

# Prospa Programming Manual (V 3.11)

## Contents

1.	The Prospa macro language .....	3
1.1.	Writing macros .....	3
1.1.1.	Running procedures from within macros .....	5
1.1.2.	Local and global variables in procedures .....	6
1.2.	Prospa Command Syntax .....	7
1.3.	The Interpreter .....	7
1.3.1.	Operand types .....	8
1.3.2.	Variables .....	8
1.3.3.	Simple scalar operations .....	8
1.3.4.	Arithmetic operations .....	8
1.3.5.	Comparison operators .....	9
1.3.6.	Boolean operators .....	9
1.3.7.	Operations on complex numbers .....	10
1.3.8.	Operations on vectors and matrices .....	11
1.3.9.	Combining matrices and scalars .....	13
1.3.10.	Extracting elements from a matrix .....	14
1.3.11.	Extracting submatrices .....	14
1.3.12.	Extracting rows and columns from a matrix .....	15
1.3.13.	Operator precedence .....	16
1.3.14.	Operations on strings .....	16
1.3.15.	Operations on string lists .....	17
1.3.16.	Structures .....	17
1.3.17.	Built-in functions .....	18
1.4.	Control Statements .....	19
1.4.1.	The for-next statement .....	20
1.4.2.	The while-endwhile statement .....	20
1.4.3.	The if-endif statement .....	20
1.5.	Formatting strings .....	21
2.	Designing a macro user interface .....	22
2.1.	Giving the window a title .....	24
2.2.	Adding controls .....	24
2.3.	Editing controls .....	25

2.4.	Selecting Multiple Controls .....	26
2.5.	Aligning controls.....	26
2.6.	Distributing controls.....	27
2.6.1.	Distribute to ends .....	27
2.6.2.	Equally space controls .....	27
2.7.	Distribute in window .....	27
2.8.	Distribute in control .....	28
2.9.	Make the same width and or height .....	28
2.10.	Running and editing window macros .....	28
2.11.	Attach to .....	29
2.12.	The edited GUI window menu .....	31
2.13.	Modifying control numbers .....	32
2.14.	Modifying tab numbers .....	33
2.15.	The executing GUI menu .....	34
2.16.	File Utilities .....	35
3.	Editing the User-Interface Macro .....	35
3.1.	Call-back procedures.....	35
3.2.	Saving a new window .....	35
3.3.	Adding call-back procedures.....	36
3.4.	Getting and setting control parameters.....	37
3.5.	Using window variables .....	37
3.6.	Using separate procedures for call-backs.....	39
3.7.	Dialog boxes.....	40
3.8.	Extracting control values.....	41
3.8.1.	Using the control's Value ID.....	41
3.8.2.	Using the control's Object ID.....	42

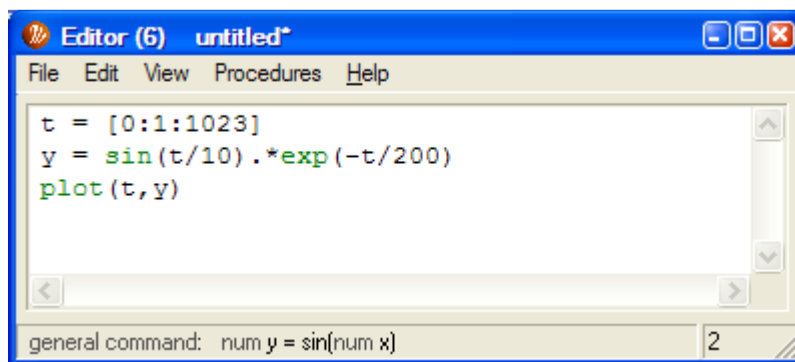
## 1. The Prospa macro language

The real power of Prospa becomes apparent when you learn how to write your own macro scripts and design window interfaces. Prospa macros scripts or just macros are based around a number of commands. These can be built in, added via the DLL (dynamic linked library) interface or file based commands. The Prospa language is largely procedural based with some object oriented features.

In chapter 7 of the user interface manual we saw that commands can be entered directly into the command line interface window where they will be run one line at a time when the enter key is pressed. However by using the text editor it is possible to enter complete scripts and run them.

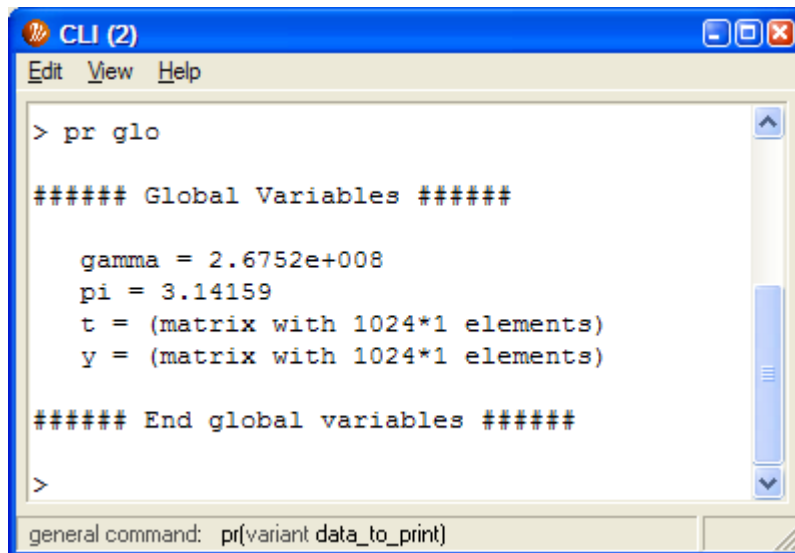
### 1.1. Writing macros

Prospa macros come in two forms - simple scripts and procedures. A simple script is a list of commands to be run when the script is executed. You execute the script by selecting the "Run text" menu option from the editor file menu (Ctrl-T) or by selecting the "Save and run text" menu option (Ctrl-R). If it has already been saved to a file you can alternatively execute it by typing the filename from the command line or main menu interface. You can't pass information to and from this kind of script except via global variables.



*An example of a simple script executed using the Run text menu option*

However all variables in this script are global and so are accessible from the command line interface. This is useful when you are developing a macro and want access to all the variables. To view all visible globally defined variables, type the command: **pr globals** or **pr glo** in the CLI window.



```
> pr glo

##### Global Variables #####

gamma = 2.6752e+008
pi = 3.14159
t = (matrix with 1024*1 elements)
y = (matrix with 1024*1 elements)

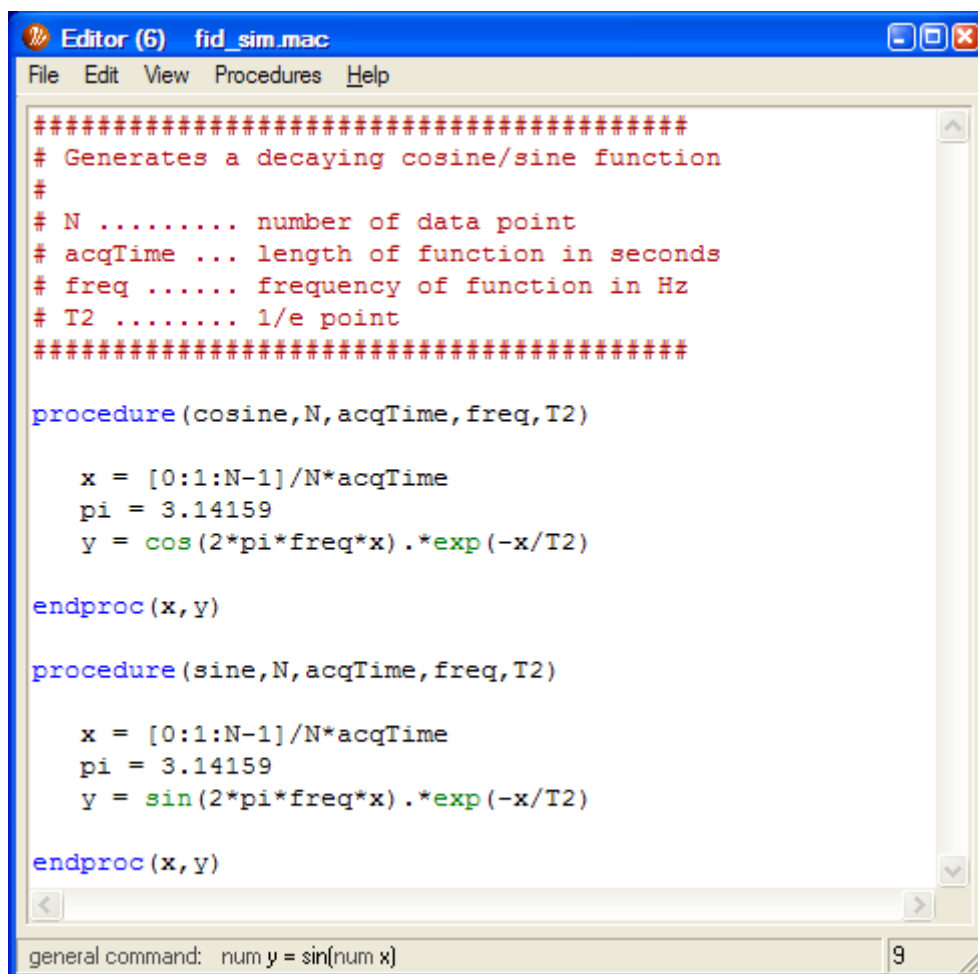
##### End global variables #####

>
```

general command: pr{variant data\_to\_print}

(Note that some global variables are hidden – to see these type **pr hidden**).

The second kind of macro - and this is the more generally useful kind - consists of procedures. These are lists of commands bounded by the `procedure()` and `endproc()` commands. You access the procedure using the name of the macro file and then the name of the relevant procedure. These two names are separated by a colon. If there is only one procedure in the file and its name matches the filename then the filename alone is sufficient to identify the procedure. For example, in the following edit window are the contents of the file "fid\_sim.mac". It contains two procedures that return either a decaying cosine or sine function.



```
#####
# Generates a decaying cosine/sine function
#
# N ..... number of data point
# acqTime ... length of function in seconds
# freq ..... frequency of function in Hz
# T2 ..... 1/e point
#####

procedure(cosine,N,acqTime,freq,T2)

    x = [0:1:N-1]/N*acqTime
    pi = 3.14159
    y = cos(2*pi*freq*x).*exp(-x/T2)

endproc(x,y)

procedure(sine,N,acqTime,freq,T2)

    x = [0:1:N-1]/N*acqTime
    pi = 3.14159
    y = sin(2*pi*freq*x).*exp(-x/T2)

endproc(x,y)

general command: num y = sin(num x) 9
```

To access the decaying cosine function, you could for example type

```
(x,y) = fid_sim(1024,1,10,0.2)
```

since it is the first procedure in the file or alternatively

```
(x,y) = fid_sim:cosine(1024,1,10,0.2)
```

while the sine function is obtained with:

```
(x,y) = fid_sim:sine(1024,1,10,0.2)
```

Both procedures generate two vectors;  $x$ , which contains 1024 numbers equally spaced between 0 and 1, and  $y$ , a decaying cosine or sine function with 1024 points, at 10 Hz and a  $T_2$  of 0.2 s.

### 1.1.1. Running procedures from within macros

Within a macro procedure you can call other procedures. There are two ways of referring to these procedures. The first includes the filename of the procedure followed by a colon and then the procedure name. This is useful if you wish to access a procedure in another file. However if the procedure is in the same file as the call then you should leave out the filename so that it only looks in the current file e.g.

```

procedure(test)

    x = [0:1:1023]
    y = :gauss(x,100)
    plot(x,y)

endproc()

procedure(gauss,x,w)

    y = exp(-x^2/w^2)

endproc(y)

```

Note that this even works if the macro has not been saved yet.

### 1.1.2. Local and global variables in procedures

All variables in a procedure are *local*, that is they are not accessible from the command line interface or other procedures, and are deleted when the procedure ends. To communicate with the procedure you pass variables or constants in an argument list (1024,1,10,0.2 in the above examples). These are copied to local variables (N,acqTime,freq,T2) which are then available to the procedure. To pass information back to the calling program, place the variables in the argument list to the **endproc** or **return** commands. These will be copied to the corresponding variables in the list to the left of the procedure call (x,y in these examples). Procedures can also access *global* variables since if the variable is not found in the local list the global list is then checked (but window variables are checked before global variables – see below.)

You can define global variables from within procedures by using the assign command:

```
assign(name, expression, "global")
```

This is much like typing

```
name = expression
```

except that “name” is now a global variable and “name” can be a variable itself. (This facility is useful when the name of the variable is to be specified from the graphical user interface): e.g.

```

a = "m1"
assign(a, noise(100,100), "global")

```

This generates a 100 by 100 global matrix with the name “m1” which has been filled with Gaussian noise.

Variables can also be given *window* scope. This means they are accessible if the procedure(s) are part of the same window interface. To learn more about this check out the graphical user interface help section.

## 1.2. Prospa Command Syntax

We start with the basic command syntax:

Commands take the form of a name followed by any arguments. These are comma delimited and must be surrounded by curved brackets. i.e.

```
command (arg1, arg2, arg3, ...)
```

unless the command does not return any values, in which case the following form is also acceptable

```
command arg1, arg2, arg3, ...
```

The first method should be the method of choice in macro scripts since it makes it easier to distinguish between commands and arguments. Also syntax colouring for procedures requires the brackets. In this manual built-in commands are indicated in **green** (this matches the editor syntax coloring).

As indicated above, commands may also return data. If they return a single value use this syntax:

```
d = command (arg1, arg2, arg3, ...)
```

If the command returns a number of values then surround them by curved brackets:

```
(d1, d2, d3 ...) = command (arg1, arg2, arg3, ...)
```

Note that you don't have to match the number of returned elements as long as there are not more values than returned from the command.

In the CLI, lines onto which data can be entered are signified by a prompt character '>'. i.e.

```
> command (arg1, arg2, arg3, ...)
```

Multiple commands can be entered on one line if they are separated by semicolons e.g.

```
> a = 2; pr a
```

In this case commands are evaluated in left to right order. **pr** is the print command. More complex command sequences can be stored in files and run as macro scripts.

## 1.3. The Interpreter

The Prospa macro language is executed using an interpreter – that is each command is executed one at a time. This provides maximum flexibility at the expense of speed

(although individual commands can be fast). The interpreter will now be introduced by giving some examples of the various operations which can be performed.

### 1.3.1. Operand types

The CLI interpreter supports several types of operand:

Numbers: e.g. 23, 23.4, 2+3j  
Vectors: e.g. [1,2,3,4]  
Matrices: e.g. [1,2;3,4;5,6]  
Strings: e.g. "Hi there"  
Lists: e.g. ["one","two","three","four"]  
Structures: see section 1.3.16  
Window objects: see section 3.8.2

### 1.3.2. Variables

Operands may be written explicitly in a command line or may be assigned to variables which may be used in their place. Variables are alphanumeric strings which must not be valid numbers. e.g.

```
scalar = 12.3  
mat1 = [1,2,3;4,5,6]  
str5 = "My string"  
lvar = "Also a variable"
```

The following strings cannot be used as variables:

"i" and "j" (reserved as equal to  $\sqrt{-1}$ )  
"nrargs", "parentlcr", (used by procedures) and "global", "local", "hidden" and "winvar"  
(used by the print command to refer to variable lists).

### 1.3.3. Simple scalar operations

In addition to about 400 built-in commands (type listcom at the CLI to see them all), the interpreter also supports the following operators for certain number types:

### 1.3.4. Arithmetic operations

#### *Binary operators*

- + , - : addition and subtraction
- \*, / : multiplication and division
- %: returns remainder after integer division (e.g. 10%3 = 1)
- ^ : raising to a power (e.g. 2^3 = 8)
- .\*: element to element matrix multiplication

#### *Unitary operators*

- : negation (e.g. -3)



' : transpose (single quote) (e.g.  $[1,2]^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ )

Scalar operations follow normal mathematical precedence and parenthesis can be applied to operands in the usual way, e.g.

```
a1 = 2*(3+7)^2
```

In this case `a1` evaluates to 200. To try this example in the CLI, type the following commands at the prompt

```
> a1 = 2*(3+7)^2
> pr a1

a1 = 200
```

The command `pr a1`, is short for `print a1` (the “pr” is optional in the CLI but must be included in a macro).

### 1.3.5. Comparison operators

`==` : equal  
`!=` : not equal to  
`>`, `<` : greater than , less than  
`>=`, `<=` :, greater than or equal to, less than or equal to

These commands result in either 0 or 1 (false or true) and are used for comparing numerical values.

#### *Example*

```
> a = 23
> result1 = (a == 20)
> result2 = (a != 20)
> result3 = (a > 20)
```

In these cases `result1` will be 0 (false) and `result2` and `result3` will be 1 (true).

Notice the difference between “=” as an assignment and “==” as a comparison. (There can be only one assignment operator in an expression, but any number of comparisons). In these examples the parenthesis are only included for clarity.

### 1.3.6. Boolean operators

`&`, `|` : logical AND and OR  
`inv` : negates all binary elements  
`not` : turns true into false and vice-versa

These first two of these operators perform the specified binary Boolean operation between two adjacent expressions e.g.

```
> result = 1 | 2
> pr result
    result1 = 3

> result = 1 & 5
> pr result
    result = 1

> a = 10
> pr (a > 5 & a < 15)

    a > 5 & a < 15 = 1
```

Note that > and < have a higher precedence than & and so no brackets are required (see below), although they wouldn't hurt.

The inv operator inverts each bit e.g.

```
> pr hex(inv(0xFF))

    FFFFFFF00
```

Note the use of the 0x?? syntax to write hexadecimal numbers and the **hex()** operator to convert the result into a hexadecimal string.

The **not()** operator simply turns true into false and vice versa

```
> a = 10
> pr not(a = 10)
    not(a = 10) = 0
```

### 1.3.7. Operations on complex numbers

Complex scalars can be defined using the following syntax:

```
z = a + ib
z = a + bi
or
z = a + jb
z = a + bj
```

where a and b are real scalars and i and j are special variables equalling  $\sqrt{-1}$ . Note that the real/imaginary order is unimportant however there should not be any spaces between the complex symbol and its associated magnitude i.e. 2i or 2\*i is ok 2 i is not.

The mathematical operators (+,-,\*,/) can then be applied between these numbers. e.g.

```
> z1 = 2+3i
> z2 = 4+6i
> pr z1*z2
```

```

z1*z2 = -10+24i
> pr z1/z2
z1/z2 = 0.5-0i

```

Complex scalars can also be combined with real scalars e.g.

```

> pr ((2+3i)*5)
(2+3i)*5 = 10+15i

```

To calculate the length or norm a complex number use the mag (magnitude) command

```

> pr mag(10+15i)
mag(10+15i) = 18.0278

```

### 1.3.8. Operations on vectors and matrices

A vector is defined as an array of real and/or complex scalars. In Prospa it is represented by a list of comma delimited scalars bounded by square brackets e.g.

```
a = [2, 5.4, 9, 10.3]
```

If the vector is to contain complex numbers use curly braces:

```
a = {2, 3+4j, 8j}
```

In Prospa all vectors are just handled as row or column matrices. A matrix is written like a vector except that each row is separated by a semicolon e.g.

```

> a = [1,2;3,4;5,6]
> pr a

a =

1      2
3      4
5      6

```

In those cases where the elements of a vector or matrix have a simple linear relationship then you can use the following syntax to define them:

```

> a = [0:0.2:1]
> pr a

a =

0      0.2      0.4      0.6      0.8      1

```

and

```

> a = [0:0.2:1; 0:0.4:2; 0:0.8:4]
> pr a

a =

```

```

0      0.2      0.4      0.6      0.8      1
0      0.4      0.8      1.2      1.6      2
0      0.8      1.6      2.4      3.2      4

```

If you don't specify the step size it defaults to 1 i.e.

```

> pr [0:5]

[0:5] =

      0      1      2      3      4      5

```

If you just want a zeroed vector or matrix use the `matrix` or `cmatrix` commands:

```

> a = matrix(x)
> a = matrix(x,y)
> a = cmatrix(x)
> a = cmatrix(x,y)

```

The first (and third) command(s) produces a (complex) row vector with x elements while the second (and fourth) produces a (complex) matrix with x columns and y rows (**note the ordering, Prospa uses (x,y) not (row,col)**) e.g.

```

> pr matrix(2,3)

matrix(2,3) =

      0      0
      0      0
      0      0

```

3D matrices are handled in the same way except they have an extra dimension - z. e.g.

```

> z = matrix(2,2,3)
> pr z

z =

Plane 0:

      0      0
      0      0

Plane 1:

      0      0
      0      0

Plane 2:

      0      0
      0      0

```

Complex 3D matrices are generated using the `cmatrix` command e.g.

```

> z = cmatrix(2,2,3)
> pr z

```

```

z =

Plane 0:

0+0i    0+0i
0+0i    0+0i

Plane 1:

0+0i    0+0i
0+0i    0+0i

Plane 2:

0+0i    0+0i
0+0i    0+0i

```

### 1.3.9. Combining matrices and scalars

Most mathematically valid operations can be used to combine 1D and 2D matrices with themselves or with scalars. A few special commands are required however:

The operators +, -, .\*, / operate on matrix elements e.g.

```

> a = [1,2;3,4;5,6]
> pr a.*a

a.*a =

1    4
9    16
25   36

```

Note the dot before the \* symbol. This indicates an element-by-element operation. Without the dot it becomes the normal matrix multiply e.g.

```

> a = [1,2;3,4]
> b = [5,6]
> pr b*a

b*a =

23    34

```

Matrices can be transposed using the ' operator (or the trans command)

```

> a = [1,2]
> b = [3,4]
> pr a*b' # or pr a*trans(b)

a*b' = 11

> pr a'*b # or pr trans(a)*b

a'*b =

3    4

```

### 1.3.10. Extracting elements from a matrix

To refer to a particular matrix element use the following syntax:

```
> a = [1,2;3,4;5,6]
> pr a[1,2]

a[1,2] = 6
```

Again the index ordering is [x,y] not [row, column]. Also note that the matrix index is zero based (not 1).

In the case of a 3D matrix the order is [x,y,z]

To extract elements from the end of an array you can use negative numbers, -1 refers to the last element -2 the second to last and so on.

```
> a = [0:10]
> pr a[-1]

a[-1] = 10
```

### 1.3.11. Extracting submatrices

submatrix access can be achieved using array within array syntax:

```
m = [1:0.5:20]
s = m[1:5]

s =

    1.5    2    2.5    3    3.5
```

and for 2D matrices

```
m = noise(4,4)
s = m[1:2,1:2]
pr m
pr s

m =

-0.582984    1.55569    1.15892   -0.265666
 1.50081   -1.26301    0.30973   -0.525005
 0.97462    0.865859   -0.274781   -1.00211
 0.829931    0.147783    0.663434   -0.947541

s =

-1.26301    0.30973
 0.865859   -0.274781
```

Access from the start of a matrix to a certain index or from a specified index to the end (more useful) can be achieved using the following syntax

```
m = [1:10]
pr m
pr m[:3]
pr m[3:]

m =
    1     2     3     4     5     6     7     8     9    10

m[:3] =
    1     2     3     4

m[3:] =
    4     5     6     7     8     9    10
```

It can also be used with the step size operator:

```
m = [0:10]
pr m
pr m[4:2:]

m =
    0     1     2     3     4     5     6     7     8     9    10

m[4:2:] =
    4     6     8    10
```

The negative index can also be used to refer to index relative to the last element (-1):

```
> pr m[2:-2]

m[2:-2] =
    2     3     4     5     6     7     8     9
```

### 1.3.12. Extracting rows and columns from a matrix

Occasionally it can be useful to be able to extract a row or column from a matrix. The following example shows how this can be done:

```
> a = [1,2;3,4;5,6]
> pr a[:,1]

a[:,1] =
    3     4
```

Here the colon operator ':' refers to all elements of row 1 (in the x direction). So in this case it says print all elements of row 1 (zero based again).

You can also set rows or columns in a matrix in the same way:

```
> a[1, :] = trans([0,0,0])
> pr a

a =

1      0
3      0
5      0
```

The same method can be used to extract data from higher dimensional data sets

(The colon operator used here is an alternative to the tilde operator ~ used in version 2).

### 1.3.13. Operator precedence

When operators act on operands they follow a certain precedence that is, some operators are applied before others. The following list defines this operator precedence from highest to lowest. In the case that they have the same precedence, operators are applied left to right.

```
1  ( ), [ ], { }
2  ^, \
3  - (Negation)
4  *, /, .*, %
5  +, -
6  ==, !=, >, >=, <, <=
7  |, &
```

So for example, in the expression

```
-6*(3+4)^5
```

The negation is applied to the leading 6 and then the 3 and 4 are added before being raised to the power of 5. The multiply is the last operator to be applied. So the answer is therefore -100842.

### 1.3.14. Operations on strings

Some of the operators (+, ==, and !=) can also be applied to strings. e.g.

```
s1 = "Hi"
s2 = "there"
pr s1 + " " + s2
    "Hi there"

pr (s1 == s2)
    (s1 == s2) = 0
```



```
pr (s1 != s2)
(s1 != s2) = 1
```

### 1.3.15. Operations on string lists

Lists are arrays of string. Each entry can have a different length. e.g.

```
names = ["Joe", "Richard", "Sally", "Patricia"]
```

You access list elements just as you would a numerical array:

```
pr names[0] + " and " + names[2] + " are married"

Joe and Sally are married
```

The only operations permissible on a list are concatenation and comparison:

```
pr names + "Fred"

> pr names + "Fred"

names + "Fred" =

Joe
Richard
Sally
Fred

> pr (names == ["not", "this", "list"])

names == ["not", "this", "list"] = 0

> pr (names == names)

names == names = 1
```

### 1.3.16. Structures

Like lists, structures are a way of collecting items together into a single object which can then be interrogated, copied or passed to other procedures as required. Unlike lists, structures can contain any type of Prospa variable, not just strings. Structures are formed using the `struct` command and the structure member access operator (`->`).

```
a = struct()
a->name = "Joe"
a->age = 23
a->address = "18 Small St"
a->map = sin([0:0.01:10])'*(cos([0:0.01:10]))
```

The variables `name`, `age`, `address` and `map` are examples of structure members. Note that in Prospa the dot character `'.'` has no special significance other than in real numbers and the dot product `'.*'` so, `a.map` would just be a string not a structure member as in many other languages.

These structure members may be used in the same way as any other variable or converted into local variables using the `assignstruct` command.

```
procedure(structtest)

    a = struct()
    a->name = "Joe"
    a->age = 23
    a->address = "18 Small Street"
    a->map = sin([0:0.01:10])'*(cos([0:0.01:10]))
    assignstruct(a,"local")

    pr local

    image(map)

endproc()
```

The `struct` command can also be used to convert local or window variables into a structure (the reverse of the above operation)

```
procedure(test)

    name = "Joe"
    age = 23
    address = "18 Small Street"
    map = sin([0:0.01:10])'*(cos([0:0.01:10]))

    s = struct("local")
    pr s

endproc(s)
```

### 1.3.17. Built-in functions

Some other useful commands which can be used with scalars and matrices are. Many commands work on single or double precision arguments. Commands new or modified in V3 are shown in red.

<b>abs(z)</b>	return the absolute value of z
<b>acos(z)</b>	take the arccosine of z
<b>asin(z)</b>	take the arcsine of z
<b>atan(z)</b>	take the arctangent of z
<b>cmatrix(x,y)</b>	make a zeroed, complex matrix (up to 4 dimensions)
<b>conj(z)</b>	take the complex conjugate of z
<b>cosh(z)</b>	take the hyperbolic cosine of z
<b>cos(z)</b>	take the cosine of z
<b>cumsum(m)</b>	calculate the cumulative sum of elements in matrix m
<b>double(z)</b>	convert a single precision number z to double precision
<b>dmatrix(x,y)</b>	make a zeroed, real double precision matrix (up to 2 dimensions)
<b>dnoise(x,y)</b>	make a real double precision matrix and fill it with gaussian noise sd = 1. (Up to 2 dimensions)
<b>eval(s)</b>	evaluate a string and return a number or a matrix.

<b>exp(z)</b>	take the exponential of z
<b>factorial(z)</b>	take the factorial of z
<b>ft(z)</b>	take the Fourier transform of vector z
<b>hex(z)</b>	convert z into hexadecimal format
<b>hft(z)</b>	take the Hilbert transform of z (generates <b>imag(a)</b> from <b>real(a)</b> )
<b>ift(z)</b>	take the inverse Fourier transform of vector z
<b>image(z)</b>	display matrix z as an image
<b>imag(z)</b>	return the imaginary part of z
<b>insert(a,x,y,b)</b>	insert matrix a into matrix b at position x,y
<b>j0(z)</b>	zeroth order Bessel function
<b>j1(z)</b>	first order Bessel function
<b>join(a,b)</b>	join two matrices a and b.
<b>linvec(a,b,n)</b>	generate a linear vector with values from a to b having n points
<b>log10(z)</b>	return the base 10 logarithm of z.
<b>log2(z)</b>	return the base 2 logarithm of z.
<b>loge(z)</b>	return the base e logarithm of z.
<b>matrix(x,y)</b>	make a zeroed, real matrix ( <b>up to 4 dimensions</b> )
<b>max(z)</b>	find the maximum value in a matrix.
<b>min(z)</b>	find the minimum value in a matrix.
<b>not(z)</b>	convert all trues into falses in z and vice versa.
<b>noise(x,y)</b>	make a real x by y matrix and fill it with gaussian noise sd = 1. ( <b>Up to 4 dimensions</b> )
<b>outer(a,b)</b>	form the outer or tensor product between matrices a and b.
<b>round(z)</b>	return the closest integer to z
<b>real(z)</b>	return the real part of z
<b>realtostr(z)</b>	convert a real expression to a string.
<b>reflect(a)</b>	reflect matrix a
<b>reshape(m,w,h)</b>	reshape matrix m into size w by h. ( <b>Up to 4 dimensions</b> )
<b>rotate(m,x,y)</b>	rotate a matrix by amount x,y.
<b>rft(z)</b>	take the real Fourier transform of vector z.
<b>shift(m,x,y)</b>	shift contents of a matrix by amount x,y.
<b>sin(z)</b>	take the sine of z
<b>sinh(z)</b>	take the hyperbolic sine of z
<b>single(z)</b>	convert a double precision number z to single precision
<b>size(z)</b>	return the dimensions of matrix z.
<b>sqrt(z)</b>	take the square root of z.
<b>submatrix(z..)</b>	return part of matrix z as a new matrix.
<b>sum(z)</b>	return the sum of all the elements in matrix z.
<b>tan(z)</b>	take the tangent of z
<b>tanh(z)</b>	take the hyperbolic tangent of z.
<b>trans(z)</b>	take the transpose of matrix z.
<b>trunc(z)</b>	return the closest smallest integer to z

## 1.4. Control Statements

Some commands are specific to macro scripts. These are commands that control the order in which commands are executed. These commands are:

```

for ..... next
while ... endwhile
if ..... elseif ... else ... endif

```

The first pair of statements allows a group of commands to be executed a fixed number of times. e.g.

### 1.4.1. The for-next statement

```

for(n = 0 to 0.5 step 0.1)
    pr("\n    n = $n$")
next(n)

```

Executing this script will result in the following output

```

n = 0
n = 0.1
n = 0.2
n = 0.3
n = 0.4
n = 0.5

```

The **for** loop has an initial value (here n = 0), an end value (here n = 0.5), and a step size (here 0.1). Without the "step" keyword the program will increment the loop variable by 1 each time. All statements between the **for** and **next** commands will be repeated until loop variable exceeds the end value.

A for-next loop can be prematurely exited by calling the exitfor command. Note that there is no equivalent to the continue statement found in such languages as C.

### 1.4.2. The while-endwhile statement

The while ... endwhile statement is similar to the for loop except that only the exit condition is implicit in the control structure. The following piece of code duplicates the function of the for ... next loop above.

```

n = 0
while(n <= 0.5)
    pr("\n    n = $n$")
    n = n + 0.1
endwhile

```

Here all statements between the while and endwhile commands will be repeated until the while condition is false

A while loop can be prematurely exited by calling the exitwhile command. Note that there is no equivalent to the continue statement found in such languages as C.

### 1.4.3. The if-endif statement

The if-elseif-else-endif structure is a way of selecting different pieces of code based on comparisons. The syntax is very similar to that found in other languages:

```
a = 1
if (a < 1)
    pr("\n a < 1")
elseif (a < 2)
    pr("\n a < 2")
else
    pr("\n a >= 2")
endif
```

This code will result in the output

```
a < 2
```

Note that the `elseif` and `else` statements are optional - only the `if-endif` pair must always be present. Also note that the conditionally executed command must be a separate statement i.e. `if` must be on a new line or separated from the control statement by a semicolon.

If you don't have a matching `endif` for each `if` (or `next` for each `for` or `endwhile` for each `while`) then the macro will not be executed and you will be provided with an appropriate error message.

## 1.5. Formatting strings

When printing expressions embedded in strings, a format specifier can be included to control the number of digits or precision. The format specifier is included after the expression i.e.

```
pr "... $expression, format$ ..."
```

The format specifier follows C syntax (the following is taken from the book "The C Programming Language" by Kernigan and Ritchie).

- Flags (any order)

-, which specifies left justification of the converted argument in its field

+, which specifies that the number will always be printed with a sign.

0, specifies padding to the field width with leading zeros.

Examples

```
> a = 23
> pr "a = $a,-10.2f$Hz"
a = 23.00      Hz
> pr "a = $a,+10.2f$Hz"
a =          +23.00Hz
> pr "a = $a,010.2f$Hz"
a = 0000023.00Hz
```

- A number specifying a minimum field width. In the above examples this was 10. This includes the decimal point.
- A full-stop (period) which separates the field width from the precision.
- A number which is the precision which specifies the number of digits to be printed after the decimal point. (2 in the above examples).
- A conversion character which should be one of:

**f** : floating point the output format is of the form `[-]mmmm.ddd` where the number of *d*'s is specified by the precision. The default precision is 6 while a precision of 0 suppresses the decimal point. The number of *m*'s will depend on the field width.

**e, E** : scientific notation. Output has the form `[-]m.ddd e ± xx` or `[-]m.ddd E ± xx`, where the number of *d*'s is specified by the precision. The default precision is 6 while a precision of 0 suppresses the decimal point.

**g, G** : *e* or *E* are used if the exponent is less than -4 or greater than equal to the precision, otherwise *f* is used. Trailing zeros or a trailing decimal point is not used.

#### Examples:

```
> a = 12345.6789
> pr ("a = $a,10.2f$")
a =    12345.68

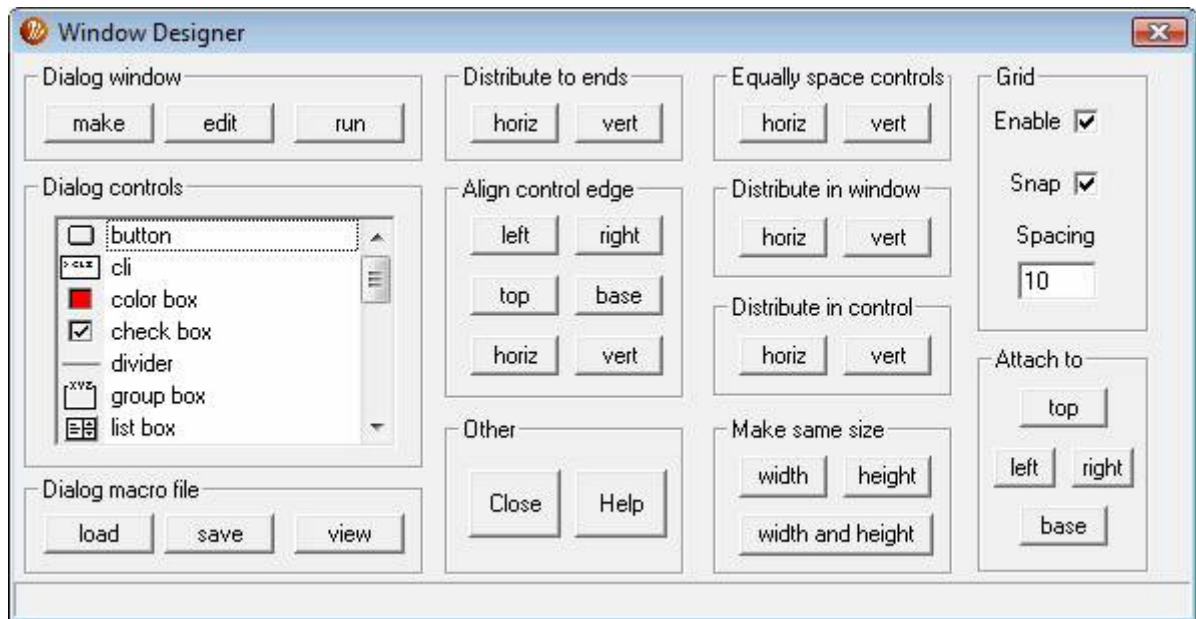
> pr ("a = $a,10.2e$")
a =    1.23e+04

> pr ("a = $a,10.2g$")
a =    1.2e+04

> pr ("a = $a,10.5g$")
a =    12346
```

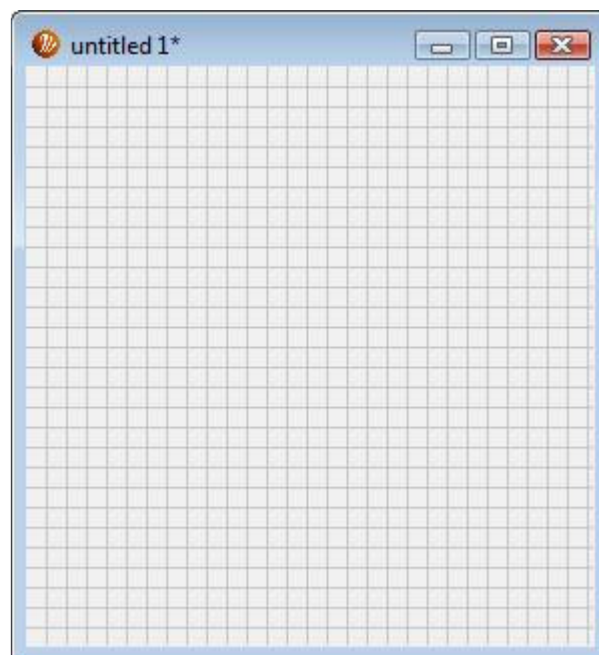
## 2. Designing a macro user interface

In many cases a macro is considerably easier to use if it has a well-designed graphical user interface (GUI) attached to it. The macro *designer*, (found in the main GUI menu), is included to assist in the design of a GUI for data processing macros.. The window consists of several regions as shown below:



A GUI interface consists of a parent window and the controls which are contained within it (occasionally referred to as “objects” in the Prospa manuals since not all provide a control function).

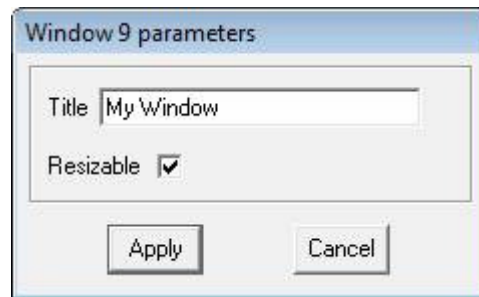
To use the designer, start by pressing the button labeled "make" in the region labeled “Dialog window”. A blank window entitled “untitled 1” will appear.



The grid displayed over the window maybe removed or modified using the controls in the section labeled "Grid" on the top right of the Designer window. With the snap option set the top left of selected controls will move to the nearest grid point when placed or moved.

## 2.1. Giving the window a title

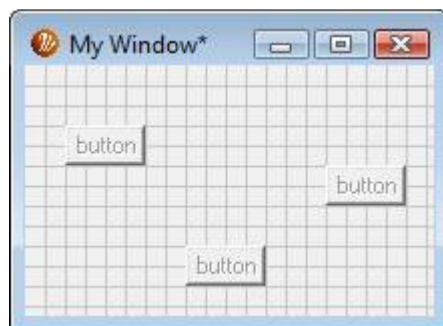
To add a title to the window, double-click on the window background (i.e. away from a control). The following window will appear:



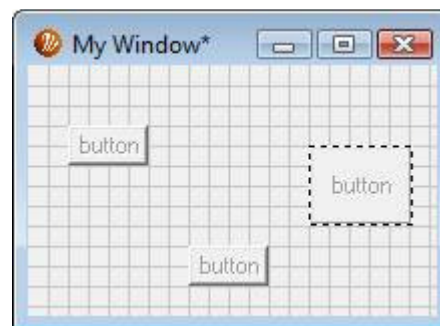
Into this window enter the name for your window and press the Apply button. The new title will now appear in the title bar of the window you are editing. You can also specify whether this window will be resizable or not (although it will always be resizable while you are editing it).

## 2.2. Adding controls

Next add controls by single or double clicking on one of the lines in the listbox labelled "Dialog Controls" in the Designer and then transfer the control to the new window by clicking the left mouse button again, once the cursor is at desired location. An example it shown below



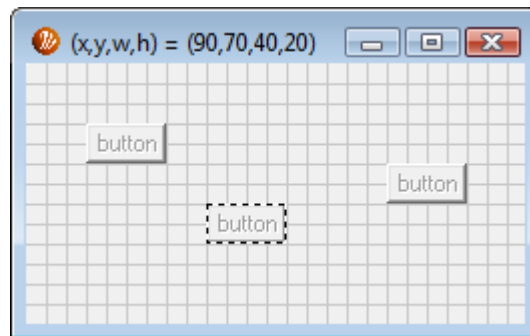
*Making buttons*



*Resizing a selected control*






For a short time after selecting a control its dimensions and position will be shown in the title bar, before reverting to the normal display:

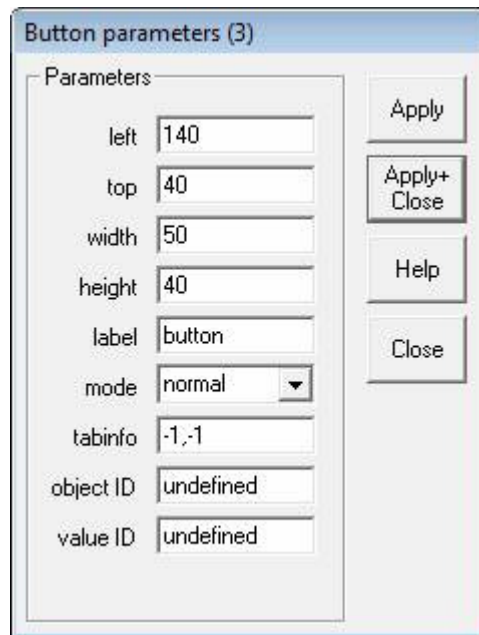




## 2.3. Editing controls

Note that the most recently generated control is surrounded by a dotted rectangle. This means that it can be edited. Another control can be edited simply by selecting it with the mouse. Once selected the possibilities are:

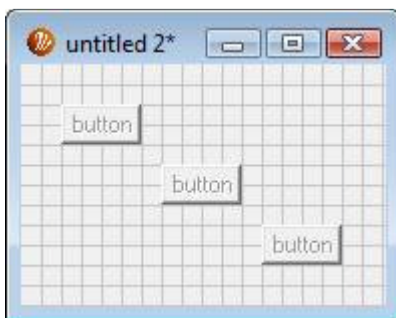
- Moving the control by dragging it with the mouse or by using the arrow keys. The cursor must look like  for this to happen.
- Resizing the control by moving the cursor to the right or lower side of the control and then dragging the mouse. The cursor will change from  to  when resizing is possible. (Note that some controls cannot be expanded or allow only horizontal or vertical expansion, in the latter case the icon will be  or . Some controls provide all three options).
- Resizing the control using the shift key and the arrow keys. Pressing the control key at the same time will cause the movement or resizing to be done in bigger steps.
- Deleting the control by pressing the delete key on the keyboard. (It cannot be pasted later on).
- Copying the control by pressing Ctrl-C (it can be pasted with Ctrl+V later on – hold down the shift key at the same time to place the new control in the same location as the parent).
- Cutting the control by pressing Ctrl-X (it can be pasted later on).
- Press combination Ctrl-Z to undo the last operation..
- Edit the control label and name and any other relevant parameters by double clicking on it. This displays a dialog specific to the type of control you have clicked on. In addition to the control type the title also include the control number in brackets. You can modify any of the control parameters and then press the “Apply” or “Apply + Close” to update the control. Close the dialog when finished.



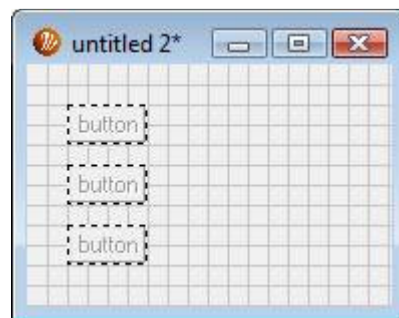
## 2.4. Selecting Multiple Controls

Several controls can be simultaneously selected in two ways:

- Shift-clicking on the controls
- Dragging a rectangle around them



*Multiple control selection*



*Left alignment*

Once the controls have been selected all the operations listed above for a single control can be applied (apart from editing). In addition, two other types of operation can also be applied; alignment and distribution.

## 2.5. Aligning controls

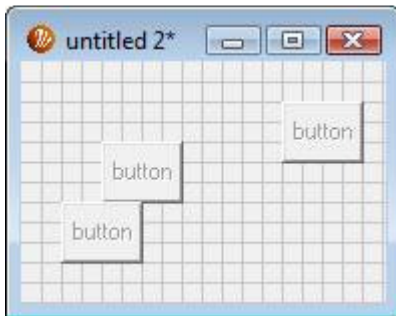
All the selected controls are shifted so either the left or right sides or the centres of the controls are lined up. The user must select the reference control before this command can be executed.

## 2.6. Distributing controls

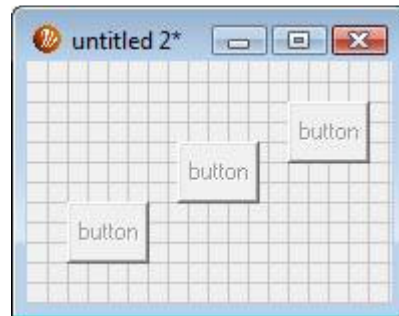
Here there are several possibilities:

### 2.6.1. Distribute to ends

All the selected controls (except the two at the extreme ends) are shifted so that the gap between them is constant. In some cases it may be necessary to move one of the end controls if it is not possible to have a constant gap between all controls.



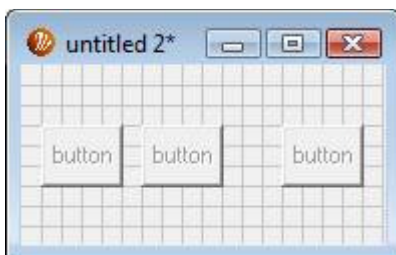
*Controls before distribution*



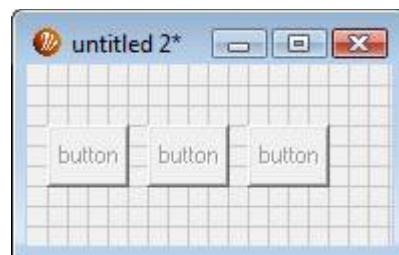
*After distribute to ends*

### 2.6.2. Equally space controls

Here the space between the two leftmost or topmost controls is used to set the spacing for all the other controls. Be careful with this option as you can have controls vanish off the edge of the window if there are too many controls selected and/or the spacing is too large.



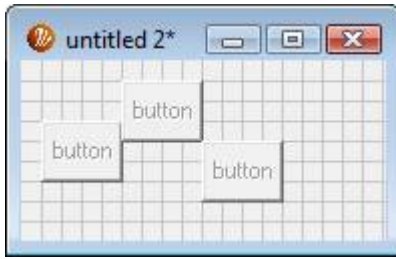
*Controls before distribution*



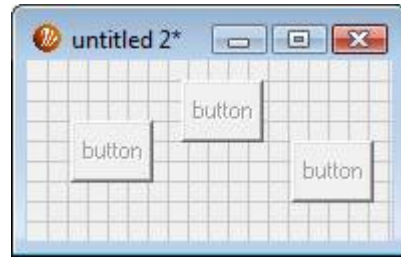
*After equal space distribution*

## 2.7. Distribute in window

In this case the controls are distributed so that they are equally spaced within the parent window:



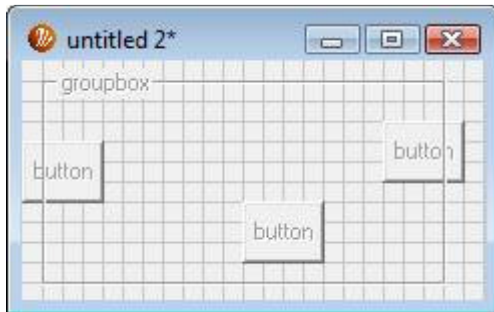
*Controls before distribution*



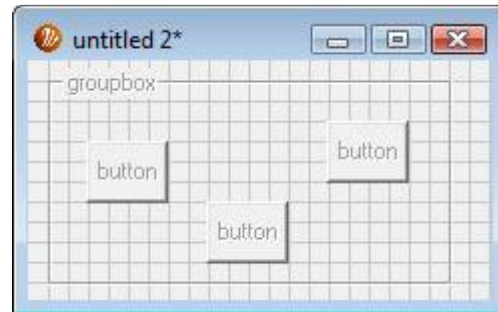
*After window distribution*

## 2.8. Distribute in control

Finally it is possible to distribute controls inside another control – typically a group-box. In this case you need to select the enclosing group-box before the command can complete.



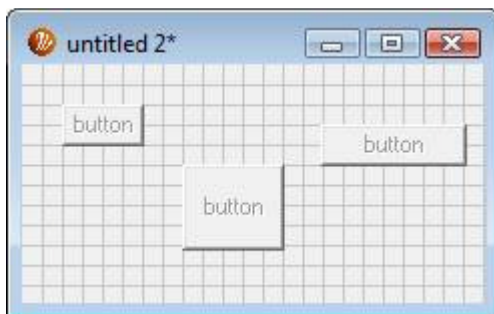
*Controls before distribution*



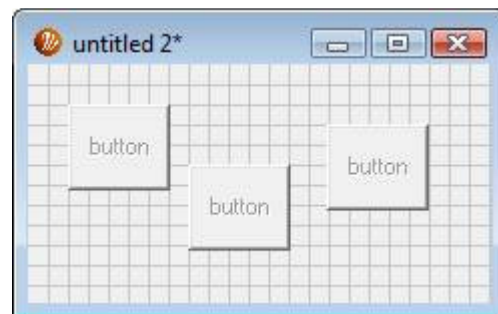
*After control distribution*

## 2.9. Make the same width and or height

These 3 options allow different controls to have the same width, height or both. Just select the control(s) to modify and then press the desired option in the designer. Finally select the reference control i.e. the control which has the desired width or height. All the selected controls will then be modified.



*Controls before resizing*

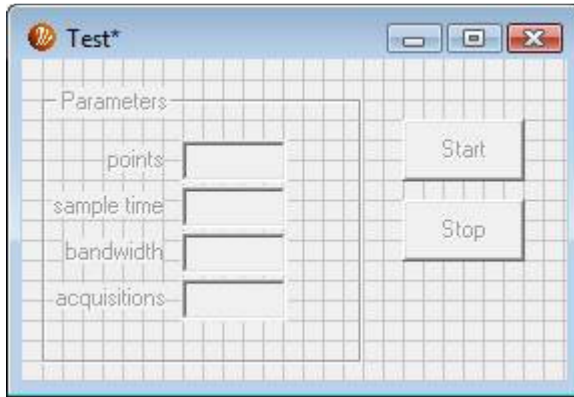


*Controls after resizing width and height*

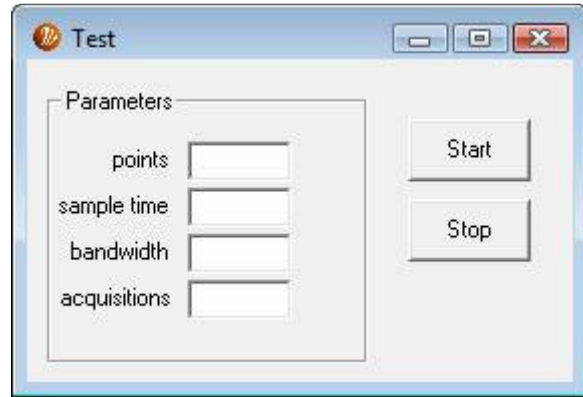
## 2.10. Running and editing window macros

The two other buttons in the "Dialog window" region modify the state of the edited window:

- edit : Click on a window to make it editable.
- run : This makes the window executable i.e. all the controls now operate.



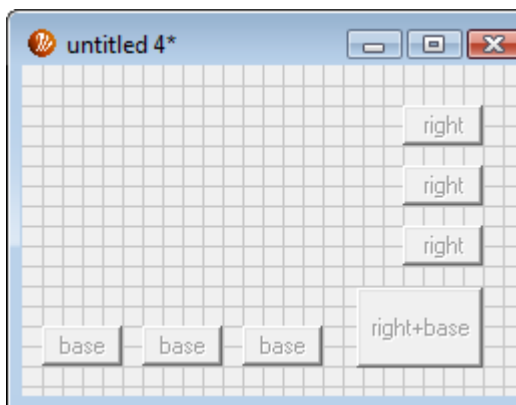
*A window in edit mode*



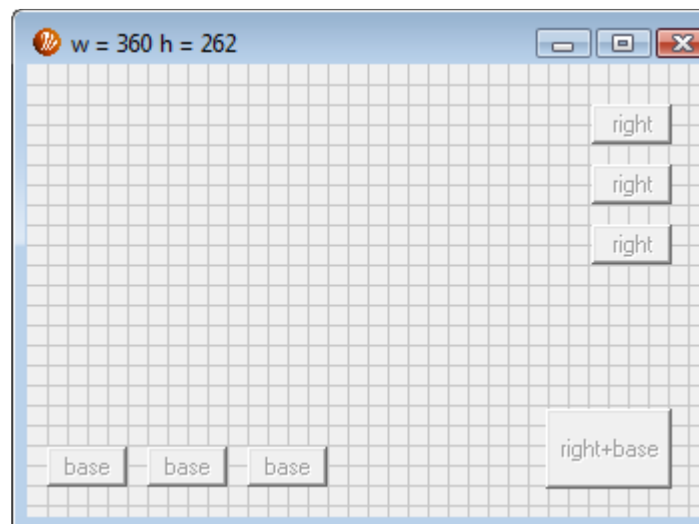
*The same window in run mode*

## 2.11. Attach to

By default manually placed controls area attached to the top and left of the parent window. In other words when you resize the window these controls do not move relative to the top and left edge. However if one the *Attach to* buttons is pressed while one or more controls are selected then those controls can be attached to the right or bottom edges. In this was the control will appear to move with the window when it is resized. Examples are given below.



*Before resizing*



*After resizing*

Internally this is achieved by replacing the x or y absolute coordinate with an equation which gives the position relative to the window width (ww) or window height (wh). ww and wh are special variables which only have meaning in these expression.

**Button parameters (7)**

Parameters

left	ww-85
top	wh-54
width	63
height	40
label	right+base
mode	normal
tabinfo	-1,-1
object ID	undefined
value ID	undefined

Buttons: Apply, Apply+Close, Help, Close

The allowable formats for the positioning equation are quite limited and have the form

```
ww*factor + offset
wh*factor + offset
```

where factor and offset are optional. factor and offset must be positive. Spaces are optional.

Controls can also be expanded with the parent window by using these constant in the width and height fields.

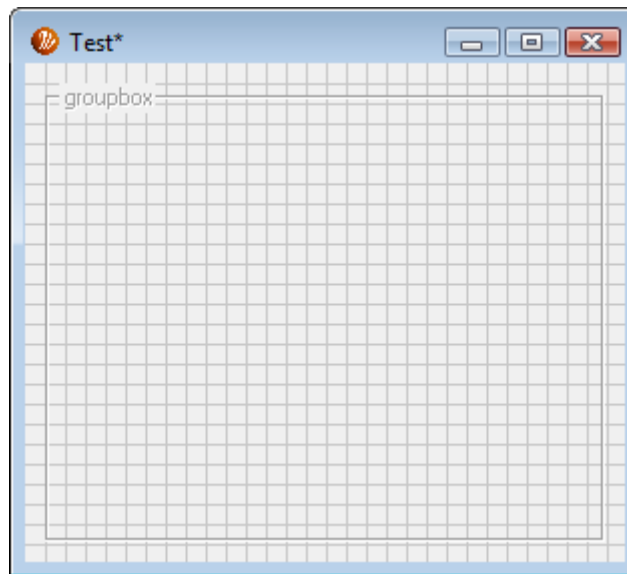


**Group box parameters (1)**

Parameters

left	10
top	10
width	ww-20
height	wh-20
label	groupbox
tabinfo	-1,-1
object ID	undefined
value ID	undefined

Buttons: Apply, Apply+Close, Help, Close



## 2.12. The edited GUI window menu

The edited GUI (graphical user interface) window can display a contextual menu if the right mouse button is pressed while over the window:

This menu contains several sections.

The first simply contains a command to execute the window i.e. make it usable. This is useful to test any modifications to the interface.

The next 3 options allow the copying, pasting and cutting of selected controls. Note that you can copy controls from one window and paste then in another.

The following 5 options allow the selection and alignment of controls.

The final two sections allow the control and tab numbers to be displayed and reset. These options require a little more explanation.

Execute	Ctrl+E
Hide window	Ctrl+W
Copy control(s)	Ctrl+C
Paste control(s)	Ctrl+V
Cut control(s)	Ctrl+X
Select All	Ctrl+A
Left Align	Ctrl+L
Right Align	Ctrl+R
Top Align	Ctrl+T
Base Align	Ctrl+B
Show control numbers	Ctrl+Shift+N
Modify control numbers	
Show tab numbers	Ctrl+N
Modify tab numbers	

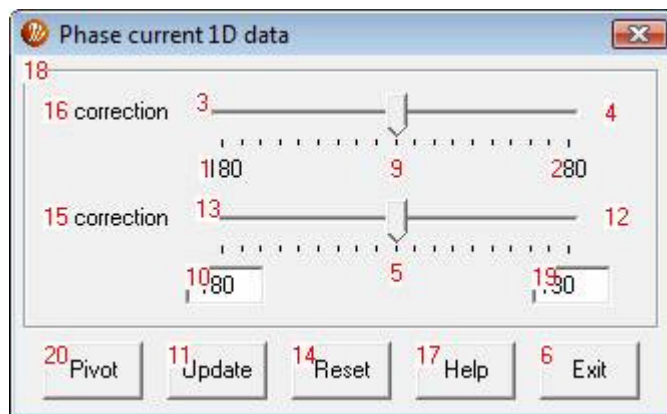
*GUI window contextual menu*

In a GUI interface macro all controls have an associated number, called the control number. This number is used to refer to the control through the `setpar` and `getpar` commands. These commands allow aspects of the control such as its appearance and function to be inspected or modified. The control numbers also determine the order in

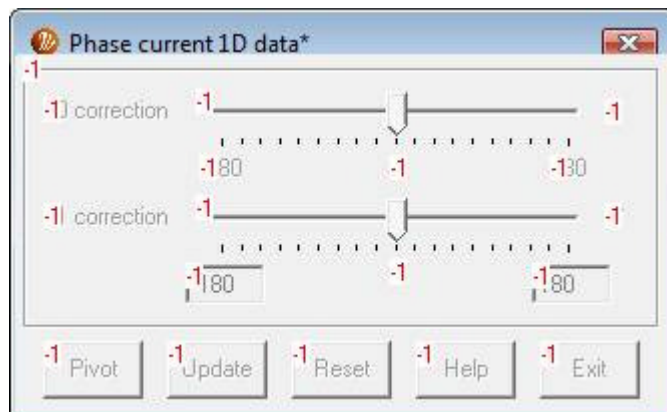
which the controls will be stored in the macro when you choose the save option in the designer. It is a good idea once you have designed the interface to reorder the control numbers in some sensible way before continuing with the macro writing. For example start with control 1 at the top and then increment the numbers by one as you move left to right and down the window.

### 2.13. Modifying control numbers

The first step is to display the control numbers in the window. To do this, choose the “Show control numbers” option from the menu.

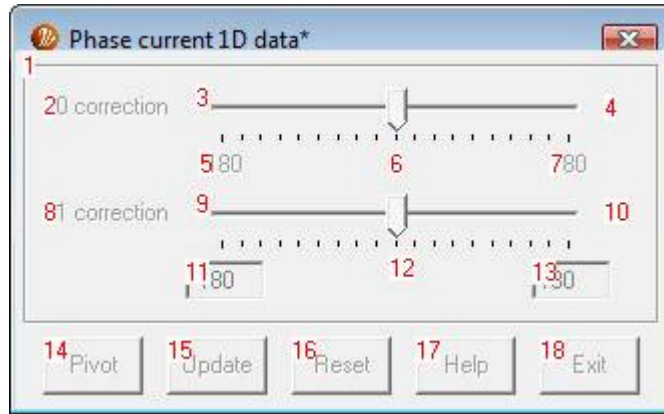


As you can see the numbers are in no particular order here. Next choose the “Modify control numbers” menu. You will be warned that the numbers will be reset – just press the “Yes” button. The result will be that all the numbers will be reset to -1.



Next start clicking on the controls to reorder the control numbers. By default the numbering starts from 1 and increments by 1 each time.





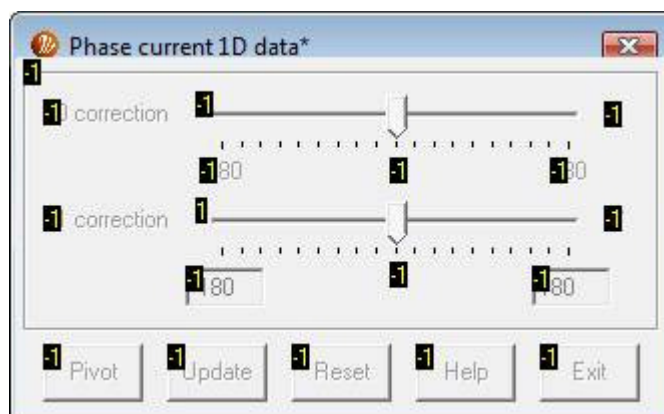
When you have clicked on all the controls choose the “Stop modifying controls” option from the menu. The numbers will disappear. If you make a mistake you will have to reset the numbers and start again.

If you add extra controls at a later date you should resist reordering the control numbers since this means that all `setpar` and `getpar` commands will need to be modified. (Note that there are alternative ways of referring to controls although using control numbers can often be the most efficient – especially when you have a an array of similar controls).

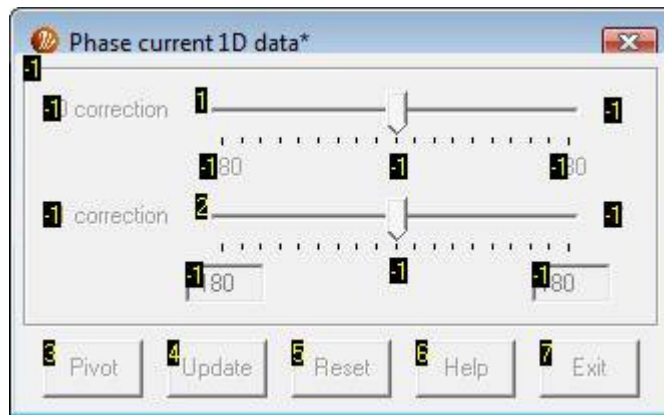
## 2.14. Modifying tab numbers

Tab numbers determine the order in which controls are accessed when the tab key is pressed. If they are not defined the control key order will be used. However, as you add extra controls this can cause the tab order to become rather randomised. By defining tab keys you can make the tab order independent of the control number order.

Modifying tab keys is done in the same way as modifying control keys. First select the “Show tab numbers” option from the contextual menu. The default tab numbers will again be -1:



Next select the controls in the order you wish to tab – note that you don’t need to select all the controls since some are not tab-able.



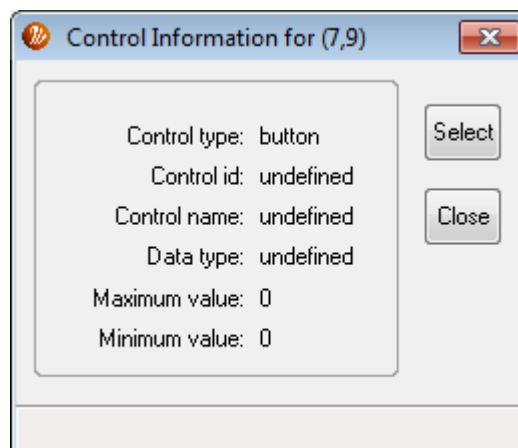
Finish off by selecting the “Stop modifying tab numbers” menu option. By selecting the “Execute” menu option you can now check that the tab order is as you expected.

## 2.15. The executing GUI menu

Once you have chosen the execute option in the GUI contextual menu a new menu will be available. This menu allows you to return to the editing mode, display the control numbers or paste objects which have been copied from another GUI window. The display control numbers option is very useful when you are writing the non-gui part of the macro, because it provides easy access to the control numbers for use with the `getpar` and `setpar` commands.

Make editable	Ctrl+Shift+E
Hide window	Ctrl+W
Close window	Shift+Esc
Paste control	Ctrl+V
Show control numbers	Ctrl+Shift+N

In some cases windows will be designed to block this menu. In this case Ctrl-left mouse click on the object to display the following window



This displays the parent window and control number in the title bar and then a bit of information about the control in the window contents. Disabled controls won't react to the Ctrl-click sequence – in this case use the Select button once the control information dialog has been displayed.

## 2.16. File Utilities

These functions can be found in the “Dialog macro files” section of the designer window.

- **load:** The *load* button reads a window layout file (which is just a macro) and then executes it, causing the window to be displayed.
- **save:** Once a window design has been completed it can be saved to a file using the *save* button. (The window must be in edit mode first). **Note** that if the window macro already exists, then you **must** save it back to the same file – saving it to a new file will lose all the non-gui parts of the macro. It is worth saving the window design frequently just in case something goes wrong.
- **view:** A window layout macro (extension .mac) can be viewed and edited by selecting the *view* button. Note that the macro associated with a GUI window can also be viewed by *shift double-clicking* on the window.

The recommended order of operations for modifying an existing GUI interface is as follows:

1. Run the macro to display the macro window.
2. Modify it using the designer.
3. Save the macro back to the *same* file.
4. Press the view button to update and edit the macro code if desired.

## 3. Editing the User-Interface Macro

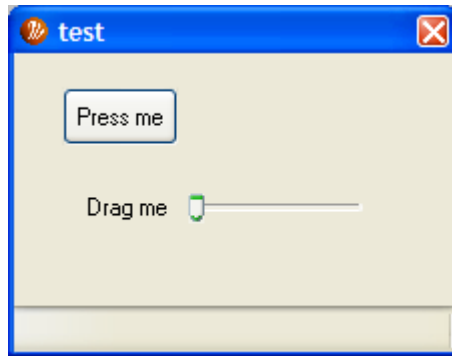
### 3.1. Call-back procedures

Once you have designed your graphical user interface you will need to add call-back procedures for each of the controls in the window. A call-back procedure contains the commands that will be run each time the control is activated (e.g. a button pressed or a slider dragged). These procedures can be appended to the control definition command, but more conveniently can be placed in a separate procedure if they are complex.

### 3.2. Saving a new window

The first step is to save the new window. Make sure it is in edit mode and then press the “save” button in the “Dialog macro files” section of the designer window. Select a filename for the new window definition and save it. The next step is to manually edit the window definition macro. Press the “view” button in the files section of the window

designer and select the window macro you have just saved. If the designer is no longer running, select the window macro from the "Open macro" menu option in the text editor file menu or double-click on the gui window while the shift-key is held down. What will appear in the edit window will be the instructions to generate the window, but no call-back procedures. For example the following window, consisting of a button, some static text and a slider is shown below along with the macro that generates it:



Note that the window definition is placed in its own procedure. This is done to allow re-editing of the window without disturbing the rest of the macro.

```
procedure(test)

    n = :windowdefinition()
    showwindow(n)

endproc()

procedure(windowdefinition)

    n = window("test", 100, 100, 225, 149)

    # Define all controls with basic parameters
    button(1, 24, 17, 58, 29, "Press me")
    slider(2, 81, 68, 100, 20, "horizontal")
    statictext(3, 36, 70, "left", "Drag me")
    statusbox(4)

    # Set other control parameters
    setpar(n,1,"tab_number",1)
    setpar(n,2,"tab_number",2)

endproc(n)
```

The name of this procedure – *windowdefinition* should *not* be changed, otherwise you will not be able to edit the window using the designer macro.

The *windowdefinition* procedure consists of the commands to generate the window, commands to generate controls in that window and then additional commands which set optional parameters (in this case the tab order). When controls are defined they are by default added to the last created window.

The main procedure – the one first called when you run this macro ('test' in this case), calls the *windowdefinition* procedure and then displays the window and its contents using the *showwindow* command. Note that the window command hides the window by default so that all its controls can be added and modified before they are made visible.

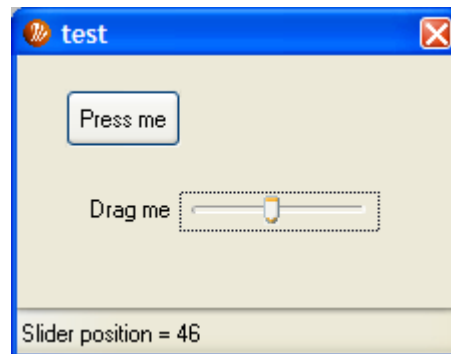
### 3.3. Adding call-back procedures

The call-back procedures for the button and slider are added at the end of each argument list. There can be any number of commands in the procedures but each command *must* be terminated with a semicolon. An example is given below:

```
button(1, 24, 17, 58, 29, "Press me",
    setpar(0,4,"text","Button pressed!"))
slider(2, 81, 68, 100, 20, "horizontal",
    position = getpar(0,2,"value");
    setpar(0,4,"text","Slider position = $position$"));
statictext(3, 36, 70, "left", "Drag me")
statusbox(4)
```

Note that a command list can appear on multiple lines. (Normally a new line or a semicolon ends a command).

When this macro is run the window is generated as before, however when you press the button, a message is now printed to the status bar, and when you drag the slider the current slider position is reported.



### 3.4. Getting and setting control parameters

The previous macro illustrated an important feature of the callback procedures - extracting current values from controls. To obtain the current text or value of a control you will use the `getpar` (get parameter) command. The first argument to this command is the window number of the command (as returned by the window command). Since this number may in fact be difficult to access from the macro level, you should usually just substitute zero (although the command `guiwinnr` can be used to access it if necessary). The `getpar` command will then know to use the current window number (that is the window which has last been selected by the user). The next argument is the control or object number, this is the first number in the control command - for example in this window the slider is control number 2. (You can find this number by choosing the "Show control numbers" option from the gui window context menu). The next argument specifies the parameter to get from the object. Normally it is either the "text" (a string) or the "value" (a number). So for example the command

```
position = getpar(0,2,"value")
```

will extract the current slider position from slider control 2 and place it in the variable `position`. Likewise

```
label = getpar(2,3,"text")
```

will return the static text label "Drag me" storing it in variable `label`.

For a full list of the possible parameters which can be extracted from different controls, see the Prospa help file "Command help" and commands `getpar` and `setpar`.

### 3.5. Using window variables

It is important to realise that all variables defined in the call-back procedures are local and are not available to other call-back procedures in the window or the main macro. However you *can* define variables that are accessible to *all* callback procedures. These are called *window variables* and should be declared as such *after* the window has been created with the window command. An example is shown below. Here the variable `position` is declared after the window has been defined and is initialised to zero by reading the current slider position.

```
procedure(testa)

    n = :windowdefinition()
    showwindow(n)
    windowvar(position)
    position = getpar(n,2,"value")

endproc()

procedure(windowdefinition)

    n = window("test", 100, 100, 225, 135)
    # Define all controls with basic parameters
    button(1, 24, 17, 58, 29, "Press me",
           setpar(0,4,"text","Current position is $position$");)
    slider(2, 81, 68, 100, 20, "horizontal",
           position = getpar(0,2,"value");
           setpar(0,4,"text","Slider position = $position$");)
    statictext(3, 36, 70, "left", "Drag me")
    statusbox(4)

    # Set other control parameters
    setpar(n,1,"tab_number",1)
    setpar(n,2,"tab_number",2)

endproc(n)
```

Note the order in which the commands are run. The window definition is completed first, and then the controls are generated. Next the window and all its controls are displayed (`showwindow`). The variable `position` is then declared and initialised. The macro then exits. However the window still exists along with its controls and so the final commands to be run are the call-back procedures. (Note that the `windowvar` command can appear anywhere *after* the window is defined.)

Normally you should avoid using window variables since they can be rather confusing. It is good programming practise to use local variables wherever possible, even if more code is required. In this example the same functionality could have been obtained by adding one more statement to the button control i.e.:

```
procedure(windowdefinition)

    n = window("test", 100, 100, 225, 135)
    # Define all controls with basic parameters
    button(1, 24, 17, 58, 29, "Press me",
           position = getpar(0,2,"value");
           setpar(0,4,"text","Current position is $position$");)
    slider(2, 81, 68, 100, 20, "horizontal",
```

```

        position = getpar(0,2,"value");
        setpar(0,4,"text","Slider position = $position$");)
statictext(3, 36, 70, "left", "Drag me")
statusbox(4)

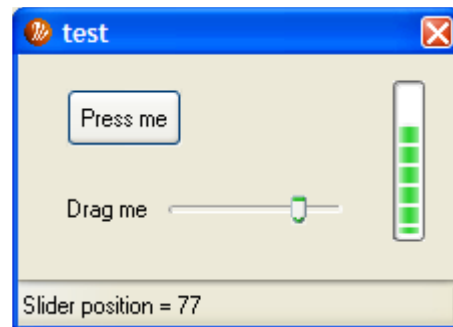
# Set other control parameters
setpar(n,1,"tab_number",1)
setpar(n,2,"tab_number",2)

endproc(n)

```

### 3.6. Using separate procedures for call-backs

If a large number of statements are likely to appear in the call-back procedure it improves readability to use a separate procedure rather than try and place all the commands directly in the control command. For example the following macro extends the previous code by adding a progress bar that displays the current slider position.



```

procedure(windowdefinition)

    n = window("test", 100, 100, 225, 135)

    # Define all controls with basic parameters
    button(1, 24, 17, 58, 29, "Press me",
        position = getpar(0,2,"value");
        setpar(0,4,"text","Current position is $position$");)
    slider(2, 69, 68, 100, 20, "horizontal", :report();)
    statictext(3, 24, 70, "left", "Drag me")
    statusbox(4)
    progressbar(5, 187, 13, 16, 80, "vertical")

    # Set other control parameters
    setpar(n,1,"tab_number",1)
    setpar(n,2,"tab_number",2)

endproc(n)

procedure(:report)

    position = getpar(0,2,"value")
    setpar(0,4,"text","Slider position = $position$")
    setpar(0,5,"value",position)

endproc()

```

The extra control was added using the designer macro. The window "test" was first displayed and then put into edit mode using the designer macro. The progress bar then was added and the window macro resaved and redisplayed in the editor. (**Note** be careful not to modify and save the *old* editor code otherwise your gui changes will be overwritten).

Finally using the editor, the progress-bar direction, which defaults to horizontal, was set to vertical and the dimensions set as required.

Note that because the macro was already in existence, only the windowdefinition procedure was overwritten during the editing process - the rest of the macro was untouched.

Finally a new procedure called "report" was added manually. This extracts the slider position and uses this to update the status bar (control 4) and the progress bar (control 5) using the `setpar` command. This command is the inverse of `getpar`, and is used to change a parameter in a window control or object. It takes 4 parameters, the window and object number (as for `getpar`), the parameter name and its new value. In this case the variable `position`, which is a number, is converted into text by the `setpar` command and then written to the window, updating as the slider is dragged. Using the "value" option, `setpar` is also used to set the position of the progress bar. Again, check out the Prospa help file "Command help" for a full list of possible parameters which can be changed.

### 3.7. Dialog boxes

An alternative GUI is the *dialog window*. In this case the window is presented to the user to gain some information and then additional processing is done *after* the window exits. An example of a dialog window is found in the `gettext` macro shown below.

```
# A window is displayed with some prompt text as a title
# The user types in their response and then presses
# the OK button. The program then returns the
# text entered to the calling program
```

```
procedure (gettext, prompt)
```

```
# Define dialog
```

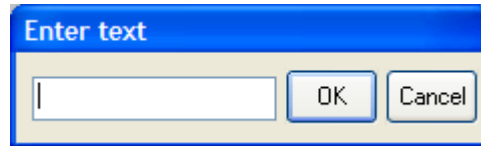
```
  n = window(prompt, -1, -1, 249, 70)
      setwindowpar(n, "type", "dialog")
      edittext(2, 9, 12, 110)
      button(1, 130, 8, 47, 26, "OK",
        result = getpar(0, 2, "text");
        closedialog(result);)
      button(3, 184, 8, 47, 26, "Cancel",
        closedialog("cancelled");)
      setpar(n, 1, "mode", "default");
      setwindowpar(n, "focus", 2)
```

```
# Display dialog and wait for text to be returned
```

```
text = showdialog(n)
```

```
endproc(text)
```





The above window was obtained by entering the following command into the CLI:

```
pr gettext("Enter text")
```

Then you can type text into the window and press the enter key (or click on the OK button) to exit and print the text you just typed.

Only three commands are required to convert the normal window into a dialog. First the window type is set to “dialog” using the `setwindowpar` command. Then instead of the `showwindow` command there is the `showdialog` command. Here the program remains until the OK or Cancel button are pressed, both of which execute the `closedialog` command. At this point the macro exits the `showdialog` command, returning the entered text (or a cancel flag). Additional commands can then be added to work on this information. Note that the dialog is a blocking window – you cannot use any other Prospa window while the dialog is being displayed.

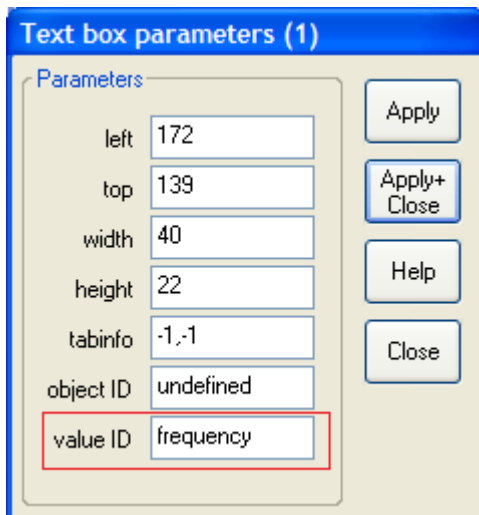
### 3.8. Extracting control values

Once a window has been defined we need some way of extracting the values associated with each control. This might be the text in a textbox or the position of a slider.

If we know the control number we can use the command `getpar` to extract the control value. This method has been described in the document "Editing the Window". However in some cases it is not desirable to use this method – especially if you are likely to modify the user interface a lot - in which case the control numbers might change. In this situation there are several other possibilities.

#### 3.8.1. Using the control's Value ID

Each control can have a value identity string or handle associated with it. This can be added in the design phase using the window editor. e.g. for a textbox used for controlling the frequency of an experiment we might name the value ID "frequency"



When the parent window is saved a new setpar option will be added for this control

```
setpar(n,3,"valueID","frequency")
```

If we need to extract the value of this control (and any other controls which have value IDs assigned we can use the following command

```
par = getctrlvalues(winNr)
```

This will return a parameter list of the form:

```
par = ["valueID = value", ... ]
```

This list can then be assigned to local variables using the command

```
assignlist(par)
```

"frequency" will then become a local variable with the last entered value assigned to it. In addition if we specify the expected data type and range we can run the

```
checkcontrols(winNr)
```

command before calling getctrlvalues to check if the entered values are appropriate.

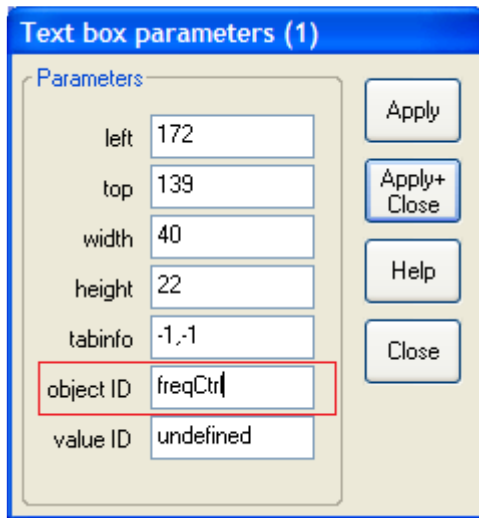
```
if(checkcontrols(winNr) != "ok")
    return
endif
par = getctrlvalues()
assignlist(par)
```

### 3.8.2. Using the control's Object ID

The getctrlvalue can only return the controls value – nothing else. In many cases this is enough but if you wish to modify the control or obtain other information then you need the controls object variable. This can be extracted directly when the control is defined e.g.

```
freqCtrl = textbox(3,70,130,40)
```

Or it can more conveniently be defined using a setpar command by adding an entry to the control editor.



When the parent window is saved a new setpar option will be added for this control

```
setpar(n,3,"objID","freqCtrl")
```

The window variable freqCtrl will then be defined automatically when the window is displayed – or if you need it earlier with the command

```
assignctrls(winNr)
```

Once you have the control object you can use arrow notation to access or modify window or control properties. For example:

```
freqCtrl->text("12.34")
freqCtrl->type("float")
freqCtrl->range([0,100])
```

```
text = freqCtrl->text()
type = freqCtrl->type()
range = freqCtrl->range()
```

The orange colouring indicates that this is a defined control command or parameter and that help is available (press F1 when the cursor is on this name).

An alternative method for obtaining the object is

```
freqCtrl = getobj(n,1)
```

Leaving off the 1 will return the object for the window n.

Lists of the accessible window and control parameters can be found in the Class documentation folder.

Note that although the same results can be obtained using the setpar/getpar syntax:

```

textbox(3,70,130,40)
setpar(n,1,"text","12.34")
setpar(n,1,"type","float")
setpar(n,1,"range",[0,100])

text = getpar(n,1,"text")
type = getpar(n,1,"type")
range = getpar(n,1,"range")

```

many people will find the variable name method easier to read. It also provides more logical access to parameters which are currently only available using the concept of current controls i.e. at any one time in Prospa there is a current text editor and current 1D and 2D plots. These are the last controls selected interactively by the user. For example you can modify the current plot in a macro using the command

```
curplot()
```

Commands such as plot, trace, axes, grid, title, xlabel and ylabel can then be used to access or modify the appearance of the selected plot. The only problem is that if the current plot changes (perhaps because the user has selected another plot) these commands will no longer access the original plot. However these commands can be accessed for a specific plot using the control object-variable concept:

```

pt1 = plot(1,10,10,500,200)
pt1->title->text("My plot")
pt1->title->font("Times Roman")
pt1->title->fontsize(12)

```

or alternatively

```

pt1 = plot(1,10,10,500,200)
title1 = pt1->title
title1->text("My plot")
title1->font("Times Roman")
title1->fontsize(12)

```

title1 is now a control *sub-variable*.

Even if the current plot changes these commands will still act on plot #1.