Macro programming - Contents

- use of internal Gifa configuration - use of blocking event to wait for the user
- graphic user interaction and clicking on the spectrum
- varying formbox via file

INTRODUCTION

One of the strength of the *Gifa* program is its macro language. Many features of the standard program are actually developed as macros. For the lay user, this insures that the program will react exactly the same on all the computer platforms for which *Gifa* is designed; and warrants a fast development of high level features.

For advanced users, the benefit is much larger of course, as you can adapt the distributed macros to more finely fit your needs, or you can write your own macros to realises specific features.

The macro language is rich enough to permit to create easily sophisticate mathematical processing or high level control structures with a full graphical user interface.

The purpose of this chapter is to help you in writing such macros.

Macros are simply files that contains commands as they could have been typed at the *Gifa* prompt. Any command that can be used in-line can be used in a macro, and nearly every command and syntax that is used in a macro can be used in-line. Of course there is more than that, you can define variables, use control structures, access contexts, use functions, build graphic interfaces, macros can be called from within macros.

Most of this manual describes feature which are not specific to macro programming, but rather are generic to the program. They have been put here because they are mostly relevant to macro programming.

Have Fun !

BASIC OF MACRO PROGRAMMATION

Introduction to the control language

It is possible in the *Gifa* program to write command files (macros). The macro file contains the very same commands that would be typed on line. The macro is then invoked by typing its name, with or whitout a leading @ sign :

```
Gifa> @file_name
```

is nearly equivalent to

```
Gifa> file_name
```

The syntax

```
Gifa> @(any_evaluated_expression)
```

or

```
Gifa> @$var
```

are perfectly valid, however, in this case, as @(..) is parsed as a command, no space is allowed within the evaluated expression.

Using the @ forces the macro execution, thus permitting to have a macro with the same name as an internal command. In some cases, the @ syntax might be slightly faster.

In VMS, a ‘.GIF’ extension will be added to the file name if no dot is present in the macro name. In UNIX the macro name is the file name, sometime names with a .g extension are chosen (used to be .gif before the v4 version, but has been changed to separate from the popular graphic file format .gif). Any characters (but the blank) can be used in macro name, so ../test/my_macro or

/usr/local/gifa/macro/startup.g or [marc]my_macro.gif;-1 are valid names.

A macro consists of a series of regular *Gifa* commands used as in the interactive mode. Up to 9 recursive calls may be nested (i.e. you can call a macro from within a macro, down to 9 nested levels). The macro execution can always be stopped with the ^C command.

Control structures, evaluated expressions, and variables are available in the *Gifa* macro language. More about this latter on.

Detailed Syntax of the Gifa Input.

Each *Gifa* command line is composed of a series of "words" separated by blanks. A word can be a command or a parameter. Words are strictly separated by blanks, however a word can be of several nature :

- **literal are as typed:**

```
chsize
```

- **literal string are enclosed within single (') or double(") quotes:**

```
'a string' or "another string" or "yet an'other valid string"
```

single and double quotes can be freely mixed at will, and are strictly equivalent.

- **The special character % (to be used only as a parameter for a command) takes the place of the default value of the command:**

```
EM %
```

- **The special string %% takes the place of the whole series of all the default values of the remaining parameters of the command :**

`ZOOM 1 % % % % and ZOOM 1 %%` are equivalent (ZOOM needs 4 parameters when in 2D)

Many commands know how to deal with this syntax, for instance `BCOOR` (the base line correction command) needs a series of point to compute the baseline correction. If you have previously used the macro `point` and clicked on a set of points on the data, you will be prompted for the values corresponding to these points. So

```
bcorr 1 1 %%
```

will just do the work.

The baseline correction interactive macros use this feature.

- **variables are substituted before use (more about this later):**

```
$variable1
```

- **expressions, enclosed within parentheses, are evaluated before use (more about this later):**

```
( 2*(cos(3) + 1))
```

- **any of the preceding syntaxes may be preceded with a < symbol. In which case the input is interpreted as a file name, and the parameters takes the value of the next line read in the file (if already `OPENed`). A complete line is read at each time.**

Each of these syntax can be used in any places; however, a single word cannot match a series of words :

These are valid syntax :

```
row $i ; select row i

row (%+1) ; select the next row

col (2*$i +1) ; select col 2i+1

set apod = 'sin' $apod 0.5 ; perform sin 0.5

read ("/usr" // $user // $exp) ; read a file

("simu" // "noe") ; execute the command SIMUNOE

set f = test open $f row <$f ; select the row whose index ; is found in file
'test'
```

These are invalid syntax :

```
set apod = 'sin 0.5' $apod
```

a single word (here 'sin 0.5') cannot be used to match several words

```
EM 1+2
```

the parenthesis are needed for expression to be evaluated

A line cannot be longer than 256 characters, otherwise the end the line will be lost. This limit of 256 characters is hard-wired every where in the macro language, no string can ever get longer than this number.

In the case of commands however (and only in this case), if a longer line is needed, a continuation sign is available : \

e.g.

```
Gifa> print 'A line cannot be very long' print \

\Gifa> 'But can be continued' print 'as many time as you wish'
```

You can put as many continuation sign as you want, however, these signs are considered as “word”, so they should be blank separated, and cannot be used within evaluated expressions (see below). There is no actual limitation on the length of one command line (using \).

VARIABLES

Basic

Variables can be used in *Gifa*, this feature is mainly useful when writing macro commands, however, it can be used at any time in the program. Variables are set (and created if not yet defined) with the command : SET :

```
Gifa> set foo = 'hello world' set bar = 3

Gifa> set var := 'hello mom' ; other syntax, see below
```

Variables are then just used by prefixing the name with $:

```
Gifa> em $bar print $var
```

Variables can be used anywhere a regular entry is needed. Variables are not typed, but contain always a string. The maximum string is currently of 256 characters. The string is interpreted as a number when needed. Variable names are not case sensitive, the name is a maximum of 31 characters long; and must be built from purely alpha-numeric characters only (including the underscore : _). The number of available variables is limited at compile time (and reported by the macro `config`, and variable $VAR_MAX (see below)), but a larger table can easily be built by recompilation if needed.

Scope of variables

Variables are allocated the first time they are set; referencing a variable not yet allocated generate an error. Variable are automatically deallocated when exiting the macro; they cannot be accessed by any other macros that would be executed during the life of the variable. In other words, variables have local scope, and are volatile (dynamically allocated and removed).

Variables created at the interactive level (not within a macro) have a special treatment: their life will be as long as the program (unless an `UNSET` command is explicitly used), and they can be accessed by any other macro: they are static and have global scope.

If you which to create such a static variable from a macro (useful to remember parameters from one call to the other), you can use the following syntax :

```
set d := 'this variable is static'
```

d is created static. If `d` was already declared as volatile previously in the macro, the preceding command has the effect of creating a new variable, called $d, but in the static storage, independent of the volatile $d.

Associative arrays

Associative arrays can be built using the construction [index] :

```
Gifa> set d[$k] = (cos($k/10))
```

Associative arrays are special entries where each entry is created when needed, entries do not have to be sequential, and the index does not even have to be integer :

```
$d[1] ; $d[3] ; $d[$var] ; $d['foo bar']
```

is a valid array. Associative arrays are coming from the standard UNIX program `awk` or `perl`, search for a manual of this program if you need some more help with associative arrays. The `FOREACH` control permits to go through all the entries of a given array (see below). The function `nextlm()` can do the same with a much finer control.

Associative arrays can be used to simulate many other data-structure. You can use them to set-up multi-dimensionnal arrays; tree structure; structured data-set (in the C sense); etc... See examples below.

Special Commands

SET, UNSET, MUNSET

These commands permits to create and set the values of variables, as well as to delete them if needed.

```
SET varname = value
```

This line set the variable called `varname` to the value `value`. Subsequent use of `$varname` will return `value`

The variable is created if it did not existed before, or set to the new value if it existed.

```
SET var2 := value ; create and set a global variable
```

Note the := sign with imforce the variable to be of global scope.

```
UNSET varname
```

remove the variable from the table

```
MUNSET list of vars finishing with a *
```

removes all the variables in the list.

DUMP

```
DUMP ; dump the content of all currently defined

; variables
```

The format of the `DUMP` is : name of the variable ; context of the variable : (20 is static, 21 to 29 are macro call levels, higher are for formboxes) ; content of the variable.

Using `DUMP` you will find that there might be entries in the variable table that do not correspond to user variables. This is due to the fact that the parser uses this table for particular use (dbm arrays, positions of label, location of `GOTO, IF, WHILE,FOR` and `FOREACH` controls). These entries have different formats that the user variables, thus cannot be mistaken with them. Some of these entries hold binary data that might disturb the terminal when using the DUMP command.

Contexts (Internal variables)

The contexts are very special variables, not defined in the variable table, but which takes the value of the internal parameters of the *Gifa* program. They are actually not *variables*, as they **should never be set** but should be always be changed by using the associated command.

The list of contexts is :

```
$_ value of the next parameter present on the calling line.
```

This context is very special, **it cannot be used within evaluated expressions**.

if the following command is used :

```
@test 3 test.001
```

within the file test, $_ will be 3 the first time and test.001 the second time. If no value is present on the calling line, the user will be prompted for the value

```
$ABSMAX current value of ABSMAX

$ARG True (1) if arguments are present on the calling line

(within macro only)

$BUTTON 1,2 or 3 depending on which mouse button was last clicked

$CALIBDI[1] the current calibrating distance, as defined with the

CALIBDI command

$CALIBDI[2] the current calibrating relaxation rate, as defined

with the CALIBDI command
```

$CCOLOR current value for CCOLOR

$CDISP2D current value of command CDISP2D

$CHI2 value of the chi2 returned by the last command : MAXENT,

LINEFIT, RT->PK

$COL index of the last extracted column in 2D

$COLOR current value for COLOR

$CONFIG_GRAPH is true if graphic is possible (X_Windows is

connected)

$CONFIG_OS The system type, as returned by the CONFIG command

$CONFIG_PLOT The plot driver, as returned by the CONFIG command

$CONFIG_WIN The window manager, as returned by the CONFIG command

$CX current value for CX

$CY current value for CY

$CZ current value for CZ

$C_ABSMAX value of ABSMAX of the currently JOINed dataset

$C_DFACTOR value of DFACTOR of the currently JOINed dataset

$C_DMAX value of DMAX of the currently JOINed dataset

$C_DMIN value of DMIN of the currently JOINed dataset

$C_DIM value of DIM of the currently JOINed dataset

$C_FREQ value of FREQ of the currently JOINed dataset

$C_FREQ1 value of FREQ_1D of the currently JOINed dataset

$C_FREQ2 value of FREQ_2D of the currently JOINed dataset

$C_FREQ3 value of FREQ_3D of the currently JOINed dataset

$C_HEADER last value accessed with the GETHEADER command

$C_JOINED is if there is a currently JOINed dataset, 0 otherwise

$C_OFFSF1 value of OFFSET_1 of the currently JOINed dataset

$C_OFFSF2 value of OFFSET_2 of the currently JOINed dataset

$C_OFFSF3 value of OFFSET_3 of the currently JOINed dataset

$C_SIZEF1 value of SI1 of the currently JOINed dataset

$C_SIZEF2 value of SI2 of the currently JOINed dataset

$C_SIZEF3 value of SI3 of the currently JOINed dataset

$C_SPECWF1 value of SPECW_1 of the currently JOINed dataset

$C_SPECWF2 value of SPECW_2 of the currently JOINed dataset

$C_SPECWF3 value of SPECW_3 of the currently JOINed dataset

$C_TYPE value of ITYPE of the currently JOINed dataset

$DFACTOR scale factor for damping scale

$DIM current value of DIM

$DISP1D current value of command DISP1D

$DISP2D current value of command DISP2D

$DISP3D current value of command DISP3D

$DIST the result of the last DIST command

$DMAX Upper bond of damping scale

$DMIN Lower bond of damping scale

$FND_PK gives the index of the found entry (FIND)

$FND_PK_DST gives the distance to target of the found entry (FIND)

$FREQ main frequency (1H) of the spectrometer

$FREQ_1D frequency in 1D (in MHz)

$FREQ_1_2D frequency in 2D in F1 (in MHz)

$FREQ_1_3D frequency in 3D in F1 (in MHz)

$FREQ_2_2D frequency in 2D in F2 (in MHz)

$FREQ_2_3D frequency in 3D in F2 (in MHz)

$FREQ_3_3D frequency in 3D in F3 (in MHz)

$GB1 value of gb in F1 (in Hz)

$GB2 value of gb in F2 (in Hz)

$GB3 value of gb in F3 (in Hz)

$GIFAPATH current PATH used for macro, set by th SETPATH command

$HOME equivalent to the $HOME variable in UNIX

$ITYPE_1D value of itype in 1D

$ITYPE_2D value of itype in 2D

$ITYPE_3D value of itype in 1D

$LB1 value of lb in F1 (in Hz)

$LB2 value of lb in F2 (in Hz)

$LB3 value of lb in F3 (in Hz)

$LEVEL current value of command LEVEL

$LICENCE The licence as returned by the CONFIG command

$LOGA current value of command LOGA

$MAX[1] value of last computed max computed with the command MAX

$MAX[2] value of last computed min computed with the command MAX

$MEM_MAX The larger data set available, as returned by the CONFIG

command

$MEM_PRO_1D The size of the protected 1D area, as returned by the

CONFIG command

$MEM_PRO_2D The size of the protected 2D area, as returned by the

CONFIG command

$MEM_PRO_3D The size of the protected 3D area, as returned by the

CONFIG command

$NAME name of the last read data-set

$NAME_1D name of the last 1D data-set read

$NAME_2D name of the last 2D data-set read

$NAME_3D name of the last 3D data-set read

$NAR number of A.R. coefficients as listed by ARLIST

$NOISE value of the noise, as given by the NOISE command

$NPK1D The number of entries in the 1D peak table

$NPK2D The number of entries in the 2D peak table

$NPK3D The number of entries in the 3D peak table

$NPOINT The number of entries in the point stack

$NRT Number of root as listed by RTLIST

$NSVD Number of root as listed by SVDLIST

$OFFSET_1D spectral offset in 1D (in Hz)

$OFFSET_1_2D spectral offset in 2D in F1 (in Hz)

$OFFSET_1_3D spectral offset in 3D in F1 (in Hz)

$OFFSET_2_2D spectral offset in 2D in F2 (in Hz)

$OFFSET_2_3D spectral offset in 3D in F2 (in Hz)

$OFFSET_3_3D spectral offset in 3D in F3 (in Hz)

$ORDER current value for ORDER

$PH0 0th order of the last phase correction

$PH1 1st order of the last phase correction

$PK1D_A[i] amplitude of the ith entry in the 1D peak table

$PK1D_A_ERR[i] error on the previous quantity

$PK1D_F[i] position (in index) of the ith 1D entry

$PK1D_F_ERR[i] error on the previous quantity

$PK1D_P[i] phase of the ith entry in the 1D peak table

$PK1D_T[i] analytical form of the ith entry in the 1D peak table

GAUSS, LORENTZ, or UNKNOWN

$PK1D_W[i] width (in index) of the ith entry in the 1D peak table

$PK1D_W_ERR[i] error on the previous quantity

$PK2D_A[i] amplitude of the ith entry in the 2D peak table

$PK2D_A_ERR[i] error on the previous quantity

$PK2D_F1F[i] F1 position (in index) of the ith entry in the 2D

peak table

$PK2D_F1F_ERR[i] error on the previous quantity

$PK2D_F1W[i] F1 width (in index) of the ith entry in the 2D peak table

$PK2D_F1W_ERR[i] error on the previous quantity

$PK2D_F2F[i] F2 position (in index) of the ith entry in the 2D

peak table

$PK2D_F2F_ERR[i] error on the previous quantity

$PK2D_F2W[i] F2 width (in index) of the ith entry in the 2D peak table

$PK2D_F2W_ERR[i] error on the previous quantity

$PK3D_A[i] amplitude of the ith entry in the 3D peak table

$PK3D_F1F[i] F1 position (in index) of the ith entry in the 3D

peak table

$PK3D_F1W[i] F1 width (in index) of the ith entry in the 3D peak

table

$PK3D_F2F[i] F2 position (in index) of the ith entry in the 3D

peak table

$PK3D_F2W[i] F2 width (in index) of the ith entry in the 3D peak table

$PK3D_F3F[i] F3 position (in index) of the ith entry in the 3D

peak table

$PK3D_F3W[i] F3 width (in index) of the ith entry in the 3D peak table

$PKNAME The name used by the last PKREAD / PKWRITE command

$PLANE[1] axis of the last extracted plane in 3D

$PLANE[2] index of the last extracted plane in 3D

$PLOTAXIS[1] current value for the unit used in PLOTAXIS

$PLOTAXIS[2] current value for the tick distance in x axis

$PLOTAXIS[3] current value for the tick distance in y axis

$PLOTOFFSET[1] current value for plot offset on X axis

$PLOTOFFSET[2] current value for plot offset on Y axis

$POINTX[i] X coordinates (in index) of ith point in the point stack

$POINTY[i] Y coordinates (0..1 in 1D; in index if 2D) of ith

point in the point stack

$RANDOM a random variable in the range 0..1 with equiprobable

distribution

$RANDOMG a random variable with normal law, unit variance and zero

mean

$RANDOMZ same as $RANDOM but resets the random series

$RELAX the result of the last RELAXRATE or SLOPE commands

$RCRYST the result of the last RCRYST command CALIBDI

$ROW index of the last extracted row in 2D

$SCALE current value of the context SCALE

$SCOLOR current value of the context SCOLOR

$SHIFT value of the offset, as given by the SHIFT command

$SI1_1D size of the 1D buffer.

$SI1_2D size in F1 of the 2D buffer

$SI1_3D size in F1 of the 3D buffer

$SI2_2D size in F2 of the 2D buffer

$SI2_3D size in F2 of the 3D buffer

$SI3_3D size in F3 of the 3D buffer

$SI_TAB size of the TAB buffer

$SIGN current value of context SIGN

$SIGN_PEAK current value of context SIGN_PEAK

$SPECW_1D spectral width in 1D (in Hz)

$SPECW_1_2D spectral width in 2D in F1 (in Hz)

$SPECW_1_3D spectral width in 3D in F1 (in Hz)

$SPECW_2_2D spectral width in 2D in F2 (in Hz)

$SPECW_2_3D spectral width in 3D in F2 (in Hz)

$SPECW_3_3D spectral width in 3D in F3 (in Hz)

$SUMREC value returned by the last SUMREC command

$SUMREC_ERR error on the previous quantity

$TAB[i] value of the ith point in the TAB buffer

$UNIT current value for UNIT

$UNITY current value for UNIT_Y

$VAR_MAX The total number of user variable available, as returned

by the CONFIG command

$VERSION The current version, as returned by the CONFIG command

$VHEIGHT current value of command VHEIGHT

$WIDGET The id of the current graphic form

$ZONE[1] lower F1 coord. of the mouse-selected region (in index)

$ZONE[2] lower F2 coord. of the mouse-selected region (in index)

$ZONE[3] upper F1 coord. of the mouse-selected region (in index)

$ZONE[4] upper F2 coord. of the mouse-selected region (in index)

$ZOOM 1 if in ZOOM mode

$ZOOM_1D[1] left coordinate of the 1D zoom window (in index)

$ZOOM_1D[2] right coordinate of the 1D zoom window (in index)

$ZOOM_2D[1] lower F1 coordinate of the 2D zoom window (in index)

$ZOOM_2D[2] left F2 coordinate of the 2D zoom window (in index)

$ZOOM_2D[3] upper F1 coordinate of the 2D zoom window (in index)

$ZOOM_2D[4] right F2 coordinate of the 2D zoom window (in index)

$ZOOM_3D[1] "left" F1 coordinate of the 3D zoom window (in index)

$ZOOM_3D[2] "right" F1 coordinate of the 3D zoom window (in index)

$ZOOM_3D[3] "left" F2 coordinate of the 3D zoom window (in index)

$ZOOM_3D[4] "right" F2 coordinate of the 3D zoom window (in index)

$ZOOM_3D[5] "left" F3 coordinate of the 3D zoom window (in index)

$ZOOM_3D[6] "right" F3 coordinate of the 3D zoom window (in index)

EVALUATED EXPRESSIONS

Expressions can be evaluated in the *Gifa* program. Expressions **should** be enclosed in parentheses. Expressions can use most of the entries found in the paragraph SYNTAX, at the exception of the special entries : `%%` `<from_file` and the pseudo variable `$_` which are illegal in expressions.(see below)

`( 2*(5-1)) (cos($i))` are examples of expressions. Expressions can be used whenever a *Gifa* input is needed (command as well as parameters). Expressions must fit on one line (i.e. 256 characters long), continuation mark cannot be used within expressions.

Values can be numeric, string or logical. Strings should be enclosed within '..' or "..". Logicals are stored as 0 for false and 1 for true (or any non-zero value).

When evaluating expressions, all the internal computations are untyped and performed on strings. This permits to mix integer and string freely. Thus the following expressions are perfectly valid :

```
( cos(1) // 'a string' // ($i==1))

( toupper($file // ($i+1)) )

( log ('0.1') )
```

Blanks

In contrast with the parsing of action and parameters, blanks are not required to separate values within expression. As long as the meaning is unequivocal, tyou can freely decide to use or not use them. So the following are equivalent

`(2 + 3 * 5)` and `(2+3*5)`

There are a few cases however where you should be careful: There should be **no** blanks between a function name and the open parenthesis `(cos($x))` is fine `(cos ($x))` is not. There **should** be one blank when using a variable name and a diadic operatr starting with a letter. eg;
`($var s= 'a')` is fine `($vars='a')` is equivocal.

List of functions and operators

The following operators and functions are implemented :

mathematical

- **the regular 4 operations : + - / * e.g. : (2*3 - 1)**
- **the modulo function : % (12 % 5)**
- **the power ^ operator (3^2.1)**
- **the regular mathematical functions :**

sqrt(x) cos(x) sin (x) atan(x) log(x) exp(x) abs(x) int(x) max(x,y) min(x,y), round(x)

- **the special function *power2(n)* will have the value of the closest power of 2 below or equal to the number n : power2(130) will be 128 (2^7)**

the alphanumeric operators :

- **the concatenation operator : // ('string1' // "string2")**
- **the formatting operator : ;*equivalent* to // ' ' //**
("Value is:" ; $a) *is equivalent to* ("Value is:" // ' ' // $a)

- ***toupper(st)*** **put string in upper case**
- ***tolower(st)*** **put string in lower case**
- ***sp(i)*** **generates a string of i blanks**
- ***len(st)*** **is the length of the string**
- ***index(st1,st2)*** **is the position of st2 located in st1, 0 if not present**
- ***subst(st,i,j)*** **is the substring from st, starting at i, ending at j**
- ***isalpha(st)*** **and** ***isnumb(st)*** **are true whenever st is (respectively) alphanumeric or numeric**

list operators

- ***head(st)*** **will be the first word in string st (blank separated)**
- ***tail(st)*** **will be the string st but the first word.**
- ***headx(st,c)*** **and** ***tailx(st,c)*** **are equivalent to head and tail but the character c is used rather than blank.**

One typical use of the head() and tail() functions is to parse parameters concatenated into a single string, e.g.:

```
set params = 'value1 value2 value3 value4'

set par1 = (head($params)) set par2 = (head(tail($params))) etc...

; one obtains : par1 contains 'value1' par2 : 'value2', ...
```

another use is using such a string for iterative constructs :

set params = 'value1 value2 value3 value4'

```
while ($params s! '')

set val = (head($params)) set params = (tail($params))

some processing on $val ...

endwhile
```

- **the next element function :** ***nextlm(array,entry)*****. If i is an entry in the associative array $array, the construction nextlm(array,i) will have the value of the next available entry. The end of the array is notified with an empty string. The series is also initialised with an empty string. The series is gone through in the internal order which is not related neither to the input order nor the sequential order. For instance the**

following macro will print all the entries of the array $table :

```
set $index = (nextlm('table',' '))

while ($index s! ' ')

print ($index // ' : ' // $table[$index])

set $index = (nextlm('table',$index))

endwhile
```

However the command `FOREACH` is much simpler to use for the same purpose.

Check for instance the macro `tunset`, which permits to remove all the entries of an array.

logical operators

- **the numeral comparison operators : == != < > <= and >= for comparing numbers :**

  ($x<=0.5)

- **the string comparison s= (equal) and s! (different) for comparing strings :**

  ($s s= 'name')

- **the logical operators : | (or) and & (and) : (($x<=0.5)&($s s= 'name'))**

- **the not operation : ! : (!($i==1)) (!$arg)**

- **the function *eof(file_name)* will be true if the last input from file file_name (with <file_name) had reached the end of the file, will be false otherwise.**

- **the function *dbm(array_name)* is true if array_name is bound to a dbm file with the DBOPEN command**

- **the function *exist(var_name)* will be true if var_name is a user variable either local or global. e.g.**

```
if (exist("i")) set j = ($i+1)
```

j is computed only if $i exists as a variable.

The special syntax `(exist("foo[]"))` checks whether the array foo exists with at least one index. It works both for regular and dbm arrays.

data and file access functions

• the functions *va1d(i), val2d(i,j)* and *val3d(i,j,k)* returns the value of the content of the main *Gifa* working buffers. In 2D i and j are the index in F1 and F2 respectively, in 3D in F1, F2 and F3. This replaces the old $VAL[] construct. *valamb(i,j)* returns the value of the amoeba buffer.

• the *hexist(name)* function returns true or false whether the parameter name is available in the header of the currently joined file. If this parameter is available, the *header(name)* returns its value

• The *sh(command)* function permit to very easily call an operating system *command* and the function returns the value of the first line returned by the command. Look at the command SH for further details.

• The *fexist(file_name)* return true if the file called *file_name* currently exists.

• the functions *itoh(index,dim,axis), htoi(hertz,dim,axis), itop(index,dim,axis), ptoi(ppm,dim,axis), htop(hertz dim, axis), ptoh(ppm,dim,axis), itos(index,dim,axis), stoi(second,dim,axis), itod(index,dim,axis), dtoi(damping,dim,axis), itot(index) and ttoi(tabulated)* perform unit conversion. Respectively : index to hertz and back; index to ppm and back; hertz to ppm and back; index to second and back, index to damping factors and back; index to tabulated and back.
For each function, index, ppm, hertz, second, damping or tabulated is the value to be converted, dim is the dimension to use (1, 2 or 3) and axis is the axis to use, depending on which spectral widths and frequencies you want to use for the conversion. Note that since there is only one tabulated axis that can be defined the additional parameters are useless. If dim = 0, then the conversion is done in the context of the currently JOINed file rather than in the context of the internal buffers.

eg:

`(itop(100,2,2))` is the value in ppm of the point of index 100, along axis F2 in 2D

`(ptoi(3.3,2,1))` is the index location of the point located at 3.3 ppm , along axis F1 in 2D

`(stoi(0.1,1,1))` is the index location of the point located at 0.1 sec from the beginning of the 1D FID

These constructs can be freely mixed.

CONTROL STRUCTURES

Control structures

There are several control structures in the *Gifa* language. They are all pretty classical, and should not make you any problems.

FOR .. ENDFOR

```
FOR var_name = initial TO final { STEP step}

.. some Gifa code

ENDFOR
```

This construction permits to write loops. The variable called *var_name*, will be created local to the macro if it does not yet exist, set to the value *initial*, and incremented by 1 for each loop. The increment can be different from 1 (but always numeric) with the optional field STEP *step*. *step* can even be negative, in which case the variable will be decremented. In any case, the test performed to determine whether the loop is finished or not is (*var_name* > *final*) if step is positive or (*var_name* < *final*) if step is negative.

FOREACH .. ENDFOR

```
FOREACH var_name IN array_name

.. some Gifa code

ENDFOR
```

Equivalent to `FOR .. ENDFOR`, but here *var_name* will take successively all the value possible for an index in the array *array_name*. Thus `$array_name[$var_name]` will take all the value of the array. *array_name* has to be, either a user associative array, or a dbm bound array. The array will be accessed in an apparently random manner,

`FOREACH` does not work for context arrays (such as $ZOOM[]).

WHILE .. ENDWHILE

```
WHILE some_value_to_be_evaluated

.. some Gifa code

ENDWHILE
```

The *Gifa* code will executed as long as *some_value_to_be_evaluated* is true (is non zero). The value to be evaluated can be a variable or a complex evaluated expression into parentheses.

IF .. ELSIF .. ELSE .. ENDIF

The IF command has two distinctive syntaxes : a complete IF .. THEN construct with ELSIF and ELSE possibilities and a one-line IF without possible nesting nor else.

The multi-line IF is as follow :

```
IF test THEN

..commands on several lines

{ ELSIF test2 THEN
```

```
..commands } (eventually many exclusive tests )

{ ELSE (default case)

..commands }

ENDIF
```

The different commands will executed conditionally on the value of the tests. Any non-zero value is considered as true. IF may be nested, with no limit on the number of nesting.

The one-line IF is as follow :

```
IF logical_value ...remaining of the line ...
```

All the commands on the remaining of the command line are executed only if *logical_value* holds true (is non-zero).

Examples :

```
IF ($X =< 0) GOTO SKIP

print 'Hello' IF ($ITYPE_1D == 1) set x = ($x+1)

IF (!eof(input)) print 'Still reading the file' \

set line = (%+1) \

IF ($line<$linemax) goto continue
```

tests to be used are described in the “evaluated expression” paragraph.

GOTO

```
GOTO label
```

will redirect the execution of the macro to the symbol =label appearing somewhere else in the same file.

Example

```
=loop

...any kind of processing...

goto loop
```

GOTO should not be used to jump into FOR FOREACH or WHILE loops, nor into IF .. THEN structures, Unpredictable execution will occur.

All these commands may be freely nested. However, except for `GOTO` and the one-line `IF`, these commands (`FOR, FOREACH, ENDFOR, WHILE, ENDWHILE`, and `=label`) should appear alone on one line, with no code preceding nor following the command (or the label), but eventually followed by a comment.

FIND, FOREACH - WITHIN

These two commands are meant for accessing arrays (dbm or plain array) storing coordinate information such as peaks for instance. The common structure of such arrays will be that the coordinates have to be stored first in the array.

Then the `FIND` command, given an array name, the number of coordinates to search for, and a target point, will return the index of the closest entry in the array. Returned values are in the contexts `$FND_PK` which give the index of the found entry, and in `$FND_PK_DST` which gives the distance to target.

Let's give two examples of arrays with coordinates set in the beginning of each entry :

```
SET chem[1] = "1.1 spin_name1"

SET chem[2] = "2.5 spin_name2"

etc..

SET peak[1] = "1.1 1.1 peak_name 1"

SET peak[2] = "1.1 2.5 peak_name 2"

SET peak[3] = "2.5 3.6 peak_name 2"

etc..
```

```
FIND chem 1 1.3
```

then $FND_PK is 1 and $FND_PK_DST is 0.2

```
FIND peak 2 1.3 2.2
```

then $FND_PK is 2 and $FND_PK_DST is 0.36 (Cartesian distance)

With the same kind of array, it is possible to restrict the scanning performed with the FOREACH command to a given zone :

```
FOREACH var_name IN array_name WITHIN dim (range values)

.. some Gifa code

ENDFOR
```

var_name will only scan the index in array_name, that fall inside the range given as argument to WITHIN.

Example given using the previous example

```
FOREACH i IN peak WITHIN 2 1 1 2 3 ; within the 2D square (1,1) to (2,3)

print $peak[$i]

ENDFOR
```

will print the value of peak[1] and peak[2] but not peak[3].

These command have been designed for the assignment module, and you will find many examples of their use in it.

SUPPORT FOR DATA-BASES

There is in *Gifa* a support for the UNIX standard data-base file format dbm. A dbm data-base is composed of two files: `base_name.pag` and `base_name.dir` which together hold the information of the base_name data-base. Each entry consist of a field indexed with an access key. Each entry is accessed with the access key. Data can be retrieve, stored, modified in the data-base. Every operations are realized on the file, but the disk accesses are minimized.
In *Gifa*, such a dbm data-base is accessed through a binding to an associative array. Each entry of the data-base appears as a element of the array. Reading, storing and modifying entries is performed by fetching and assigning array values. The access key of the entry in the dbm file appears as the index of the associative array, and the content appears as the value of the associative array. Due to internal *Gifa* limitation, only keys up to 31 characters and entries up to 256 characters can be handled.

DBOPEN, DBCLOSE

These two commands permits to associate the dbm file with a pseudo internal array variable. The command

```
dbopen array_name file_name
```

will associate a pseudo associative array with the dbm files of base name : *file_name*. The file is created if it does not exist yet. No actual variable *array_name* is created, but each operation on it is performed on the dbm file.
`DBCLOSE` closes the file and forget about the array *array_name* .
The functions `dbm()` and `nextlm()` permits to handle the dbm file in macro processing. The control `FOREACH` permits to scan easily through all the entries of a dbm file.
Let's give an example :

```
dbopen array file_db                    ; associates the file
set array['dim'] = $dim          ; store entries (associative array)
set array['size_1d'] = $si1_1d
dbclose array                           ; and closes the file
```

after closing `array[]` is not available anymore (it was here as a tool to writing into the file `file_db`. However, values have been stored into the file, so next time you will `DBOPEN` it (eventually binding an array with a different name this time), they will be there again.

```
dbopen next_time file_db
print $next_time['dim']
dbclose next_time
```

ENVIRONMENT

The GifaPath, SETPATH, SETPROMPT

When calling a macro, the file is first searched in the local working directory, and then is searched in a series of directories called the GifaPath. The default search order in the GifaPath is: the macro sub-directory of the home directory of the user (`SYS$LOGIN[macro]` in VMS; `$HOME/macro` in UNIX) and finally the standard directory : `Gifa$MACRO` in VMS or `/usr/local/gifa/macro` in UNIX. However the GifaPath can be modified at will (shortened or extended) with the `SETPATH` command. The current GifaPath can be examined with the `$GIFAPATH` context.

This is reminiscent with the notion of *path* in UNIX. This permits to build a library of macros, available for every *Gifa* user, as well as a set of commands personal to each user and to each project. The local directory is always searched first, so it is useless to add "." in the GifaPath.

We have seen above that the `SETPATH` permits to modify the path used for searching macro files. There is also a `SETPROMPT` command, which permits to modify the prompt which is given to the user in the text window. Together with the macro and graphic capabilities, this permits to build easily some applications, which does not even have to related with NMR.

The startup.g macro

When the *Gifa* program is first entered, it tries to execute the macro file called : `startup.g`, the file is searched in the GifaPath as any regular macro. Thus you can easily set-up you own environment. For instance your startup.g file (in your home directory, or in your working directory) may look like :

```
...any personal set-up

disp1d 1

/usr/local/gifa/macro/startup.g ; to be sure to have general set-up
```

LOG_FILE CONNECT DISCONNECT

In permanence, all the input generated by the user, and the output generated by the program are journaled in a file called gifa.log. This file can be either kept or removed when exiting the program. At any time the user may redirect the journaling to another file with the command : `CONNECT` file_name. The program will then create a file called file_name where all the journaling will go. The command `DISCONNECT` will resume the journaling to the gifa.log default file, and close the previously `CONNECTed` file. If the command `CONNECT` is issued while a file is already `CONNECTed`, the former file will closed, and the new file will be opened.

PARAMETER PASSING AND INTERACTIVITY

PRINT, alert, MESSAGE

`PRINT` *text* will display the content of *text* to the user. A formatted print is also available (`printf`) see below.

`alert` *text* displays the text to user in a graphic box. If no graphic mode is on, `alert` will issue a string on the terminal.

`MESSAGE` is used in macro, it is equivalent to `PRINT`, except that the string will be output to the user only if no parameters are available on the call line. It is thus nearly equivalent to

```
if (!$arg) PRINT text
```

However, it is different in the sense that the string will be presented in the graphic dialogue box when the macro is called from a menu button.

Formatted output, the printf, fprintf, sprintf macros

There is no direct support for formatted output in *Gifa*, however, three macros have been written to implement this facility : `printf, fprintf`, and `sprintf`. They are called by giving first a format, then a list of parameters (finished with a *). `fprintf` has an additive parameter which is the name of the file (which must have been `OPENed` first). `sprint` puts the result into a static variables called `$returned`.

e.g. :

```
printf "Size of the data-set \t%d x %d\n" $si1_2d $si2_2d *
```

These macros are implemented by calling *'awk'* (a UNIX facility) with the `SH` command. This implies that : i) they are a bit slow; ii) they may fail (`SH` fails when memory is low on the machine); iii) errors in formats will be detected by *awk*, not by *Gifa* (*nawk* has much better error messages that the old *awk*); iv) type *man awk* to get information on the available formats.

MESSAGE

This command permits to build macros which have exactly the same behaviour than built-in commands. The macro `example`

```
message "Enter new size" set newsize := $_
```

will react as follow :

nothing will appear if called with a parameter :

```
example 256
```

the user will be prompted if called without parameters :

```
example
```

the user will be prompted in a dialogue box if the macro is called without parameters from a menu button.

In any case, this is independent of the way the macro is actually called, either directly or from within another macro.

Interactive macros

During macro execution, it is possible to get input from the user in several ways :

• if a command misses a parameter, the user will be prompt for it, even if the macro is called deep in a nested loop. The user will be prompted with the current value. e.g.

```
print 'Enter new size' chsize
```

(actually the message is not really needed, since the command `CHSIZE` will issue one)

If the macro is called from the graphic interface (with `BUTTONBOX` or `FORMBOX`), either directly or indirectly called, a dialogue box will be used to prompt the user for the value, the message from the command (here chsize) will be in the dialogue box.

• by the same token, it is possible to assign the input into a variable :

```
set b = 10 print 'Enter new value' set b =
```

The user here will be prompted with the value 10 as default for b.

The command `MESSAGE` permits to put the message in the dialogue box if the macro is called from a menu button, and on the terminal if called from the prompt. Thus this is a better construct :

```
set b = 10 message 'Enter new value' set b =
```

• It is possible to get mouse coordinates with the `MONOPOINT` commands and the `$POINTX[]` and `$POINTY[]` variables. The `alert` macro permits to build simple alert boxes, and `BUTTONBOX` and `FORMBOX` permit to build more sophisticate graphic interfaces.

Passing parameters to macros

Parameters can be sent to macros when called by following the call with the parameters to pass; within the macro file, these parameters will be obtain with the pseudo variable $_. each use of $_ "eats up" one parameter on the call line. If no parameter is available on the calling line, the $_ pseudo variable will prompt the user directly. The variable $arg permits to test the presence of a value obtainable with $_.

For instance, if the file test_arg is as follow :

```
; to test argument passing

print $_

set b = $_ print (2*$b)

set c = $_
```

the following call

```
test_arg hello 3 $SI1_2D
```

will produce the following output

```
hello

6
```

and will prompt for the value to be applied to c. Thus $_ can be used to both the command line or the user, much in the same way as regular *Gifa* commands do. It is even possible to have an optional message, depending whether there is a parameter available or not, with the variable $arg :

```
if (!$arg) print 'Enter new value'

set b = $_
```

The MESSAGE command, has a similar built-in mechanism, the string is sent to the user only if no parameters are available on the calling line. It is better to use the MESSAGE command, since the message will then go to a dialogue box if the macro is called from the graphic interface.

```
message 'Enter new value'

set b = $_
```

Returning values to the caller : RETURN

The RETURN command sets the internal context $RETURNED to the value given to the command. You can return any kind of values, however only one value can be returned, so you will need to cancatenate values in a string to return several independent pieces of information.

Note that before version 4.4 another technique was used, using a global variable called $returned. This technique cannot be used anymore, as it clashes with the current RETURN technique which is used throughout the set of standard macros.

TEXT FILE INPUT / OUTPUT

Gifa has the capability of reading and writing on ASCII text files.

OPEN CLOSE FPRINT <file_name eof(file_name)

It is possible to read and write directly into text files. The command `OPEN` file_name, will open the file file_name for input or output. The command `CLOSE` will close the file. Ten different such files may be opened at once. An `OPENed` file may be written with the command `FPRINT` file_name output. A `OPENed` file may be read with the `<file` syntax. When `OPEN` is issued on an already `OPENed` file, the file is rewound to the first line. The `eof(file_name)` function (see below) will have a value of 1 if the end of file was reached during the last `<file_name` operation and 0 otherwise.

GRAPHIC PROGRAMMING & GRAPHIC INTERACTIVITY

REFMACRO

`REFMACRO (0 or 1)`

depending on the value of this context, display will be refreshed at the end of each macro line (REFMACRO 1) as for regular processing; or only at the end of the macro execution (REFMACRO 0). Default value is 0; a value of 1 is useful only for interactive macros, or for debugging.

REF, UNREF

The 1D and 2D display are refreshed whenever the data are changed (or when a display parameter has been changed (SCALE for instance)), `REF` and `UNREF` can force this behaviour. `REF` sets the internal *Gifa* flag, telling it to refresh the display, `UNREF` reset this flag such that no refresh will take place. This flag is tested by *Gifa* internally at the end of each command line, so `REF` and `UNREF` are usually the last commands on a line.

For computation, display and data processing

SETVAL, SETPEAK, SETPEAK2

```
SETVAL i {j { k } } val
```

will set the value of the ith point in the current buffer to val. The number of parameters for determining the point to be set depends on the value of DIM.

Similarly SETPEAK and SETPEAK2 permit to change the coordinate of a peak table entry.

SIMU, SIMUN, ONE, ZERO

SIMU simulates a complete experiment as a set of exponentially decaying sines plus noise in 1, 2 or 3D. Noise can also be added to the simulation

SIMUN is different, it permits to simulate only one line, but this line is added to the current data-set, with all the current parameters (spectral width, sizes, etc...). The command ADDNOISE permits to add a Gaussian noise to the current data-set.

The commands ONE and ZERO put respectively the value 1.0 and 0.0 in the current buffer. Very useful for initialising a data-set before SIMUN, or for visualising an apodisation function.

SHOWLINE, SHOWTEXT, PLOTLINES, PLOTTEXTS, PLOTLINE, PLOTTEXT

are commands that permit to draw a line (xxxLINE) or to write a string (xxxTEXT) on the screen (SHOWxxx) or on the plotter (PLOTxxx). The first 4 commands take coordinates in index on the current data-set. They will both draw according to the zoom state. SHOWxxx commands will use the value of the context SCOLOR as the colour. PLOTxxx command will also depend on CX and CY. The 2 last commands take coordinates in cm.

SHOWLINETAB

Is different from the precedent command as it uses the content of the TAB[] and working 1D buffers for rapidely drawing a set of vectors. TAB[] is used as the X coordinates (interpreted in various unit)

and the working 1D buffer is used for Y coordinates. The number of point to draw, and the precise location on the drawing on screen have to be precised.

GRAPHIC USER INTERFACE

It is possible to modify and adapt the user interface within the *Gifa* program. Actually, the graphic user interface (GUI) you are used to when using *Gifa* is completely built in the macro langage. Everything is built from a simple set of basic commands which permits to build menus, dialogue boxes, etc...

To write your own user interface, you should understand first some standard concepts commun to all GUI environment. There is a quite small number of primitive command for building GUI element, each refer to quite different type of graphic. However, each of these command accept a very large range of parameters. The basic commands are : BUTTONBOX for building menu bars, DIALOGBOC and FORMBOX for building forms to be filled. All theese GUI commands handle several piece of data which have to be differenciated.

1) Each graphic element (a menu, a dialog box) has a name, this name is a character string, it usually appears as the name of the window in which it is displayed. 2) The GUI elements are designed to create actions from a user generated event (typically a click), this action is here simply a *Gifa* command (what else could it be). This action is called the 'callback' of the GUI element. 3) The GUI elements are built from a library of standard widjets. A widjet is that kind of graphic element every user understand its use when seeing it (an entry in a menu, a pop-up menu, an entry field, etc...) The kind of widjet available depends on the kind of GUI comman you are using. 4) The GUI is built by the execution of a macro, but the callback of the GUI will also execute a macro. This is knd of confusing the first time, even more if you thank that tha callback macro can very well build a new GUI element.... To clear this up a little, we will refer to buildtime (when interpreting the macro that build the GUI) and to runtime (when the built GUI calls a macro through the callback technique for further action)

BUTTONBOX, PULLDOWNMENU, CLOSEBUTTON,

The command `BUTTONBOX` creates a new entry in the menu bar. It creates the menu bar if called for the first time. Parameters of the command are the name of the menu as it appears in the menu bar, then a list of entries consisting of the name of the button and the *Gifa* command associated to it. The lists end with a *. The special entry *separator* does not set a button but a separator in the box.

Any legal command can be used to be associated with a given button, a built-in command as well as a macro, and will be executed as if entered form the terminal. So the `WHILE, FOR, IF .. THEN, GOTO` commands are not available but the `IF` .. *any_command* syntax is valid. The action of the command in independent from the way it is called, except for the user prompting for values, which is performed with dialogue boxes in the case of a command called from a button. Example given :

```
BUTTONBOX "my first menu" \
```

```
Hello  "print 'Hello World'" \
separator \
Dim dim \
"Test Dim" "if ($dim == 2) print 'We are in 2D'" \
Back ("read" ; $name) \
Reread  "read $name" \
*
```

Which gives :

Note :

- **How the command should be on a single line, but how the continuation sign (\) can be used to make a very long command**
- **How the quotes are needed only to isolate field with blanks in it, for button names as well as commands (`Hello Dim`)**
- **How single and double quotes can be mixed to build composite commands (`Hello`)**
- **How the on-line IF can be used (`Test Dim`)**
- **How evaluated expression are evaluated when building the menu if not within quotes : `Back` reads the file which was last read when building the menu (the expression is evaluated when building the menu), `Reread` reads the file which is last read at the time the command is executed (`Back Reread`)**

Depending o the state of `PULLDOWNMENU`, the menus that are created will appear as static button boxes (as shown above) or as regular pull down menus. The choice cannot be changed while there is a menu bar already opened, but it is possible to close the current menu bar, to change mode, and to rebuild a new menu bar.
The command `CLOSEBUTTON` closes the menu bar and all the associated menus. It is equivalent to closing the menu bar from the close box.

FORMBOX, DIALOGBOX, CLOSEWIDGET

The first two commands are closely related.

`DIALOGBOX` permits to build sophisticated dialogue boxes, several fields can be built into the dialogue box, that the user has to fill-up. Each filed is internally associated to a *Gifa* variable, that the remain of the macro processes. The command appears as follow :

```
DIALOGBOX [series of entry] *
```

Each entry consists of a name, that appears as such in the dialogue box, and of a type for the field. Types can be : **message, text, string, int, real, file, enum, multienum, cursor**.

message type consists of a string that is displayed on the form and permits to put text in it. **text** permits to display the content of a text file, in a scrollable sub-window.

The others type of field are associated to a variable which can be modified by the user, and can be used in the following of the macro.

string, int, real and **file** have two other entries which give the name of the associated variable and its default value. These types correspond to editable fields that the user has to fill-up. **string, int, real** correspond to string, real values or integer values respectively; **file** corresponds to a file-name entry, and presents a small additional button in the dialogue box, which permits to pop-up a standard Motif file selection box.

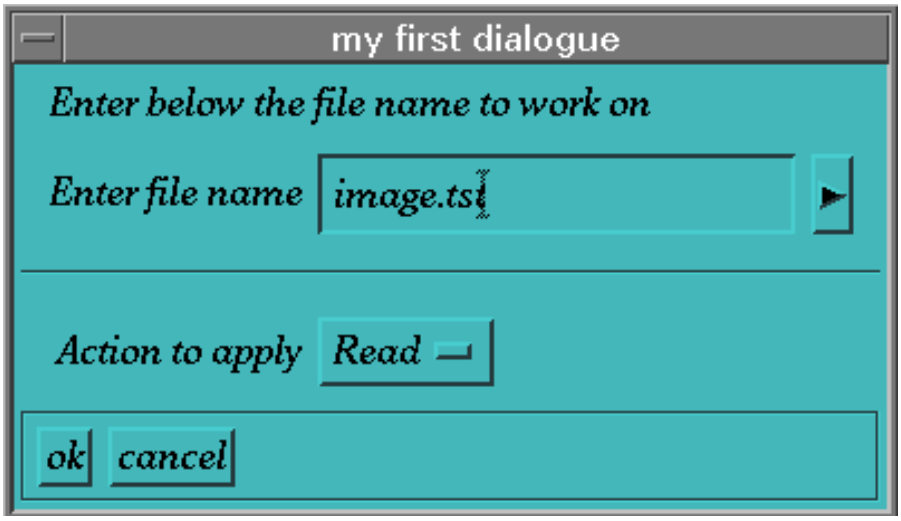
enum realises a pop-up menu, for an enumerated choice. It needs an additional entry which described the list of choice.

multienum uses the same syntax than enum, but permits a selection within the list of several entries.

The last type **cursor** realises a sliding cursor, permitting to enter a numeric variable. Compared with other entries, it has 3 additional field : number of decimal points, starting value, end value.

The special entries **separator** and **noreturn** are meant for formatting the Dialogbox. **separator** draw a thin horizontal line in the box, **noreturn** indicates that the previous and the following field will be on the same line.
Example given :

```
Dialogbox "my first dialogue" \
        "Enter below the file name to work on" message \
        "Enter file name" file var_file $name \
        separator \
        "Action to apply" enum "Read,Write" var_act % \
        *
$var_act $var_file
```



This macro will build a dialogue box with 2 editable entries : (filename and action) and will apply the action to the selected file if the user clicks on Ok

Note :

- **How the command should be on a single line, but how the continuation sign (\) can be used**
- **The separator special field which builds a thin line in the dialogue box**
- **How the enumerated list is comma separated**
- **How the default value can be anything, here a *Gifa* internal variable ($name), and the default prompted value (%). In this last case, the default prompted value will be the value of the variable itself if the variable exists before the dialogue is built. If this is not the case, it will be the first entry for** enum**, 0 or** real **and** int **and the empty string for** string **and** file.

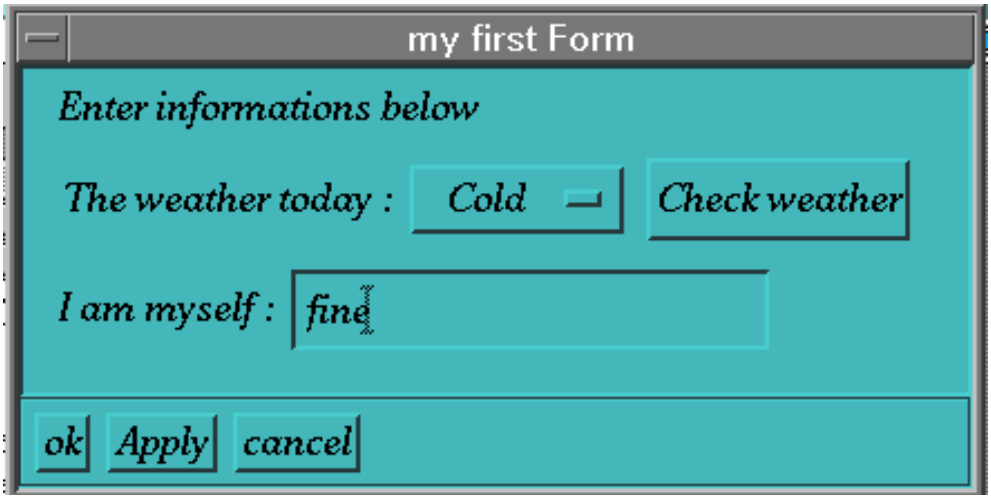
- **How the returned values are usual variables, and can be used for anything (here even as *Gifa* commands)**

FORMBOX is the static version of DIALOGBOX. It builds a form that will remain after the completion of the command (as BUTTONBOX does) and will survive as long as the user does not explicitly closes it. FORMBOX need an additional field (the callback) which describes the *Gifa* command to be executed when the user clicks on the Ok or Apply buttons. Another difference is that FORMBOX can accept an additional field type : **action** which takes its following argument as a *Gifa* command, and binds it to a new button.

Apart from this, the definition of FORMBOX is strictly equivalent to that of DIALOGBOX. If not global, a variable associated to a field in a FORMBOX is local to the FORMBOX, and cannot be accessed in any other macros but the in callback line.

Example given :

```
Formbox "my first Form" \
        "print ('today it is';$the_wet;', but I am';$myself) " \
        "Enter information below" message \
        "The weather today :" \
        enum "Sunny,Rainy,Cold,Stormy,Terrible" the_wet % \
        noreturn \
        "Check weather" action "if ($the_wet s! 'Sunny') print 'BAD !'" \
        "I am myself :" string myself "fine" \
        *
```



Which builds the following form :

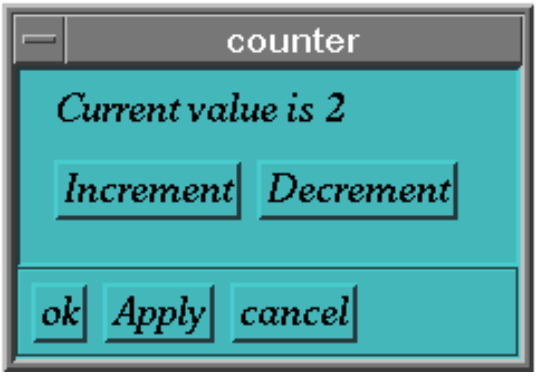
Clicking on *Check weather* produces : BAD !
Clicking on *Apply* or *Ok*, produces : today it is Cold , but I am fine
Note :

- **How the form variables can be used in the callback, in expression, as parameters for other commands**
- **the use of the** noreturn **field and of the** action **field type**

Some time you want to close a Formbox from within a macro executing within the Formbox! The CLOSEWIDGET command is here for you. From within a from, the id of the form can be accessed with the $WIDGET context. It is then used as argument by CLOSEWIDGET to close the form.
Example given :

```
filename : do_count
; $nct is global variable holding some value
; this text is in a file called do_count
```

```
Formbox counter \
  DO_NOTHING \                                          ; this is a dummy macro
  ('Current value is'; $nct) message \
  'Increment' action  'set nct := (%+1) closewidget $widget do_count' \
  noreturn \
  'Decrement' action  'set nct := (%-1) closewidget $widget do_count' \
  *
```



Then this macro builds a form like this :

Clicking on Increment increments the global variable, and closes the old form and creates a new one with the updated value.
Note :

- **How a composite action can be built within quotes**
- **That in this case, no action is associated to the `Ok / Apply` buttons (which cannot be removed, unfortunately.**

INITINPROGRESS, INPROGRESS

These 2 commands permits to build a bar showing the progress of the current command:

or as a text bar of the form:

```
In Progress : 0%....25%....50%....75%....100%
```

with a dot every 1/2Oth of the process, if the program is used in a non-graphic mode.
`INITINPROGRESS` tells *Gifa* how many iterations are to go, `INPROGRESS` actually does the display. `INPROGRESS` can be called at any rate, there is no need to do arithmetic to call `INPROGRESS`.
Example :

```
set max = 1000
initinprogress $max
for i = 1 to $max
   ... do something
   inprogress $i
endfor
```

CUSTOMIZING THE INTERFACE

One of the benefit of having a graphic user interface built from a set of macro, is that you can customise the interface to more closely fit your needs. This can be doe in two manners: modifying the standard behaviour by changing the standard macro; adding new menu entries.

Modifying macros

Most of the high level actions performed by the program are actually macros. These macro are located in the `/usr/local/gifa/macro` directory, and the subdirectories therein.

You can adapt these macro to more closely fit your needs. But rather than actually changing one (and you would have to change it at each new release of the software), it is a better idea to copy it in another directory, with the same name and to modify it. Then, if you add the directory in which you put this new macro into the GifaPath, it will be used instead of the standard one.

For instance, when using the standard Fourier Transforms, Varian users observe that there spectra appears reversed, up-side down and left-right. This is simply corected with the REVERSE command. However, Varian users would of course prefer to have this command automatically inserted in all the standard processing. This has been done in a copy of all the FT macro, as they can be found in the `/usr/local/gifa/macro/varian` directory. Also the easy2d macro as been modified to directly accept VNMR data files.

Of course, many other examples are possible.

When calling a macro, it is searched in in the GifaPath, as set with the SETPATH command, and examined with the `$GIFAPATH` context (see above). Default value for GifaPath is (working directory "." is always searched first):

`$HOME/macro` `/usr/local/gifa/macro`

The GifaPath is handled left to right, so a good place were to put your modified macro is in your own `$HOME/macro` directory.

With the Varian example above, you probably don't want to copy the Varian macro into your own macro directory, but to transparently use the Varian macro instead of the regular one, it is enough to have the `/usr/local/gifa/macro/varian` before `/usr/local/gifa/macro` (where the standard macros are) in the GifaPath.

so saying:

`SETPATH ('/usr/local/gifa/macro/varian'; $GIFAPATH)`

will set the GifaPath to :

`/usr/local/gifa/macro/varian $HOME/macro /usr/local/gifa/macro`

and will give you the Varian behaviour (the `Varian` macro does exactly this).

startup.g

A good place where to put this kind of change in the `startup.g` macro which is called by *Gifa* when starting. When thus macro is called, the GifaPath is already initialised, so the `startup.g` macro is searched first in the working directory, then in your personnal macro directory, and finally in the general directory.

If you create a startup.g macro in your own macro directory, it is a good idea to also call the standard startup.g file. Your own startup.g macro could be like:

```
plotter HPGL

cx 50 cy 50

@/usr/local/gifa/macro/startup.g ; force the standard by giving the full name

env_plot.g ; add the plot menu by default

menubox 'my menu' .... ; put here your own menu
```

If you want that the Varian (sticking with this example) is the default working mode for all your *Gifa* users, modify the standard startup.g file by putting the `SETPATH` command shown above.

MISCELLANEOUS COMMANDS, NOT SPECIFIC BUT USEFUL IN MACROS

For control and interaction

EXIT, QUIT, or BYE

These commands, when used within a macro, simply exit the macro, as a normal exit would do.

VERBOSE and DEBUG

These 2 contexts will generate verbose output from some module in *Gifa*. For instance `VERBOSE` will detail the processing of Maximum Entropy run, of macro files, of baseline correction, of fitter evaluations etc...

In DEBUG mode, the name of each macro is typed on the terminal when the macro is called. This may be very useful when searching for macros called by a graphic interface. In DEBUG mode some additional information is also given such as more detailed error codes, etc...

SH, Unix calls

The SH command will send its parameter to the operating system. For instance you can type :

```
Gifa> sh "dir [marc...]*.ser " ; on VMS machine
```

or

```
Gifa> sh "render upper.uis" ; on VMS machine
```

or

```
Gifa> sh "ls -l /usr/data" ; on UNIX machine
```

You can also type SH alone, this has the effect of creating a sub process at the operating system level (csh is used on a UNIX machine), you then get back to *Gifa* by loging out (logout on VMS, ^D on UNIX).

Using SH, several macros have been created that mimic UNIX : more, rm, ls, vi, pwd, etc... There are also two special editing command : `vip` will edit in your $HOME/macro directory, and `vim` in /usr/local/gifa/macro

Note that the function *sh()* (new with version 4.4) permits to call the operating system and to get the value returned by the command.

CD

`CD` *`dir`* permits to change the current working directory to *`dir`* . Equivalent to UNIX. Note that this is very different from :

```
sh 'cd dir'
```

which actually creates a sub-process which executes cd, thus having no effect on the current job.

Error Handling

When an error is encountered, the program stops the execution and resumes to the user with a specific message. If the error happens during the execution of a string of commands on a single line, the string is aborted. If it happens during a macro, the macro is aborted.

Typical error conditions are :

- **unknown command**
- **wrong parameter for a command (e.g. FT F3 in 2D; SIN 1; FT on real data, etc..).**
- **A impossible READx or WRITEx is tried (e.g. READ DATA.001 and DATA.001 does not exist).**
- **mistyping a parameter (e.g. EM 0,5 instead of 0.5).**
- **a ^C was typed by the user.**
- **an ERROR command was issued.**
- **a syntax error was detected within a () expression, in this case, a graphic pointers show were the error might be**
- **an invlaid string was interpreted as a number**
- **a variable which does not exists was used.**

If the error occurs when the command was executed from a menu button, a alert box will be displayed.

ERROR

This command generates an error during the execution of the macro. The string given as argument is used to notify the user.

ONERRORGOTO

This command modifies the standard error handling. The command requires an argument, which is a label within the current macro. After it has been issued, if an error is generated, the execution of the macro will resume to this label rather than stopping the macro. It thus permits a gracefull handling of the error condition.

Once the error condition has been trapped, the `ONERRORGOTO` state is lost, and must be reset with a new instance of the command. Note also that the condition is strictly local to the macro. If an error happens in a called macro, this second macro will stop as normal, and the error will be handle in the caller.

Code optimisation

Code optimization can be realized at several levels.

First, for data processing, always choose the commands realizing the processing on the larger possible piece of data, as this is usually realized in the processing kernel rather than in macro. For instance, if you want to add two 2D files (let say) it will be much more efficient to add the 2D at once, rather than adding row by row.

You also have to know that control structures are somehow compiled when processed the first time, and the result of this compilation is used for the following passes. This is true for the `FOR .. ENFOR ; WHILE .. ENDWHILE ; IF..ENDIF` structures, as well as for the `GOTO`.

As a matter of speed optimization, you have to know that macro calling is lengthy, as that your code will be faster if you manage to put every thing in a given macro; also notice that calling a macro explicitely with @, and with its full path (`@/usr/local/gifa/macro/speed`) is probably faster than using the implicit call : `speed`.

TIMER

The command `TIMER` permits the measure of the CPU time spent for a given command. When `TIMER` is set to 1, each command issued at the console, or from the Graphic user interface is timed. Timer is returned as the CPU time used for the process, the system time (system calls) used and the sum of both.

If TIMER is reset to 1 during a macro execution, the value is issued and time counting is reset to zero.

PROFILER profile

The PROFILER command tells you how much time each executed command takes. When set to 1 during macro processing, each macro line is timed, and the times are cumulated, permitting to profile exactly the execution of lengthy macro. Look to PROFILER help for details.

Alternatively, you may use the profile macro, which uses PROFILER, and gives you a complete analysis of the execution times. Simply type

```
profile macro_name
```

Batch Mode

It is perfectly possible to create files in order to run the program in background.

In VMS

Here is the structure of such a COM file (VMS):

```
$ gifa ! this call gifa from the VMS level

dim 2 ; now you just type the commands the way you would do

; in normal interactive processing

read [marc.data]stuff.ser

revf f2 sin .3 f12 ; you can link command on one line

; or span on several lines (different from macros)

rft
```

```
f12

@baseline ; even call macros.

write [marc.data]stuff.smx

exit y ; DO NOT forget the exit at the end.

$ dir [marc.data]stuff.*/siz ! we're back to VMS.

$ exit ! end of the run.
```

You can execute such a macro either in a pseudo-interactive mode by typing

```
$ @name_of_the_COM_file
```

or in batch mode by submitting it to a batch queue :

```
$ submit name_of_the_COM_file
```

With this last technic the file is executed in background, and you do not have to wait for it to be finished to log out of the system.

In UNIX

On UNIX machines, type a file containing the command as you would type them and just start *Gifa* by "piping" the file as standard input, and the output on another file. The & sign run the process in background :

```
% gifa < name_of_the_COM_file > process.out &
```

DO NOT forget to end the input file with the final ' EXIT N ' which is the end of a normal interactive session (or 'EXIT Y' if you wish to keep the log file). Also you may experience problems if you use interactive shell commands such as 'vi' or even 'more'.

On All Systems

The input in the batch file is entered to the program as a user would do, so it is different from macro in at least to points : • control structures are not allowed • error conditions will not stop the execution, but the program will continue, So be careful when deleting important files within batch files!

Of course any macro can be called during the batch process, without any restriction on control structures.

When running in batch mode it is usually useless (though not forbidden) to activate the display. However if you want to have *Gifa* running while you are not logged on, you should choose a graphic-less mode. This mode is entered if there is no X_Windows DISPLAY available. For instance with

```
unsetenv DISPLAY
```

Do not forget to adapt a special startup.g macro (for instance in the current directory) in order not to launch the standard graphic environment (which would give an error anyway).

It is perfectly possible to plot and print results while in batch mode.

EXAMPLES :

Basic examples

The PRINT command permits to output to the user a string. If you want to show more than a single line you can always use sh :

```
sh 'ty/page text_file' ; VMS or

sh 'more text_file' ; UNIX
```

As an example the 2D Fourier Transform could be written

```
; do the 2D FT in macro (specially silly),

if ($dim != 2) error 'Only in 2D' ; do some checking

if ($itype_2d != 3) error 'Only on hypercomplex data-sets'

if ($si1_2d*$si2_2d) > 1024k error 'too big for 4 times z-f'


print 'Processing in F2'

chsize % (2*power2(%)) ; zero-filling in F2

initinprogress $si1_2d ; set up in progress bar

for i = 1 to $si1_2d ; loop over rows

row $i dim 1 ; get ith row in 1D buffer

ft ; process it

dim 2 put row $i ; put it back

inprogress $i ; display in progress bar

endfor


print 'Processing in F1'
```

```
chsize (2*power2(%)) % ; zero-filling in F1

initinprogress $si2_2d

for i = 1 to $si2_2d ; loop over cols

col $i dim 1 ; get ith col in 1D buffer

ft ; process it

dim 2 put col $i ; put it back

inprogress $i

endfor

exit
```

but don't try that one for real processing, it would be really slow!

A more useful example :

```
; do contour plotting in colours, 1 colour per level

; using scale, level and loga

if ($level>8) error 'Too many levels, should be <= 8'

set loc_lev = $level level 1

set loc_sca = $scale

for i = 1 to $level

pen $i % plot %

if ($loga!=1) scale (%/$loga)

if ($loga==1) scale ( (%*$i) / ($i+1))

endfor
```

```
pen 1 % page %

scale $loc_sca level $loc_lev

exit
```

Another example showing how you can build a special apodisation function in the window buffer and then use it :

```
; this macro builds a cosine roll-off apodisation function

; and applies it to the current data set

; works in 1D !

; in 2D, build the WINDOW only once and loop APPLY WINDOW on the 2D

put data ; keep data aside

chsize (%/8) one sqsin 0

chsize (%*8) addbase 1 mult -1 reverse ; build apodisation

put window

get data apply window ; apply apodisation
```

Here is a small (silly) interactive example :

```
; to compute the square root of a given number

sh 'more intro.txt' ; some greeting message'

set i = 1 ; i should be set here for the following test to work!
```

```
while ($i <> 0)

print 'Enter your number (0 to end)'

set i = 0 ; pre-set the value and ask for it

set i =

if ($i>0) then

print ('Square root of';$i;'is';sqrt($i))

elsif ($i<0) then

print 'No real square root !'

endif

endwhile

exit
```

A more interesting example is found in the gif directory. The file GIFA$GIF:MENU.GIF (or /usr/local/gifa/gif/menu.gif on UNIX) is a working example (in French) of a menu interface put on the top of GIFA command interface. This program is used by the chemists in our lab. (who know nothing about GIFA (not even about NMR (I'm not sure about chemistry))) as an every day tool for processing NMR files off the spectrometer.

For speed consideration it is always better, when working on a large macro, to make a small file for a very repetitive piece of work and call it. The looping as well as the parsing may get slow on a large file. It is also slightly faster to call a macro with the @file syntax.

comments, automatic documentation

Comments are created with a semi-colon. The comment can be alone on the line, are can appear after some code.

One important feature is the automatic documentation facility built in *Gifa*. All the leadig comments are used as an interntal help file. The help system uses this feature, so that, as soon as a macro is created, the `help` command will describe this macro, using the leading comments. The `apropos` command will also use this leading comment when searching for a given word in all the availabe macros.

You will find examples of this comment techniques throughout the examples given in this manual.

$_ and argument passing

Arguments can be passed to macros when calling them, and value can be returned from them. You will find many examples of this in all the macros in *Gifa*. The basic mechanism for passing argument is the context (pseudo variable) $_ . This one takes the value of the next entry present on the calling line.

file: simplest

```
; this one print all the values present on the calling line.

while ($arg)

print $_

endwhile
```

typing

```
simplest 12 (3*3) $SI1_1D
```

will produce

```
12

9

256
```

You see here how $_ takes successively the values on the calling line, and how variables and expressions are evaluated. Note how the `$arg` variable is true as long as a value is available on the line.

For returning values from a macro, see next example.

alternative cases - macro returning values to the callers

There is no `switch .. case` control structure in *Gifa*, however the `elsif` structure can be used for the same purpose. We also see here how a static variable can be used to return values to the caller.

filename : percentile

```
; percentile nb_degre_of_freedom
;
; gives the 95% confidence limit on Chi2


set n = $_
if ( abs($n - int($n)) > e-6 ) error "nb_degre_of_freedom should be integer"
if ($n == 1) then
set val = 3.84
elsif ($n == 2) then
set val = 5.99
elsif ($n == 3) then
set val = 7.81
elsif ($n == 4) then
set val = 9.49
elsif ($n == 5) then
set val = 11.1
elsif ($n == 6) then
set val = 12.6
elsif ($n == 7) then
```

```
set val = 14.1

elsif ($n == 8) then

set val = 15.5

elsif ($n == 9) then

set val = 16.9

elsif ($n == 10) then

set val = 18.3

else

set val = ($n*(1-2/(9*$n)+1.6449*sqrt(2/(9*$n)))^3)

endif

return $val
```

Here the percentile macro, which returns the 95% confidence limit on Khi2 given a number of degrees of freedom (used to check whether a given statistic actually follows the Khi2 law). The number of degrees of freedom n, is always an integer, and the law is a complex, non analytic expression, however, for n larger than 10, a good enough approximation exists, that we will use. The following structure :

```
if ($n == 1) then

set val = 3.84

elsif ($n == 2) then

set val = 5.99

elsif ($n == 3) then

...
```

is equivalent to using a a switch / case structure as we would do in C. There is actually no limit on the number of elsif that can be accumulated this way. Note that the correct spelling is `elsif`, not `else if` which works also, but requires an additionnal closing `endif`, on the other hand `elseif` (single word) has no meaning.

The macro returns the value by using the RETURN command, the value of which is found by the caller in a context call $RETURNED. Note that before version 4.4 another technique was used, using a global variable called $returned. This technique cannot be used anymore, as it clashes with the current RETURN technique which is used throughout the set of standard macros.

Building string - mixing ' and "

There are two different quotation marks in *Gifa* : " (double quote) and ' (single quote). There are strictly equivalent, and induce NO difference whatsoever in the parsing. However it is quite usefull to have both, as you can mix them in complex expression. This replaces the escape mechanism (\") which is lacking in *Gifa*. For instance :

```
("I'm"; '"quite" sure of what I am saying.')
```

gives

I'm "quite" sure of what I am saying.

Thus mixing both quotes, and using the ; which is the concatenation operator (inserting a blank).

This is something you have to play with quite heavily, when you're building strings that have to interperted latter on as a *Gifa* command.

interaction with the shell

It is very easy to intact with the underlying POSIX unix shell. The `sh` command and the sh() function call the (C) shell for you with the string argument given.

filename : readgz

```
; readgz filename

;

; reads a compressed regular Gifa file

;
```

```
; see also : writegz WRITEZ ZEROING


message 'Enter file name (without .gz extension)'

set f = $_

set i = (index($f,'.gz'))

if ($i != 0) set f = (subst($f,1,$i-1))

sh ('gunzip -c '; $f // '.gz >';$f)

read $f

sh ('/bin/rm';$f )

set compress_ratio = (sh("gzip -l " // $f // ".gz | awk '/%/ {print $3 ; }'") )

; this command extract the compress ratio as given by gzip -l

print ("Compression :"; $compress_ratio)


```

This example shows how the `gzip / gunzip` unix commands can be used to directly read from compressed files. The macro first removes the .gz if one was given by mistake, then call `sh` by providing it with a string which calls gunzip to uncompress the file to another dummy file which is then read. If the file is data, `sh` is called with the `'gunzip -c data.gz > data'` string. Note than we use here `/bin/rm` rather that `rm` to remove the uncompressed file; this is because many people have aliased rm to some expression which prompts the user to confirm the file removal, using /bin/rm insures that this alias is not used.

Finally, note how the sh() function can be used equivalently to call the shell and use the returned value in the macro itself.

list parsing with head() and tail()

There is no list data structure *per se* in *Gifa*, However, strings can be used as simple lists, along with

the `head()` and `tail()` function.

The idea is to cut the string into pieces. `head()` takes the first entry, `tail()` gives you the remaining (this is reminiscent with `cdr` and `car` in LISP).

Here is an example extracted from the assignment module. At this stage, the list of spin Id is in a (blank separated) string called `$ll_res`; the purpose of this piece of code is to build the list of all the chemical shifts (will grow in `ppmlist`).

taken from att/mod_sys

```
... some code before

; build list of spins

set p = (head($ll_res)) set ll = (tail($ll_res))

set ppmlist = " "

while ($p s! " ")

set ppm = (head($spin[$p])) ; extract ppm value

set ppm = (int(1000*$ppm)/1000) ; truncated to 3 digits after the dot

set ppmlist = ($ppmlist//':'//$ppm) ; push onto the growing list, ":" separated

... some code removed here

set p = (head($ll)) set ll = (tail($ll))

endwhile

... some code after
```

$p contains the current head of the list (thus the spin id to process), the next entry is taken with the head($ll) command. The end of the iteration is obtained when the entry is empty. There several ways for implementing this iteration, notice how initiated the iteration by taking the first entry before entering the `while` loop, permits to simplify the loop control.

This will sound familiar to people used to LISP, note however that *Gifa* does not permits recursive

calls, so the while loop.

double use macro - the (if (!$arg)) syntax

When creating a graphic user interface, one has usually to realize two independent things : -design the graphic user interface (GUI) -makes the needed action(s). Of course, this two things are linked, and the callback in the graphic interface are there to call the actions. However, it is not optimum to have 2 different files, one for the GUI and one for the actions itself. Moreover, quite often the same piece of information have to be used in both phases, and it is a kind of waste to have them twice in two files.

A simple set-up is used in most *Gifa* macro to solve this problem. Many interactive macros have two different behaviors depending whether an argument is available on the calling line or not. If no argument is available, then a GUI is built. On the other hand, if arguments are present, there are taken to be parameters for the actions, as defined from the GUI itself, and the action is performed. Thus we end-up with a syntax like :

```
if (!$arg) then ; literally, if there is no arg available

formbox ....

....

else

set param = $_

....do some work depending on $param

endif
```

see an example of this in the next example.

related commands; command parser

In some cases, you want to handle several related commands. Examples of this are the easy1d or easy2d tools, which perform all the needed processing commands, the mutizoom or integrate tools which are able of displaying, storing, defining, etc.

You could do this by writting a set of macros realizing each one of the elementary actions (display, definition, etc...), with the inconvenience of cluttering the macro directory with many related files.

Gifa does not provide a way of putting several macros in a single files, however it is quite simple to write a unique macro, with one argument being the action to take, and writing a small command parser. Here is one example taken from multi_zoom (a bit simplified)

filename : multi_zoom

```
; creates a tool which permits to handle several zoom box on a data-set.
;
; you can define zoom regions
; jump to a given window
; store and load a set of zoom definitions (stored as macro commands)
; draw the zoom definitions on screen
;
; when loaded, the zoom definition are stored in an associative array called
; zmem[]


set fnm = "zoom_window" ; the name of the storage file, if any


if ($arg == 0) then
=redo
; =redo is used by the "define" action
; first prepare every thing for form.
set zoom_list = " "
if (exist("zmem[]")) then
set i = (nextlm("zmem"," "))
```

```
while ($i s! " ")

set zoom_list = ($i // ',' // %)

set i = (nextlm("zmem",$i))

endwhile

endif

if ($zoom_list s= "") set zoom_list = "Empty!"

formbox "multi_zoom" \

"multi_zoom Jump $area" \

"Zoom region :" enum $zoom_list area % \

noreturn Rzoom action rzoom \

Jump action "multi_zoom Jump $area" \

noreturn Define action "multi_zoom Define closeform $widget" \

noreturn Draw action "multi_zoom Draw" \

noreturn Load action "multi_zoom Load closeform $widget" \

noreturn Store action "multi_zoom Store" \

*

else

set action = $_

if ($action s= "Jump") then

set area = $_

set d = (head($zmem[$area]))

set z1 = (head(tail($zmem[$area])))

set z2 = (head(tail(tail($zmem[$area]))))
```

```
set z = (tail(tail(tail($zmem[$area]))))

if ($d s= "1D") then

dim 1 zoom 1 $z1 $z2

elsif ($d s= "2D") then

set z3 = (head($z))

set z4 = (head(tail($z)))

dim 2 zoom 1 $z1 $z2 $z3 $z4

endif

elsif ($action s= "Define") then

if ($zoom == 0) error "Should be in zoom mode"

dialogbox "Define name" \

"Give a name (no blank) to the current region" message \

"Area name :" string nm "new-name" *

; to define zoom regions, use ppm

if ($dim == 1) then

set zmem[$nm] := ("1D";itop($zoom_1d[1],1,1)//'p';itop($zoom_1d[2],1,1)//'p')

elsif ($dim == 2) then

set zmem[$nm] :=

("2D";itop($zoom_2d[1],2,1)//'p';itop($zoom_2d[2],2,2)//'p';

itop($zoom_2d[3],2,1)//'p';itop($zoom_2d[4],2,2)//'p')

endif

goto redo

elsif ($action s= "Draw") then
```

```
set i = (nextlm("zmem"," "))
while ($i s! " ")
set d = (head($zmem[$i]))
set z1 = (head(tail($zmem[$i])))
set z2 = (head(tail(tail($zmem[$i]))))
set z = (tail(tail(tail($zmem[$i]))))
if (($d s= "1D") & ($dim == 1)) then
set y = ($random*40+50)
showline $z1 0 $z1 $y
showline $z1 $y $z2 $y
showline $z2 0 $z2 $y
showtext $i $z1 $y
elsif (($d s= "2D") & ($dim == 2)) then
set z3 = (head($z))
set z4 = (head(tail($z)))
showline $z2 $z1 $z2 $z3
showline $z2 $z3 $z4 $z3
showline $z4 $z3 $z4 $z1
showline $z4 $z1 $z2 $z1
showtext $i $z2 $z1
endif
set i = (nextlm("zmem",$i))
endwhile
```

```
... see below
```

```
endif ; $action
```

```
endif ; $arg
```

In this long example you can see many programation techniques

First the double use macro, with the `if (!$arg)` technique (here used in the opposite : `if ($arg == 0)`), which permits to use the macro without arguments, to build the graphic user interface, and with arguments for realizing the several actions.

Then the command parser itself; when the macro is called with arguments, the first one is the name of the action, (the `set action = $_` line), then we enter a large test structure testing the name of the action : `Jump Define Remove Draw Store and Load` (the last 2 are in the next example), and realizing the desired action. Notice also how the number of argument is varying with the kind of action, as for instance Jump, which requires the name of the zone where to jump, but the other actions do not require additional arguments.

This is riskless, as the only way of calling the macro with arguments is to from the formbox itself, the lines :

```
Jump action "multi_zoom Jump $area" \

noreturn Define action "multi_zoom Define closeform $widget" \

noreturn Draw action "multi_zoom Draw" \

noreturn Load action "multi_zoom Load closeform $widget" \

noreturn Store action "multi_zoom Store" \
```

Notice also the goto structure (with the `=redo` label), permit to jump in the code used for drawing the formbox, and needed to redraw the box when some changes have been made (defining a new zone or load previously defined zones).

using text-files

Gifa as the full capability of writing and reading in text files. Here is another example is taken from the same file as the previous one.

```
... part extracted from the previous example

elsif ($action s= "Store") then

open $fnm

set i = (nextlm("zmem"," "))

while ($i s! " ")

fprint $fnm ("set zmem['" // $i // "'] := '" // $zmem[$i] // "'")

set i = (nextlm("zmem",$i))

endwhile

close $fnm

alert ("Written to a file called :"; $fnm)

elsif ($action s= "Load") then

print ("Loading from file '"; $fnm; "'")

@($fnm)

goto redo

...
```

Writing consists simply in opening the file, and using `fprint` to write into it (`$fnm` is defined in the header of the macro). In this case, *Gifa* code is directly written into the file, so reading the file consists simply in interpreting it as macro. Try to use this trick as often as possible, as it is much simpler to handle.

In other case, you want to read the file directly. The following example is taken from the macro which is used to read and interpret (as much as possible) the Varian parameter files.

filename : varian_param

```
; varian_param procpar_file_name
```

```
;
; this macro reads-in the procpar Varian file and sets
; the Gifa relevant parameters from the values found in the file.
; The parameters which are sets are :
; FREQ - SPECW
;


set f = 'procpar'
message 'Varian parameter file'
set f = $_
open $f
set l = <$f
print "Scanning the param file ..."
while (!eof($f))
if (index($l,'sw1') == 1) then
set l = <$f
set sw1 = (tail($l))
elsif (index($l,'sw2') == 1) then
set l = <$f
set sw2 = (tail($l))
elsif (index($l,'sw') == 1) then
set l = <$f
set sw = (tail($l))
```

```
elsif (index($l,'sfrq') == 1) then

set l = <$f

if (tail($l)!= 0) set sf = (tail($l))

elsif (index($l,'dfrq2') == 1) then

set l = <$f

if (tail($l)!= 0) set df2 = (tail($l))

elsif (index($l,'dfrq') == 1) then

set l = <$f

if (tail($l)!= 0) set df = (tail($l))

endif

set l = <$f

endwhile

close $f

if ($dim == 1) then

specw $sw

freq $sf $sf

elsif ($dim == 2) then

if (!exist('sw1')) set sw1 = $sw ; happens if it is not a 'true' 2D

specw $sw1 $sw

freq $sf $df $sf

endif
```

In a nut-shell, Varian parameter files re text files, with the name of the parameter on one line, and the

value on the next line.

Without going into the logic of the parsing, note how the file is first opened, and how we go through the file with the `while (!eof($f))` (while we have not reached the end of the file) technique.

Notice that file handling is somewhat simplified in *Gifa* compared to more complete langage : i) the file is handled by its filename only, so be VERY careful when opening a file in the current directory and then changing of working directory, ii) the open command has no switch for telling whether it is opening a file for reading or writing, if the file exists, it is opened for reading, if it does not exists it is opened for writing.

creating a temporary file

In many cases, you want to create a temporary file. The way to do in *Gifa* is to create a regular file, with a strange dummy name which minimize the risks of being the name of an important file. This is typically done in *Gifa* with the following code :

```
code for dummy file name

set file = ("Gifatmp"//int(100000*$random)) ; gives a file name like
Gifatemp234532

; or eventually

set file = ("/tmp/Gifatmp"//int(100000*$random)) ; gives a file name like
/tmp/Gifatemp234532

;use it, and remove it with :

sh ('/bin/rm';$file )
```

This technique is not perfect, as it may create twice the same filename, but is good enough for most NMR work. Note that the first technique will clutter the working directory, with the advantage of being probably a place reserved to the current user, but which might be a slow access file (if you use NFS as we do in our lab); whereas the second will clutter the /tmp directory (you may prefer to use /usr/tmp) which is local to the machine, and thus corresponds always to a fast access, but where all the users have to share the place. Take care to choose a filename explicitly dummy, so that the user will

be able to recognize it as dummy, and may remove it by hand if for reasons the processing was stopped before the completion of the rm line.

This technique is used through out the set of standard macro files, for data files as well as text file.

Building interactive commands

One easy way to interact with the user during macro execution, is to block the user with any blocking event (in the following example, alert is used, which calls DIALOGBOX). The following example is used to measure the noise level in a given spectral zone. It uses the EVALN command which does the work, and simply build the graphic user interface around the EVALN command.

filename : evaln.g

```
; graphic interface for the EVALN command.
;

... removed some irrelevant code


alert 'Select an empty area of the data-set THEN click on Ok'
if ($dim == 1) then
print ($zone[1] ; $zone[2] ; $zone[3] ; $zone[4])
evaln $zone[2] $zone[4]
else
evaln $zone[1] $zone[2] $zone[3] $zone[4]
endif
```

This macro calls the macro `alert`, which creates a window on screen with the text as an argument, and blocks the macro at this level, the macro continues when the user clicks on Ok. However it is possible to change the display and to zoom while the user interface is blocker, this is used to ask the user to select the spectral zone on which the noise is going to be computed. Finally the noise is computed on the zone defined by the $zone[...] context, which hold the last zone drawn on the spectrum with the mouse.

Note that `alert` is given in the next example.

use of internal Gifa configuration - use of blocking event to wait for the user

You have here a very simple example of using the `$CONFIG_XXX` contexts. All aspects of the internal program state are described with such contexts.

filename : alert

```
; alert "text to display"

;Creates a graphic alert box displaying the text. User must click in

;the alert box to continue.

;

;see also : PRINT ERROR DIALOGBOX


if ($CONFIG_GRAPH) then

refmacro 1

dialogbox "alert" $_ message *

refmacro 0

else

print $_
```

```
message "Type on return to continue"

set a =

endif
```

Here we see the `alert` macro, used in other macro to give a blocking alert text, the user as then to click on Ok (in graphic mode) or to type return (on batch mode). The user can also abort the processing, clicking on Cancel, or by typing ^C. There is actually 2 different programs here. First we check whether the program is running in graphic mode (`$CONFIG_GRAPH` is true) or if it is running in batch mode. In graphic mode, dialogbox does the whole work, with $_ displayed as a message in the box, in batch mode, we first print $_, then wait for the value of a dummy variable to be set.

graphic user interaction and clicking on the spectrum

It is perfectly possible (and often very useful) to have fully interactive macros, during which the spectral display is modified, and the user can take actions depending on the results.

filename : rowint

```
; interactive

; permits to choose rows interactively on a 2D by clicking on the data-set

;

;see also : colint vertint planeint ph2dc ROW COL


if ($dim==1) error 'Works only in 2D'

refmacro 1

print "click on the data set, finish by pressing the third button of the mouse"
```

```
=loop

monopoint

if ($button==3) goto end

set x = (round($pointy[1]))

if ($itype_2d == 2 | $itype_2d == 3) set x = (2*int($x/2)+1)

print ('row :';$x)

row $x

goto loop

=end

refmacro 0 exit
```

You see an example of several techniques. First, the `refmacro` context, determines if the spectral display will be refreshed during macro execution. Depending on what you are doing (interactivity or Fourier transform) you want or you do not want the display to show what is happening. The `refmacro`, when set to 1, determines that the display will be refreshed, and when set to 0 that the display will not refresh until returning to the prompt level. So here we set it to 1 when entering, and set it to 0 when exiting. Then, the execution is blocked by the MONOPOINT command, which wait for user to click in the active window. After the click, the following contexts are set : `$pointx[1]` `$pointy[1]` and `$button`, which contains respectively the (spectral) coordinates of the point which was clicked, and the number of the button used (1 is the left most one).

varying formbox via file

In some cases, you may want to build a form with a variable number of entries. This is a problem as the FORMBOX command has to be called all at once in order to design a given form; you cannot use IF controls in the design of forms. One way to go is to write a temporary file, containing the code for the form. Here is an example taken from the assignment module (which heavily uses this trick).

This macro builds a long formbox, with one line per residue in the primary sequence of the protein under study. Assigned residues are clickable.

filename : show_prim_seq

```
; first search for assigned sys


;scan prim seq
open db/primary
set l = <db/primary
set i = 1
while (!eof('db/primary'))
set seq[$i] = $l
set i = (%+1)
set l = <db/primary
endwhile
close db/primary
; open temp file
set tmp = ('tmp' // int(1000000*$random))
open $tmp
fprint $tmp 'formbox primary DO_NOTHING \'


; for all residues
for j = 1 to ($i-1)
fprint $tmp ("'" // $j; $seq[$j] //"'" ; 'message \')
foreach p in sys within 1 ($j-0.1) ($j+0.1)
```

```
set ss = (head(tail($sys[$p])))

if ($p s! 'LARGEST' ) then

fprint $tmp ("noreturn '"; $ss; "#"; $p; "' action 'show_sys"; $p; "' \")

fprint $tmp ('noreturn Edit action "mod_sys'; $p; '" \')

endif

endfor

endfor

fprint $tmp "*"

close $tmp



@($tmp)



sh ("/bin/rm"; $tmp)

;dump
```

In this one example you can find many examples of the techniques described above : temporary files; writing *Gifa* code to a file; etc...

Additionally, note how the file is called : `@($tmp)` which is the syntax for calling a macro the name of which is help into a variable.

Macro programming

Gifa MACRO PROGRAMMING

- EXAMPLES :

- Contents