidate for the placement of pascal source code which you develop. If you 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 fairly fast and simply to the Image program. Since the User.p module has been strategically placed in the build order below most other modules you can call just about any routine in 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.
```

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;
SetupMenus;
GetSettings;
AllocateBuffers;
AllocateArrays;
ConvertSystemClipboard;
DoStartup;
LoadDefaultMacros;
UnloadSeg(@Init);
{InitUser;}                <=======Uncomment this line for User.p
.....{rest of main loop}
end;
```

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.

# Image's and operating on an Image

## Global declarations

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 are listed below. The choice for which to use largely depends upon the application at hand.

**Technique One**: Use ApplyTable to change pixels from their current value to pixels of another value. You fill the table with your function. See example below.

**Technique Two**: A: Use a procedure such as GetLine to move sequentially down  lines of the image. You can access each line as an array. 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.

**Technique Three**: Use Apple's "CopyBits" to wholesale copy a ROI,  memory locations, or an entire image. Example's of CopyBits can be seen in the Paste procedure, some of the video capture routines and many others. Be prepared to not easily understand this one.

**Technique Four**: Use the macro commands GetColumn, GetRow and PutColumn, PutRow. 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 probably be worth your while to examine the pascal examples below. 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;
```

**Technique 1 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;
```

This simple example, 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.

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 2a 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
       end;
    end;
 end;
end;
```

**Technique 2b example**

This prolonged example will perform the same function as the 2a, although it might be easier to see the built in functionality. 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;
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 2b 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.

**Technique 4 example**

The example below is a macro procedure from the Video macros file in the macros folder distributed with Image. If you are interested in using a macro to get at image data, this example should be fairly clear. You don't need a framegrabber or video camera to try using this macro procedure, although many of the example video macros require one.

```
procedure ExtractEvenField(NewWindow:boolean);
{
Replaces odd scan lines with average of neighboring even lines. Can be used to improve the quality
of images that have even and odd fields that are out of sync as the result of subject movement
during capture.
}
var
 i,width,height,row1,row2:integer;
begin
 SaveState;
 if NewWindow then Duplicate('Even Field');
 GetPicSize(width,height);
 row1:=0; row2:=0;
 for i:=1 to height/2 do begin
   GetRow(0,row1,width);
   PutRow(0,row2,width);
   row1:=row1+2;
   row2:=row2+1;
 end;
 MakeRoi(0,0,width,height/2);
 Copy;
 MakeRoi(0,height/4-1,width,height/2);
 Paste;
 RestoreRoi;
 SetScaling('Bilinear; Same Window');
 ScaleAndRotate(1,2,0);
 RestoreState;
end;
```

# Regions Of Interest (ROI) in software

Each time a ROI is drawn on the screen the Info^.RoiRect variable stores information that can be used in your addressing only those pixels within the ROI. Of course there are many types of ROI's. Depending on what you want to do, some of them might be a little more work than your basic rectangular ROI in accessing pixels. Other global variables useful to know:

## type

```
RoiTypeType = (RgnRoi, RectRoi, OvalRoi, LineRoi, FreeLineRoi, SegLineRoi, NoRoi);

PicInfo = record
.....{othr variables}
  RoiRect: rect;
  roiRgn: rgnHandle;
  RoiType: RoiTypeType;
  RoiShowing: boolean;
.....{other variables}
end;
```

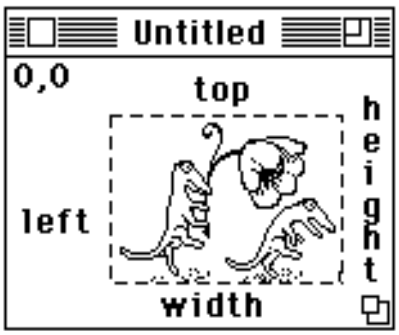
It is often useful to have your routine automatically define the entire image as the area which you will operate on. To automatically select the image you might do the following:

```
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.

# Macro ROI handling

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.


Untitled
0,0
top
left
height
width

## 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

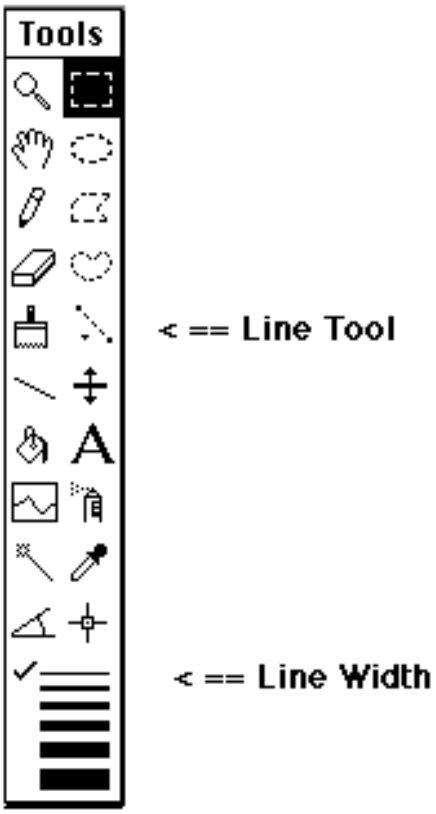
MoveRoi(dx,dy)

Use to move right dx and down dy.

InsetRoi(delta)

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

## Macro Line ROI handling




Outside of macros, lines are drawn by selecting the line tool and using click and drag. The width of the line is set at the bottom of the tool window.

GetLine(x1,y1,x2,y2,LineWidth)

This macro call returns coordinates of the Line ROI and the current selected 'Line Width'. This macro should not be confused or associated with the pascal Getline procedure.

SetLineWidth(width)

Performs the same operation as clicking in the tool window at the bottom..

MakeLineRoi(x1,y1,x2,y2)

Create a line ROI.

The sample macro below shows several of the ROI macro calls.

```
macro 'Make circle from line';
var
 x1,x2,y1,y2,ignore, top,left,width,height:integer;
begin
 GetLine(x1,y1,x2,y2,ignore);
 if x1<0 then begin
   PutMessage('This macro requires a line selection.');
   exit;
 end;
 GetROI(left,ignore,width,height);
 top := y1-((x2-x1)/2));
 MakeOvalROI(x1,top,width,width);
end;
```

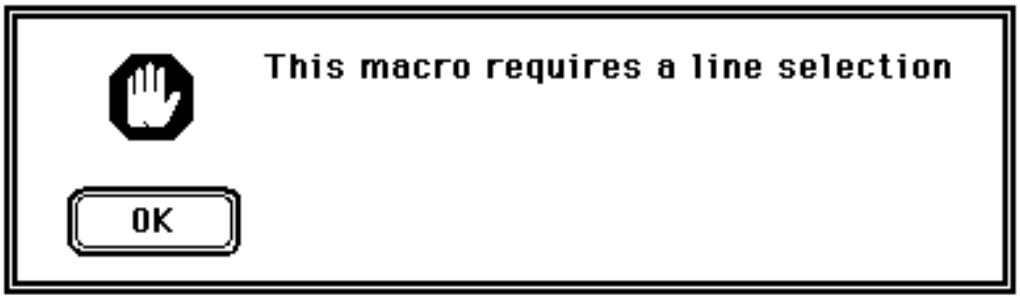
## Other routines involving ROI's

RestoreROI,KillRoi,Copy,Paste,Clear,Fill,Invert,DrawBoundary

# Utilities

# Put and show message

## 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.

Pascal:

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

Macro:

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

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

Macro:

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

Pascal:

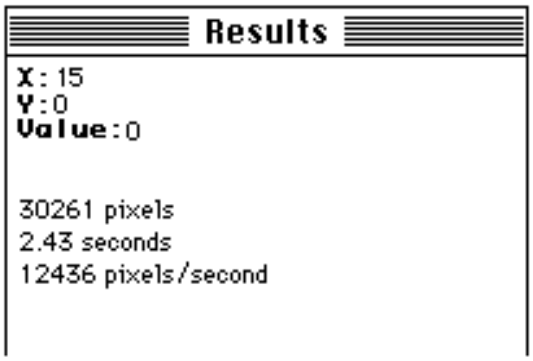
```
PutMessage(concat('Have a ', 'Nice day'));
```

or even something like:

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

## ShowMessage

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




Macro:

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

Pascal:

```
ShowMessage(CmdPeriodToStop);
```

or more involved:

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

# Window

## What kind of window is it?

There are many types of windows in the Image program, although the most important types for the reader might be picture windows and possibly a text window such as the Measurements window. As windows are created, software assigns a 'Kind'. These kinds are nothing but integer values which have been defined in the global declarations.

```
PicKind = 88;
HistoKind = 89;
ProfilePlotKind = 90;
LUTKind = 91;
MapKind = 92;
ToolKind = 93;
ResultsKind = 94;
CalibrationPlotKind = 95;
PasteControlKind = 96;
MeasurementsKind = 97;
```

Example assignment that takes place in the pascal MakeNewWindow procedure:

```
WindowPeek(wptr)^.WindowKind := PicKind;
```

To find out what type of window is frontmost you might write something like the following:

```
var
 fwptr: WindowPtr;
 kind: integer;
begin
 kind := WindowPeek(fwptr)^.WindowKind;
```

After you do this you can check your kind before trying to alter, read or write data. For example:

```
If kind=PicKind then begin
.....DoSomething;
end;
```

Aside from having a kind, a window may also have particular attributes which designate it as a special type of window. This is evident from the global PicType:

```
PicType = (pdp11, NewPicture, normal, PictFile, Leftover, imported, QuickCaptureType, NullPicture,
BlankField, TiffFile, InvertedTIFF, FourBitTIFF, ScionType, PicsFile);
```

**Window Creation**

Should your routine need to, there are several easy ways to create windows in both pascal and a macro.

## Pascal:

```
function NewPicWindow (name: str255; width, height: integer): boolean;
```

There is also a:

```
procedure MakeNewWindow(name:str255);
```

The pascal MakeNewWindow not to be confused with the macro call 'MakeNewWindow'. You should use the NewPicWindow function to create your picture window. It is easy to use, and checks up on available memory before it calls the MakeNewWindow to do the creating. If you are deep in a routine and need to make a window, chances are you already know the width and height which you want. This could be the same size as the window you are working with already, i.e.

```
Info^.nlines
Info^.PixelsPerLine
```

or you might want a window the size of an ROI, the coordinates of which are contained in the Info^.RoiRect rectangle. For the RoiRect, you can find your width and height as:

```
var
  width, height:integer;
.......{other code}
with info^.RoiRect do begin
  width := right - left;
  height := bottom - top;
 if (width = 0) or (height = 0) then
     exit(YourProcedure);
```

For more information on the rectangle structure see Inside Mac I.

Defaults for the width and height are stored in the globally declared:

```
var
 NewPicWidth, NewPicHeight: integer;
```

You set the value for these variables in the preferences dialog box

# Macro:

```
SetNewSize(width,height);
MakeNewWindow('Name')
```

These two macro calls are about as simple as their name implies. The SetNewSize changes the globally declared NewPicWidth and NewPicHeight variables as would using the preferences dialog box. The MakeNewWindow invokes the pascal NewPicWindow function to create your window.

**Using a window in a macro as a temporary dump of data**

Because you don't have access to a full repertoire of pascal and Apple routines in the macro language, you can create a window to dump information to. 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 its 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(nPics);  {The ID of the last window opened = nPics.}
  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;
```

# Key & Mouse State

```
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;
```

If you need to keep the status of one of these functions then assign the status to a boolean variable. There is a global boolean for use with the option key.

```
var
  OptionKeyWasDown:boolean;
```

So in your code you could keep the key status by:

```
OptionKeyWasDown := OptionKeyDown;
```

## CommandPeriod

```
function CommandPeriod: boolean;
```

The CommandPeriod function is used 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

```
Function Button:boolean;
```

These are explained in Inside Mac I.

# 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.

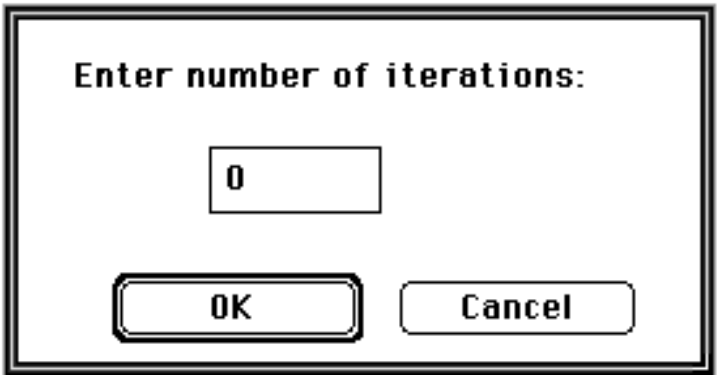
Pascal:

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

Macro:

```
var
 n:integer;
begin
n:=GetNumber('Enter number of iterations:',0);
```

The GetNumber macro will return a real number.



# 'D' is for dialog

# 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. 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);
```

The DoVideoOptions or DoPreferences procedures are good examples for handling a dialog.

# Memory

Show below are two examples 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.

```
procedure Import16BitImage;
 type
  IntArrayType = packed array[0..5000000] of integer;
  IntArrayPtr = ^IntArrayType;
  PixelLUTType = packed array[0..65535] of Unsignedbyte;
  PixelLUTPtr = ^PixelLUTType;
var
..... much deleted
begin
..... much deleted
   PixelLUT := PixelLUTPtr(NewPtr(SizeOf(PixelLUTType)));
   if PixelLUT = nil then begin
     PutMessage('Not enough memory to do 16 to 8-bit scaling.');
     exit(Import16BitImage);
    end;
..... much deleted
 DisposPtr(ptr(PixelLUT));    {free the memory}
```

For a picture window, the allocated memory will have the size of the PicInfo data structure.

```
function NewPicWindow(name:str255; width,height:integer):boolean;
var
  iptr: ptr;
  lptr: ^LongInt;
  SaveInfo: InfoPtr;
  NeededSize: LongInt;
begin

 SaveInfo := Info;
 iptr := NewPtr(SizeOf(PicInfo));
 if iptr = nil then begin
   DisposPtr(iptr);
   PutMemoryAlert;
   macro := false;
   exit(NewPicWindow);
  end;
 Info := pointer(iptr);
 info^ := SaveInfo^;
```

# Text buffer for non-image data

## Where is the text buffer used?

Generally at any time when text handling of long strings or data is needed. It is particularly useful for file I/O.
1) When a plot (example: histogram plot) is converted to text so that it can be saved to disk.
2) An exportation of images as text.
3) When columnar calibration curve-fit XY measurements are saved to disk.

## 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.

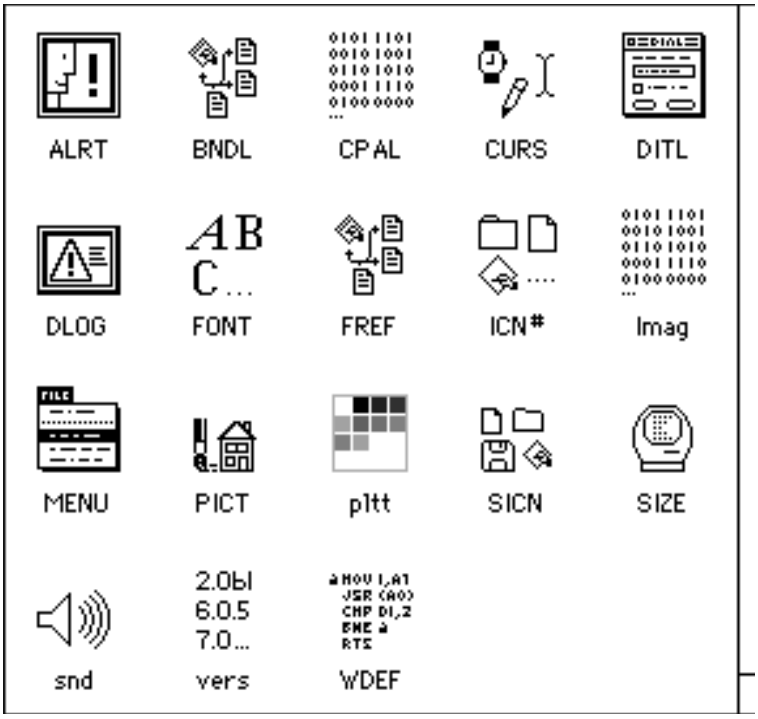
# Text data associated with the "Measurements" window

## Global declaration

```
var
 ListTE: TEHandle;
```

ListTE uses Apple's TextEdit data structure. This data structure is used with the Measurements window to display textual data. Apple's TextEdit data structure is currently limited to 32,000 characters. This same data structure is used in Apple's TeachText software that comes free with your system. Apple has supplied a number of routines for use with TextEdit. This includes TESetText and TEInsert which can be used to place a buffer of text into the TextEdit structure. This and the many other routines required for use with TextEdit are discussed in Inside Macintosh volume 1.

## Resource file




The Image.rsrc file is used for all the dialog template creations, menus, alerts, sounds, etc. If you are not at all familiar with ResEdit, there are now several books on ResEdit in bookstores. There is also a document called HMG™ ResEdit Primer which explains the basics. Think Pascal will include any changes you make when you recompile. As always, work on a copy in case something happens that you didn't quite expect.

# Device drivers (framegrabbers)

## The Following global variables relate to framegrabber support.

```
var
 FrameGrabber: (QuickCapture, Scion, NoFrameGrabber);
 DTSlotBase, ScionSlotBase: LongInt;
 ControlReg, ChannelReg: ptr;
 Digitizing: boolean;
 QuickCaptureInfo, ScionInfo: InfoPtr;
 InvertVideo, HighlightSaturatedPixels: boolean;
 VideoChannel: integer;

 FramesToAverage: integer;

 DTStartTicks, DTFrameCount: LongInt;

 qcPort: cGrafPtr;
 qcWidth, qcHeight: 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 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
  CaptureAndDisplayQCFrame;
```

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

```
procedure GetQuickCaptureFrame;

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

and then use CopyBits to copy to the video memory.

## Index