MACRO NOTES

DOCUMENTATION

1. The NIH Image manual, appendix A provides a description of the macro language and menu by menu explanation of the commands.
2. Reference card - this comes as a macro/text file with NIH-Image (in the Macros folder) and gives an alphabetic listing of the commands.
3. Inside NIH-Image, a supplementary manual written by Mark Vivino which explains many features of the macro language with examples. This also documents the alternative approach of writing user.p Pascal Code to compile with Image.
4. Example macros - numerous macro files come with NIH-Image and provide examples of most commands in use (N.B. One way of using these for reference is to import them into a Word-Processor, then all the files can concatenated into a single, very long, file and keywords searched for).
5. This document - an `alternative' reference file designed to help people who want to use the macro language for biometric work and have no Pascal experience.

GETTING STARTED

1. Launch Image
2. Open an Image file
3. Open the `Image Macros' file (a good one to start with)
4. Do a SaveAs giving another name (to be on the safe side)
5. Experiment - writing new macros etc. in the text window; loading with the load macros commands (special menu, cmnd-9); activate the image window and test operation.

SYNTAX

{comment} - anything inside curly brackets is ignored is ignored by the macro language. N.B. Not terminating the brackets will result in errors such as the macro continuing into the next macro.

[n] - array indices. E.g. rUser[1]
In a macro name defines key which will activate the macro, e.g. Macro `Test [T]`.

() - many uses including: conventional uses in mathematical and logical statement e.g. **25:=(7-2)*5**; procedure calls e.g. **procadd(a,b)**; function parameters e.g. **GetPixel(x,y)**.

`**..**` declares a string, used in various functions, e.g. **Write(`string`).**

; ends a statement, omission will cause error messages, usually referring to the next line in the macro.

rtn always(?) optional, many statements can be included on one line, providing they are separated by semi-colons. Also blank lines can be left in macro files to improve readability.

, separate variables - e.g. when declaring, in procedure calls, in function calls. E.g. **GetPixel(x,y)**.

space Usually optional between items, but cannot be introduced within items. Compulsory between string items e.g. **SetOptions(`Area Mean`)**.

: a few special uses.

`\` carriage return, useable in most message writing functions.

:= arithmetic equals, = can only be used in boolean logic statements. Thus **a:=2+3;** vs **IF a=5 THEN...**

case - Image is not case sensitive, all identifiers (variable and command names etc.)

can be written in upper or lower case, e.g. **KILLROI** or **killroi** or **KillRoi**.

PARTS OF A MACRO FILE

Var statements
= declaration of variables
- placed at the beginning of macros or procedures
- comments placed on the same line are an easy way of documenting variables

Macros
= Discrete programs called by the operator from within NIH-Image
- Cannot be nested, call each other etc.... So, if the same code needs to be used in different macros use procedures.

Procedures
= sub-programs caled by macros.
- need to be listed before the macro(s) or procedures that call them
- using procedures can greatly shorten macro files, improve the general structure of the programming, and simplify revision/development of macro files.
- it is often helpful when writing macro files that make complex use of procedures to include a simple test macro for each procedure.
- procedures can be called from inside other procedures, so long as they listed before them in the macro file (if not an error will be returned when the macro file is loaded).

Comments - anything in curly brackets - {}
- these do not slow macro execution and do speed macro debugging and rewriting.

Menu Manager commands
- these are included in macro names, the main ones are
Macro `(-`; create a dividing line in the menu
Macro `Test /6`; assign command key 6 to the macro (not very useful as Image already uses most command keys).

VARIABLES

A. Global variables (declared at beginning of macro file)
- Initially zero, and reset every time macros are loaded.
- Allow values to be passed between macros.
- Also often convenient if several macros call the same procedure.
- If an initiating macro is run then they can conveniently store constants (e.g. **hscale**, **vscale**, **pi**).

B. Macro variables (declared at beginning of macro)
- Initially zero, and reset whenever the macro is run.
- Are available to procedures called by the macro.
- Are not available to other macros.

C. Procedure variables (declared at beginning of proc, OR in procedure statement, e.g.**procedure procadd(x,y)** - **x** & **y** are implicitly declared.
- Initially zero, reset whenever the procedure is called
Only available within the procedure.

D. Measurement Arrays (**rLength[]**, etc.)
- Although primarily provided for storing the results of measurements these arrays can be used for many purposes and are worth considering as alternatives to variables.
- Initially zero, when Image is launched.
- Values NOT reset by loading macros (cmnd-9).
- **SetCounter(n)** resets values between **n** and current value of **rCount** (higher and lower values are unaffected).
  ResetCounter is equivalent to **SetCounter(1)**
- **Measure** issues a **SetCounter(rCount+1).**
- **Measure; ...; ResetCounter(rCount-1);** will have the effect of doing a measure then disposing of the results without affecting anything else.
- **ShowResults** only shows values up to **rCount**, but higher array values are accessible to macro calls.
- Cannot be used in procedure calls - e.g. **procadd(rX[4],rY[4])** will return an error message.

E. rUser Arrays (**rUser1**, **rUser2**)
- Initially zero, when Image is launched.
- Values are never reset; unaffected by **SetCounter**, **ResetCounter**, etc.

VARIABLE TYPES
Boolean - These can have two values 0-false or 1-true. Attempting to give bolean variables other values will produce error messages when the values are called in boolean statements.
Real - real numbers, e.g 2.345
Integer - whole numbers, e.g. 1,4,8. N.B. By default real numbers rare converted to integers with rounding. The functions **Round(n), Trunc(n)** and **Abs(n)** are available for further control.

<u>LOGIC and LOOPs</u>
CONDITIONAL OPERATION

```
IF <boolean statement> THEN BEGIN
.
.
END ELSE BEGIN
.
.
END;
```

- Extra complexity can be added by **ELSE IF** statements etc.
- The boolean statements can use the operators **= < > <= >=** with either boolean or numeric variables. If **AND** or **OR** statements are used then the sub-statements must be put in braces, e.g. **IF (a>5) OR (a<15) THEN...**
- **BEGIN** and **END** are not needed if there is only a single statement, e.g. **IF .... THEN a:=+5**; *instead of* **IF .... THEN BEGIN a:=+5; END;**

CONDITIONAL LOOPS

```
REPEAT
```

.
.
UNTIL <boolean statement>;

WHILE <boolean statement> DO BEGIN
.
.
END;

- These two forms are diferent in syntax but very similar in practice.
- <boolean statement> can be complex, as conditional operations.

INCREMENTAL LOOP

FOR counter:=var TO var DO BEGIN
.
.
END;

- counter should be an integer variable, it will be incremented by one at the end of each loop.
- Complex arithmetic statements are allowable instead of simple variables/values in the **FOR ... TO ... DO** statement.
- The counter can have its value changed by statements inside the loop; e.g. **IF found THEN counter:=100;**

INTERACTION

1- GETTING INPUT FROM THE OPERATOR

- The number of options here is strictly limited, but with ingenuity quite a lot is possible...

Button Returns true/false depending on whether the operator is pressing the mouse button. E.g. **IF button THEN** ....

GetMouse(x,y) Returns the mouse co-ordinates.

WaitForTrigger Delays macro operation until the mouse button is pressed. E.g. **FOR n:= 1 TO 10 DO BEGIN WaitForTrigger; GetMouse(x,y); PutPixel(x,y,1); Wait(0.2); END;** By combining **WaitForTrigger** and **GetMouse(x,y)** sophisticated conditionality can be written.

GetNumber(`prompt`,default) Gives a simple dialog box to allow input of number. E.g. **newy:=(`Please input new value`,oldy);**

**cmnd-`.'** Stops the macro, after it has finished the command it is working on - there may be a delay if it is a complex command (e.g. a convolution), in these cases it is necessary to hold the cmnd-`.' until the macro stops.

2 - DISPLAYING OUTPUT

Numerous functions display output graphically but the following are specially designed for macro output;

A. GRAPHIC FUNCTIONS DEPENDANT ON "CURRENT-LOCATION"

MoveTo(x,y) - sets current-location, as used in the subsequent functions. N.B. Current-location is not set by the mouse position.

LineTo(x,y) - Draw line from current-location to (x,y)

DrawNumber(n) - Draw number at current-location

DrawText(`text`) - Draw text at current-location

Write(e1,e2,..) - Draw text and/or numbers at current-location

WriteLn(e1,e2,..) - Draw text and/or numbers at current-location, but with redefinition of current-location.

B. NON-GRAPHIC OUTPUT

ShowMessage(e1,e2,..) - displays message in values window. The message will stay after the macro has run until the values window is redrawn - e.g. by use of one of the measuring tools. Useful for displaying progress of a macro, debugging, etc.

PutMessage(e1,e2,..) - displays message in a dialog box. Useful for warning messages.

Beep - issues a beep. Unsophisticated but often handy.

ShowResults - Open the results window

MISSING FEATURES/THINGS YOU CANNOT DO

Define new arrays - instead exploit the measurement and rUser arrays.

Access numerous constants - many constants can be set, e.g. **SetScale(scale,units)** but only a very few can be read, and some constants can be neither read nor written. E.g. Pixel aspect ratio, can only be set via the Set Scale dialog box and can only be determined by a macro by measuring a square ROI.

Access results Image must know but doesn't want to tell you - a variation on the above. Essentially the only results you can get are those which special routines have been written to provide you with. Other results are unobtainable even if they must have been calculated - e.g. Measure will return the maximum pixel value found, but not the location of this pixel, so forget that as a peak finding approach. (N.B. As of Image 1.56 a commnd GetScale has been introduced, but the general principle that not all constants are accessible remains).

Access extra Maths functions - the range of Maths functions provided by NIH-Image is limited, but carefully selected, consult your maths texts to remeber hw to derive the others. E.g. **arctan** is provided but not arcsin or arccos. But tan(a)=sin(a)/cos(a) and sin(a)^2=cos(a)^2, so you can get round that shortcoming.

Access Maths constants - no mathematical constants are defined. Define them in the macro or wrie out the number each time. (N.B. pi=3.141592654)

Select tools - the active tool on finishing a macro is always the same as when starting the macro.

Scroll or zoom windows - tough you just have to do this using the tools (N.B. cmnd-U unzooms the current window).

Define irregular ROIs - only square and line ROIs can be defined from within the macro language. This is restricting if for instance you have used macro routines to identify a sequence of points defining an object and would then like to measure it.

Import data from files into the measurement arrays - but Wayne is thinking of implementing this.

Retrieve overwritten pixels - there is no overlay available in Image so e.g. writing text on an image removes data pixels. The alternative is to work on a duplicate Image, use **RevertToSaved** (cmnd-R) etc. - there are lots of useful window handling commands in Image.

Work on the plot, or histogram (or any non-image windows), the macro language works essentially on image windows and you cannot e.g. draw results on the plotprofile - although it is possible to make an image file with a picture of the plot and then work on that - see the plotting macros which come with Image.