The ISIS

OPEN GENIE

Reference Manual

F A Akeroyd

R L Ashworth

S D Johnston

J M Martin

C M Moreton-Smith

D S Sivia

version 1.1

Acknowledgements

As with any large work there are too many people who have had some influence or given us help over the years to acknowledge everyone by name but without them Open GENIE would not be as it is today and we would like to thank them.

Open GENIE does rely heavily on the ever growing pool of software that has been generously made available at no charge to the scientific community and we would like especially to mention and gratefully acknowledge the contribution of Tim Pearson whose PGPLOT graphics package has become a de-facto standard within the scientific community and is used by Open GENIE, also Steve Byrne who put his time into writing the GNU smalltalk interpreter on which Open GENIE is based.

Contents

This manual describes Open GENIE in detail and is intended to provide detailed technical backup for the experienced user. It is also available on line at *www.isis.rl.ac.uk/GENIEReferenceManual* and in the Open GENIE distribution kit. This manual is complementary to the *Open GENIE User Manual* which may be more suitable for someone using or installing Open GENIE for the first time.

Chapter 1

Data Analysis Functions

Arguably the most important purpose of Open GENIE is as a tool for scientific data analysis. Not just for preliminary data analysis, but as a framework which provides, or from which can be called, all the tools necessary to perform a complete analysis of the data. The end result of an Open GENIE analysis will be numbers and visualizations of those numbers (plots!) suitable for interpretation with words in a final scientific report.

Currently, a large amount of data analysis is performed with a set of disparate FORTRAN programs which have to support their own routines to access data, to drive a user interface and to work with a variety of different graphics packages. Usually, it is not in the interest of the scientists themselves (or of the resulting science) that this load is carried by those who are most expert in the science. What is critical and is very much at the heart of the philosophy of Open GENIE is that *the scientist must have control over is how the data is processed*.

It may be, that a scientist is willing to trust the Open GENIE peak fitting routines (written by an acknowledged expert in the field), in this case, part of the analysis can be delegated to Open GENIE. Alternatively, and probably more the case at the moment, the scientists will want to continue to use their own analysis code for most things but can leave Open GENIE to provide the user interface, graphics and file access.

How is all this currently achieved by open GENIE?

Firstly, Open GENIE provides a powerful mechanism for allowing a scientist do define a new function which takes input from Open GENIE and returns output to Open GENIE transparently but in the meantime calls a C or FORTRAN subroutine to do the work. This mechanism is described fully in the *Module Reference* section of this manual and allows the code to run as if it is a compiled part of Open GENIE; it is fast, and any crashes within the code are safely caught by Open GENIE.

Secondly, Open GENIE provides routines which have been found to be useful in several different areas of analysis. These too are hard coded in C/C++ or FORTRAN for efficiency, they fall into two categories:

- *High level analysis procedures* - Routines which have some understanding of the aims of the analysis and the physical meaning of the data.

- *Low level analysis procedures* - More generic routines which can be used for a wide variety of purposes and allow users to build their own high level routines.

The difference between these two categories is blurred, but the distinction we use here can be described as follows:

High level procedures usually take an Open GENIE Workspace containing a fairly complex grouping of experimental parameters and data (See *Workspace operations* for more details about workspaces). These analysis functions tend to be fairly data specific and may call on several lower level functions. An example is the *Focus()* command which focuses neutron scattering spectra from a Time-of-Flight instrument.

On the other hand, the low level procedures generally take relatively few parameters and do not take workspaces, an example here is the *Peakfit()* routine which can be used to fit a peak to any data.

High Level Analysis Procedures

There are not yet many high level data analysis procedures. We have begun to add those which appear to be fairly common amongst groups of neutron scattering users at the ISIS facility and we will start to add these in earnest as the analysis needs become clear. If you have routines that you feel should be included, please contact us at *genie@isise.rl.ac.uk* and we will look at how best to integrate them for you, if you can supply them coded as a module, even better. The current routines are:

Focus()	Focus a range of spectra with given parameters
Integrate()	Integrate one or more spectra
Peak()	Interactive peak fitting of a workspace, see *Graphics commands*.
Rebin()	Rebin a workspace given the specified binning scheme
Sumspec()	Sum spectra in a given range from the specified file
Units()	Convert the units of a Time of Flight workspace as specified

Low Level Analysis Procedures

The low level analysis procedures also include may Open GENIE basic operations, for example *multiplication of arrays* as well as *mathematical functions* such as taking logs of a data array. All we document here are those which have been written specially for data analysis and don't fit in one of the other groups.

Peakfit()	Fits various peaks to data supplied in arrays
Peakgen()	Generates peak data from an X-array and peak parameters

High Level Analysis Procedure Reference

Focus()

Focus a range of spectra with given parameters

FOCUS()	**spectra**=*Range or IntegerArray* [**file**=*String*] [**detpars**=*Workspace*] [**specpars**=*Workspace*]	Focus a set of TOF spectra correcting for detector efficiency.
[:D]		Convert to D-spacing before summing
[:Q]		Convert to momentum transfer before summing
[:T]		Sum in time of flight
[:VERBOSE]		print out TOF parameters used in conversions

example:

```
# Focus a time of flight spectrum in D-spacing
# get the TOF parameters from the raw data file
>> w = focus( 1:17, "irs12838.raw" )
# Focus default file, every third period
>> assign 3045
>> refl = focus( 3:90@3, _, mydet )
```

Note: Focussing is done in D-spacing by default.

Focus

This command provides for focussing groups of neutron time of flight spectra from detectors at different scattering angles. This command, although written specifically for focussing multiple banks of detectors consecutively, can be used for single scanning detectors data as long as the data from consecutive runs has been stored as numbered spectra in a data format Open GENIE understands, for example an old GENIE-V2 intermediate file (see *Supported File Formats*). If the data has been stored in an Open GENIE intermediate file, the parameters can be stored in workspace format, see the example below.

If the detector efficiency parameters and/or T-O-F parameters are available from some other source, the Focus() command can be extended easily to read these parameters using the *Alias()* command as in the following example.

Example

```
# First alias the focus command so we can still get at it
alias "original_focus" "focus"
# Now define a new FOCUS procedure which can
# read the parameters from a convenient file.
PROCEDURE FOCUS
    PARAMETERS SPECTRA FILE=String
```

```
        QUALIFIERS /D /Q /T
        RESULT res
        LOCAL Tofpars Detpars
        # read these from a convenient intermediate files, could also use
        # the Asciifile() procedures to read any ASCII file format
        Tofpars = get("PARS", "parms.in3")
        Detpars = get("EFF", "detpars.in3")
        IF Q; res = original_focus:q(spectra, file, detpars, tofpars); ENDIF
        IF D; res = original_focus:d(spectra, file, detpars, tofpars); ENDIF
        IF T; res = original_focus:t(spectra, file, detpars, tofpars); ENDIF
    ENDPROCEDURE
```

The new focus command can be used transparently in place of the original command.

Parameters:

/Verbose

Print out parameters whilst focussing

/D, /Q, /T

Specify whether to focus in D-spacing, momentum transfer or by time.

Spectra (Range)

This parameter specifies the spectra to be focussed from the input file as a Range.

Open GENIE supports unique data types called *Interval* and *Range.* A Range or Interval can be used to specify a range of indices respectively to access multi-spectra data. For more information on specifying intervals, please see a description of the *Interval syntax*.

For the most general ability to specify a group of arbitrary spectra an Integer array of spectrum identifiers can be given as the the "Spectra" parameter.

File (String) [default = default input file]

The name of the input data file. This parameter will default to the default input file set by the *Set/File/Input* command if it is not specified.

Detpars (Workspace) [default = no detector efficiency correction]

Parameters for correcting for detector efficiency. These are given in a workspace with the following field names:

Field name	**Type**	**Description**
PRESSURE	Real	Gas pressure (atms)
GAS_SIGMA0	Real	Gas cross section at lambda 0 (cm-2)
PATHLENGTH	Real	Gas path length (cm)
WAVELENGTH	Real	Characteristic wavelength (cm)

WEIGHT	Real	Molecular weight of container atoms (g mol-1)
DENSITY	Real	Density of container (g cm-3)
THICKNESS	Real	Container wall thickness (cm)
WALL_SIGMA0	Real	Wall cross section at LAM0 (cm-2)
TYPE	String	Detector description, eg "He3 gas" - printed out during focussing

Specpars (Workspace) [default = values from raw file]

Time of flight parameters. These are given in a workspace with the following field names:

Field name	**Type**	**Description**
TWOTHETA	Realarray	Two theta scattering angles per spectrum (degrees)
EMODE	Integer or Integer Array	mode 0=inelastic, 1=incident, 2=transmitted
EFIXED	Real or Realarray	Fixed energy value or values if EMODE=2
L1	Real or Realarray	Primary flight path (m)
L2	Realarray	Secondary flight paths (m)

The arrays need to be of size "nspec" where "nspec" is the total number of spectra being focussed.

RESULT = (Restype)

The final focussed spectrum.

Integrate()

Integrate one or more spectra

INTEGRATE()	**wksp**=*Workspace* [**xmin**=*Real*] [**xmax**=*Real*]	Integrate a workspace (1d or 2d)
INTEGRATE()	**spectra**=*Interval or IntegerArray* [**xmin**=*Real*] [**xmax**=*Real*] [**file**=*String*]	Integrate spectra from a file

example:

```
# print the integrals of two spectra
>> printn integrate(1:2, "tfx00345.raw")
INTEGRATE: Integrating between 38000 and 78000
  Workspace []
  (
    error = [2341.4700 1027.1312 ] Array(2 )
    sum = [5482585.0 1054968.0 ] Array(2 )
  )
```

Note: The sum is always in total counts

Integrate

The Integrate() command is used to perform integration of one or more spectra.

Parameters:

Wksp (Workspace)

A one or two dimensional workspace to integrate.

Spectra (Range)

This parameter specifies the spectra to be integrated from the input file as a Range.

Open GENIE supports unique data types called *Interval* and *Range*, a Range or Interval can be used to specify a range of indices respectively to access multi-spectra data. For more information on specifying intervals, please see a description of the *Interval syntax*.

For the most general ability to specify a group of arbitrary spectra an Integer array of spectrum identifiers can be given as the the "Spectra" parameter.

Xmin, Xmax (Real) [default = spectra minimum & maximum respectively]

Optional interval over which to integrate. If the interval is not on a bin boundary, the counts are automatically corrected proportionally to the amount of the bin included.

File (String)

Allows specification of a file from which to get the spectra when used with the second form of the command. Otherwise, the input file is taken to be the file set by the *Set/File/Input* command.

RESULT = (Workspace)

The result of the integrate command is returned as a workspace with two fields (see example above). The "SUM" field contains either a single value or an array of integrals. The "ERROR" field contains the propagated error on the result, similarly, either a single value or an array of values.

Rebin()

Rebin a workspace given the specified binning scheme.

REBIN [/LIN] [/LOG]	**wksp**=*Workspace* **bound1**=*Real* **step1**=*Real* **bound2**=*Real* [**step2**=*Real*] [**bound3**=*Real*] [**step3**=*Real*] [**bound4**=*Real*]	Rebin a workspace in linear or logarithmic steps
REBIN	**wksp**=*Workspace* **xmin**=*Real* **xmax**=*Real*	Rebin over a range
REBIN	**wksp**=*Workspace* **xarray**=*Realarray*	Rebin to the given X values.

example:

```
# Read a spectrum and re-bin the same as an
# already loaded vanadium workspace
>> w = s(1)
>> w = rebin( w, vanadium.x )
```

Note: You can combine rebins by concatenating the results with the & operator.

Rebin

The rebin command performs what can loosely be described as an interpolation from one histogram into another histogram with the same integrated count, but with modified X-boundaries. Some numerical precision is lost during a re-binning operation and there is usually a degree of peak broadening so multiple re-binning is to be avoided.

The essential element in all the Rebin() command formats is to provide a new set of X-values for the spectra being re-binned. The first format allows ranges of linear or logarithmic steps to be specified explicitly. The second, simply acts to trim or expand a workspace to fit the given range, often useful when a series of spectra with different offsets need combining. The final form ensures the compatibility of two spectra by re-binning one spectrum to the X-values of the other.

Linear and logarithmic re-binning can be provided together by concatenating the different rebin ranges with the "&" operator (an end-on join of two spectra, see *workspace operations*). For example,

```
w = rebin:lin(s(1), 303.0, 1.0, 700.0) & rebin:log(s(1), 700.0, 0.01, 800.0)
```

(note that for an append to work, the ending value of the first range must be identical to the starting value of the next range).

Parameters:

/Lin

Linear rebinning between the boundaries specified in steps given by the "Stepn" parameters.

/Log

Gives logarithmic bin widths. The bin widths are calculated such that for for any given binning range

$$X_{n+1} - X_n = Step * X_n$$

where *Step* is the step value chosen for the range. For example, given the command

```
w = rebin:log(w, 10.0, 0.1, 15.0)
```

The bin boundaries generated will be [10, 11, 12.1, 13.31, 14.641, 15].

Wksp (Workspace)

The workspace to be re-binned

Boundn, **Step**n (Real)

Lower bound and step value (D_n above). There must always be one more bound specified than step value.

Xmin, Xmax (Real)

Interval over which to re-bin. If the interval is not on a bin boundary, the counts are automatically corrected proportionally to the amount of the bin included. This command leaves all complete bin boundaries within the range the same as before. It is used for extending or contracting the X-range of the spectra in a data safe way.

Xarray (Realarray)

An X-array to completely specify the desired new bin boundaries. This could be an array from a comparable spectrum or it may be automatically generated by the *Fill()* command as an arithmetic or geometric progression.

RESULT = (Restype)

The re-binned workspace. If the rebin command is used in the keyword form (i.e. Rebin w ...), workspace will be destructively re-binned. In the functional form, the input workspace will remain unaltered.

Sumspec()

Sum spectra in a given range from the specified file. (See also Focus/T)

SUMSPEC	[**spectra**=*Range or IntegerArray*] [**file**=*String*]	Sum spectra in Time-of-Flight

Note: Obsolete command, use Focus:T()

Sumspec

See the *Focus()* command for a description of the parameters, this command is functionally equivalent to *Focus/T* and is only kept for compatibility.

Units()

Convert the units of a Time of Flight workspace as specified

UNITS()	**wksp**=*Workspace* [**xmin**=*Real*] [**xmax**=*Real*]	Convert workspace units
[/C] or [/Channel]		To Channel numbers (one way)
[/D]		To D-Spacing (A)
[/E]		To Energy (meV)
[/LAM]		To Wavelength (A)
[/Q]		To Momentum Transfer (A^{-1})
[/SQ]		To (Momentum Transfer)^2 (A^{-2})
[/T]		To Time of Flight (us)
[/LA1]		To primary flight path wavelength (A)
[/W]		To energy transfer E1-E2 (meV)
[/WN]		To energy transfer E1-E2 (cm-1)
[/TAU]		To reciprocal time-of-flight (us/m)

example:

```
# Read in a spectrum whilst converting to Wavelength
>> w = units:Lam( s(1) )
```

Note: TOF Parameters must be set correctly in the workspace (or set Set/Par).

Units

The Units() command provides for units conversion of Time-of-Flight spectra. The conversions usually rely on specific parameters being set correctly in the input workspace. These parameters may be read in from the data file, or they may have to be set in the workspace before the Units() command is carried out. To find the parameters see the section on *workspace operations*.

A good way of ensuring parameters are read correctly is to overload the *S()* command, see *Alias()* and the example given for overriding the *Focus()* command.

Parameters:

Wksp (Workspace)

The Time-of-Flight workspace to convert.

Xmin, Xmax (Real) [default = converted spectra minimum & maximum respectively]

Optional interval to select the section of the workspace to convert (specified in the final units). For energy only, the default is -1000 to 5000meV.

RESULT = (Workspace)

The converted workspace. If the keyword syntax is used, the workspace will be converted destructively (e.g. Units/C w).

Low Level Analysis Procedure Reference

Peakfit()

Fits various peaks to point mode data supplied in arrays.

PEAKFIT()	**x**=*Realarray* **y**=*Realarray* **e**=*Realarray* [**pars**=*Realarray*] [**ipropt**=*IntegerArray*]	Fits a selection of peaks to supplied data.
[:GAUSS]		Gaussian
[:GEXP]		Gaussian convolved with exponential
[:LOREN]		Lorentzian
[:LEXP]		Lorentzian convolved with exponential
[:VOIGT]		Voigt
[:VEXP]		Voigt convolved with exponential
[:POLY]		Polynomial of a specified degree

example:

```
# Read some data and fit a peak to it
# remembering to convert from histogram
# to point data.
>> w=s(1)
>> res = Peakfit:Lexp(centre_bins(w.x), w.y, w.e)
```

Note: Centre_bins() converts to point mode.

Peakfit

The peakfit command is designed to give full access to a good selection of peak fitting routines. It is designed so that it can be used in conjunction with the *Peakgen()* command which allows a regeneration of the fitted peak, usually to allow a graphical comparison of the goodness of the fit.

Parameters:

X, Y, E (Realarray)

One dimensional X, Y, and error data arrays of the same length. Note that a lot of data in Open GENIE is in histogram format so it is important to ensure that the X-array is converted before attempting to fit a peak. The *Centre_bins()* function is provided for this purpose.

Pars (Realarray) [default = best guess line fit]

Optional input parameter that allows an initial guess or fixing of one or more of the peak parameters. This operates in conjunction with the corresponding values in "Iprompt". The length of the pars array differs depending on the number of peak parameters required for a particular fit (see below for descriptions of the individual fitting routines).

Iprompt (Integerarray) [default = no initial estimates]

Controls the treatment of the corresponding parameter in "Pars". This array consists of a set of integer flags.

0 = no initial estimate
1 = use parameter as a guess
2 = fix parameter value as given

By using these two parameter arrays, it is possible to completely control the operation of the peak fitting routines, However, for simple fitting it is not necessary to supply a "Pars" or "Iprompt" array.

RESULT = (Workspace)

The result of the Peakfit() command is returned as a workspace containing the following fields:

Field name	**Type**	**Description**	
IGOOD	Integer	Status value for goodness of fit:	
		< -1	Terrible
		-1	Maybe OK
		>= 0	Good
PARS	Realarray	Best estimates of the parameters	
SIGPAR	Realarray	1-sigma error-bars for PARS (may be negative if IGOOD < 0)	

Peakfit:Gauss

Estimates the parameters of a Gaussian-peak sitting on a straight-line background, given a pertinent set of data:

```
    y(x) = Mx + C + A*gauss(X0,SIGMA) ,
where
                            1            [ - (x-X0)**2  ]
    gauss(X0,SIGMA) = ---------------- exp|--------------| .
                      SIGMA*sqrt(2*pi)    [ 2 * SIGMA**2 ]
```

Parameters:

Pars (Realarray)

Pars[1] = M
Pars[2] = C
Pars[3] = A
Pars[4] = X0
Pars[5] = SIGMA

Peakfit:Gexp

Estimates the parameters of a peak, consisting of a Gaussian convolved with a sharp-edged exponential, sitting on a straight-line background, given a pertinent set of data:

```
y(x) = Mx + C + A*gsexp(X0,SIG,TAU) ,
```

where

```
                           1            [ - (x-X0)**2 ]
gsexp(X0,SIG,TAU) =  -------------- exp|-------------|
                     SIG*sqrt(2*pi)     [  2 * SIG**2 ]
```

Convolved with

```
{        0           if x/TAU < 0
{                                     .
{ exp(-x/TAU)/|TAU|  if x/TAU > 0
```

Parameters:

Pars (Realarray)

Pars[1] = M
Pars[2] = C
Pars[3] = A
Pars[4] = X0
Pars[5] = SIG
Pars[6] = TAU

Peakfit:Loren

Estimates the parameters of a Lorentzian-peak, sitting on a straight-line background, given a pertinent set of data:

```
y(x) = Mx + C + A*lorentz(X0,GAMMA) ,
```

where

```
                                GAMMA
lorentz(X0,GAMMA) = ---------------------------   .
                    pi [ (x-X0)**2 + GAMMA**2 ]
```

Parameters:

Pars (Realarray)

Pars[1] = M

Pars[2] = C
Pars[3] = A
Pars[4] = X0
Pars[5] = GAMMA

Peakfit:Lexp

Estimates the parameters of a peak, consisting of a Lorentzian convolved with a sharp-edged exponential, sitting on a straight-line background, given a pertinent set of data:

```
y(x) = Mx + C + A*lzexp(X0,GAM,TAU) ,
```

where

```
                                   GAM
lzexp(X0,GAM,TAU) =   -------------------------
                      pi [ (x-X0)**2 + GAM**2 ]
```

Convolved with

```
{        0           if x/TAU < 0
{                                       .
{ exp(-x/TAU)/|TAU|  if x/TAU > 0
```

Parameters:

Pars (Realarray)

Pars[1] = M
Pars[2] = C
Pars[3] = A
Pars[4] = X0
Pars[5] = GAM
Pars[6] = TAU

Peakfit:Voigt

Estimates the parameters of a Voigt-peak sitting on a straight-line background, given a pertinent set of data:

```
y(x) = Mx + C + A*voigt(X0,SIG,GAM) ,
```

where

```
                             1           [ - (x-X0)**2 ]
voigt(X0,SIG,GAM) =  -------------- exp|-------------|
                     SIG*sqrt(2*pi)    [  2 * SIG**2 ]
```

Convolved with

```
          GAM
--------------------  .
pi [ x**2 + GAM**2 ]
```

Parameters:

Pars (Realarray)

Pars[1] = M
Pars[2] = C
Pars[3] = A
Pars[4] = X0
Pars[5] = SIG
Pars[6] = GAM

Peakfit:Vexp

Estimates the parameters of a peak, consisting of a Voigt convolved with a sharp-edged exponential, sitting on a straight-line background, given a pertinent set of data:

```
y(x) = Mx + C + A*vtexp(X0,SIG,TAU) ,
```

where

```
                          1            [ - (x-X0)**2 ]
voigt(X0,SIG,GAM) =  -------------- exp|-------------|
                     SIG*sqrt(2*pi)    [  2 * SIG**2 ]
```

Convolved with

```
         GAM
  --------------------  .
  pi [ x**2 + GAM**2 ]
```

Convolved with

```
 {        0           if x/TAU < 0
 {                                     .
 { exp(-x/TAU)/|TAU|  if x/TAU > 0
```

Parameters:

Pars (Realarray)

Pars[1] = M
Pars[2] = C
Pars[3] = A
Pars[4] = X0
Pars[5] = SIG
Pars[6] = GAM
Pars[7] = TAU

Peakfit:Poly

Estimates the polynomial coefficients of a peak or background by attempting to fit an nth degree polynomial, the degree of the polynomial fit is given by the number of parameters supplied to pars, for this routine "Pars" is not optional.

```
y(x) = p1 + p2*x + p3*x^2 + p4*x^3 + ...
```

For example

```
>> pars = dimensions(3)                       # create an array
```

```
>> fill pars 0.0                          # fill with 0.0
>> fit = Peakfit:Poly(x, y, e, pars)      # quadratic fit.
```

Parameters:

Pars (Realarray)

Pars[1] = p1
Pars[2] = p2
Pars[3] = p3
Pars[4] = p4
Pars[5] = p5
Pars[6] = p6

Peakgen()

Generates peak data from an X-array and peak parameters.

PEAKGEN()	**x**=*Realarray* **pars**=*Realarray*	Fits a selection of peaks to supplied data.
[:GAUSS]		Gaussian
[:GEXP]		Gaussian convolved with exponential
[:LOREN]		Lorentzian
[:LEXP]		Lorentzian convolved with exponential
[:VOIGT]		Voigt
[:VEXP]		Voigt convolved with exponential
[:POLY]		Polynomial

example:

```
# Produce data to show goodness of a previous fit
# res is the result from a previous peakfit() command.
>> y_fit = Peakgen:Lexp(centre_bins(w.x), res.pars)
>> y_goodness = w.y - y_fit  # difference for plotting
```

Note: Centre_bins() reduces X-array length by one and picks bin centres.

Peakgen

The Peakgen() command is designed to be used in conjunction with the *Peakfit()* command, it allows a regeneration of the fitted peak, usually to allow a graphical comparison of the goodness of the fit.

Note that it is not essential to have done a Peakfit() command for the Peakgen() command to work. By supplying an X-array and suitable parameters, any of the peak types may be generated.

Parameters:

X (Realarray)

One dimensional X array, usually the same X-array which was supplied to the corresponding Peakfit() command.

Pars (Realarray)

The parameters defining the peak. See the *Peakfit()* command for a description of the individual parameters for each type of peak.

RESULT = (Realarray)

The result of the Peakgen() command is an array of Y-values, the same length as the supplied X-array which if plotted will give the required peak.

Chapter 2

Graphics Commands

The graphics command section is divided into sections on *Primitive commands* and *High Level commands*. The primitive commands provide the most flexibility and are suitable for writing sophisticated programs in the GENIE command language (GCL). The high level commands are provided to give a simple to use set of commands which can be used by a less experienced user to get pictures of their data on the screen quickly and easily.

It is quite possible to mix both sorts of command once experience has been gained with the way in which the Open GENIE graphics system operates.

High Level Commands

The High level graphics commands are listed below. These are based on similar commands used in the original GENIE-V2 program although the command syntax and functionality are enhanced.

Alter	Alters graphics control parameters
Cursor	Emulates Genie II cursor routine
Display	Display a new workspace, or redisplay previous one
Hardcopy	Saves a chosen picture into a file
Limits	Sets graph limits for DISPLAY command
Multiplot	Do a multiplot (Genie II style)
Peak	Fit a peak shape to user data
Plot	Overplots an existing graph (Genie II style)
Toggle	Toggles graphics control parameters
Zoom	Uses cursor to zoom in on a plot

A more general overview of the operation of these commands is available in the "*User Notes*" section of this manual

Primitive Commands

The primitive commands listed here give maximum control over the operation of the Open GENIE graphics sub-system. As a result they are slightly less easy to use for the novice user. For example, before any drawing primitives can be used, the *Device()* command must be used to open a display device.

Examples of the use of many of these commands may be found in the GCL procedures in the examples directory (usually /usr/local/genie/examples on UNIX or [OPENGENIE.EXAMPLES] on VMS). Where possible, an example or two showing a possible use of the primitive command is given as part of the reference section for that command.

For a basic introduction to the primitive graphics system, please see the advanced graphics user note in the "*User notes*" section of this manual.

Note on Picture/Window/Index (PWI) references

The Open GENIE graphics sub-system supports two ways of referring to graphical objects, direct references to the object itself (stored in a variable), and Picture/Window/Index references which allow the location of graphical objects by the picture, window and device within which they are drawn.

Direct references

All objects including pictures, windows, and individual graphics primitives can be referred to by keeping a record of the object itself when it is created, for example

```
my_pict = picture()
```

The stored object "my_pict" in the example here is only useful for the graphics system and is of a special type "GObject". Several of the graphics commands take GObjects when it is necessary to identify a particular object, for example, when altering it.

Although they are not used in the rest of Open GENIE, graphical objects such as "my_pict" can be identified or debugged using the normal *Print()* commands and tested for equality using the "=" operator.

PWI references

Alternatively, an object may be "found" using its PWI reference and one of the object locating functions *Pic()*, *Win()*, *Dev()* or *Obj()*. Each of these functions take one or more integers which locate the objects within the hierarchy of the graphics system by numbers. These numbers are based on the order in which the graphical objects were created. For example, a reference such as Obj(2,3,4) is saying "the 4th object created in the 3rd window created in the second picture created". For all of these functions, a value of 0 represents the current item (-1 for the last item of that type created). The result of all of these locating functions is a Gobject type which can be used as described above.

The primitive graphics commands are listed alphabetically in their functional groups below.

Control of graphics objects

Delete	Deletes unwanted graphics objects from memory
Pic_add	Copy graphical objects from picture to picture
Select	Select a current device, window or picture
Redraw	Redisplay previous pictures or objects
Undraw	Undraw a primitive graphics operation.

Devices, Pictures and Windows

Device	Selects and controls graphics devices
Picture	Creates a Picture
Window	Command for compatibility with older programs
Win_autoscaled	Controls and creates autoscaled windows for plotting
Win_multiplot	Controls and creates multi_plot windows for plotting
Win_scaled	Controls and creates scaled windows for plotting
Win_twod	Controls and creates twod windows for plotting
Win_unscaled	Controls and creates unscaled windows for plotting
New_zoom	Zooms in on a graphics device (experimental - very !!)

Graphics input

Getcursor	Cursor routine for a window or a picture

Simple drawing and plotting

Draw	Command for compatibility with older programs
Axes	Displays axes on the graphics device
Errors	Displays error-bars on a plot
Graph	Plots a Graph on the graphics device
Graticule	Displays a Graticule on the graphics device
Histogram	Plots a histogram on the graphics device
Labels	Displays x/y labels on the graphics device (combined with axes).
Line	Displays a line on the graphics device

Markers	Displays markers on the graphics device
Polygon	Displays a polygon on the graphics device
Text	Displays some text on the graphics device
Title	Displays a Title on the graphics device

Two-Dimensional plotting

Colour cell contouring

Cell	Plots a cell on the graphics device
Cell_array	Plots a cell_array from supplied arrays
Cell_function	Plots a cell_function from supplied function
Cell_wedge	Displays a cell_wedge on the graphics device

Contour plotting

Contour	Plots a contour on the graphics device
Contour_array	Plots a contour_array on the graphics device
Contour_function	Plots a contour_function on the graphics device
Contour_label	Labels the contours in the contour plot

Multiple plots

Multi_plot	Controls multiplot creation and plotting

Utility Commands

Defining colours

Colour	Allows user definition of individual colours
Colourtable	Manipulate graphics colour tables

Locating objects

dev	Gets a device object from its number
obj	Gets a primitive object from its location
pic	Gets a picture object from its number
win	Gets a window object from its location

High Level Graphics Command Reference

Alter()

Sets various defaults for the subsequent Display(), Plot() and Multiplot() commands. These defaults only apply to high level commands and will not affect a primitive command like *Graph()*.

ALTER/STATUS		Reports the values of all settings.
ALTER/BINNING	**p1**=*Integer*	Sets bin grouping for plotting with Display() and Plot()
ALTER/DEVICE	**p1**=*String*	Change default interactive device
ALTER/FONT	**p1**=*Font*	Set text font
ALTER/HARDCOPY	**p1**=*String*	Hardcopy device
ALTER/LINECOLOUR	**p1**=*Colour*	Axes/box colour
ALTER/LINETYPE	**p1**=*LineStyle*	Line style for line plots
ALTER/LINEWIDTH	**p1**=*Real*	Overall line thickness
ALTER/MARKERS	**p1**=*Marker*	Sets the marker plot symbol
ALTER/MARKERSIZE	**p1**=*Real*	Size of markers on a Display:Markers command
ALTER/PLOT	**xmin**=*Real* **xmax**=*Real* **ymin**=*Real* **ymax**=*Real*	Change plot size
ALTER/PLOTCOLOUR	**p1**=*Colour*	Data plot colour
ALTER/SIZE	**p1**=*Real*	X window display size (scale factor)
ALTER/TEXTHEIGHT	**p1**=*Real*	Text height
ALTER/TEXTCOLOUR	**p1**=*Colour*	Label/title colour

example:

```
# Change the axis colour to blue
>> Alter/linecolour $BLUE
```

Note: See also the Toggle() command for changing other plot parameters

---

Alter/Status

Shows the current values of any settings which may be changed using the Alter() command.

Alter/Binning

This command alters the bin grouping value used by the *Display()* command when displaying a histogram or bin-mode workspace. When the bin grouping value is set to a value greater than one, the display command groups the data by averaging the values of n-bin groups starting from the first bin in the histogram. This averaging can be viewed as a rudimentary form of smoothing the data which is done on the fly, each time the display command is used. It is possible to apply the same operation to a standard workspace permanently, see the *Groupbins()* command for more information.

Parameters:

P1 (Integer)

Number of bins to group together, no grouping occurs if this value is set to 1.

Alter/Device

This command changes the device to which the output of the display command will be sent.

Parameters:

P1 (String)

This is the name of the *supported graphics device* to which to set the display. Following the Alter/Device command all high level graphics commands will use the selected device.

Alter/Font

This command changes the default font used by the *Display()* command. This will change the font for all the text used in the display.

Parameters:

P1 (Font)

This is the name of a *supported font* to which to set the display. After an Alter/Font command all high level graphics commands will use new font.

Alter/Hardcopy

This command changes the default hardcopy device to which the output of display command is sent when a *Hardcopy()* command is issued.

Parameters:

P1 (String)

This is the name of the *supported graphics device* that will be used to save a hardcopy of the plot.

Alter/Linecolour

This command changes the default colour the *Display()* command uses for drawing the axes and surrounding boxes.

Parameters:

P1 (Colour)

This is a colour variable returned by one of the colour functions or listed as one of the *supported colours*.

Alter/Linetype

This command changes the default line type the *Display()* and *Plot()* commands use for drawing line plots. Useful for separating plots on monochrome terminals.

Parameters:

P1 (LineStyle)

This is a line style listed as one of the *supported linestyles.*

Alter/Linewidth

This command changes the default line width used by the *Display()* and *Plot()* commands for drawing line plots.

Parameters:

P1 (Real)

This is a real multiplier for the default line plot width of 1 unit. The width of the line may vary on different graphics devices.

Alter/Markers

This command changes the default marker used on a Marker plot.

Parameters:

P1 (Marker)

This is one of the available markers listed as one of the *supported markers*.

Alter/Markersize

This command changes the default marker size used by the *Display()* and *Plot()* commands for drawing marker plots.

Parameters:

P1 (Real)

This is a real multiplier for the default marker size of 1 unit. The size of the marker may vary on different graphics devices.

Alter/Plot

This command changes the area of the display device which will be taken up by a *Display()* or *Plot()* command. The limits of the box are specified in device co-ordinates in the range 0.0-1.0 where the the values (0.0, 0.0) and (1.0, 1.0) represent the largest square available on the plotting device. It may be possible to draw slightly outside this area for devices with rectangular aspect ratios.

Parameters:

Xmin (Real), **Xmax**=(Real), **Ymin**=(Real), **Ymax**=(Real)

Normalised device co-ordinate values in the range 0.0 to 1.0. Xmin < Xmax, Ymin < Ymax.

Alter/Plotcolour

This command changes the default colour the *Display()* command uses for drawing the data plot (line, histogram or markers or errors).

Parameters:

P1 (Colour)

This is a colour variable returned by one of the colour functions or listed as one of the *supported colours*.

Alter/Size

Changes the size of the display device by a scaling factor. 1.0 is the default scaling factor.

Parameters:

P1 (Real)

Multiplier for the display device size.

Alter/Textheight

This command changes the default text height used by the *Display()* and *Plot()* commands.

Parameters:

P1 (Real)

This specifies the text height as a multiplier of 100th of the height of the display. The default text height is 3.0 or 3 hundredths of the display height.

Alter/Textcolour

This command changes the default colour the *Display()* command uses for drawing text.

Parameters:

P1 (Colour)

This is a colour variable returned by one of the colour functions or listed as one of the *supported colours*.

Cursor()

Displays a cursor on the graphics screen to allow interactive annotation.

CURSOR	[**xval**=*Real*] [**yval**=*Real*]	Displays a cursor and waits for key press: X-display X coordinate" Y-display Y coordinate" P-display both coordinates" T-add text" L- lower left corner of box" U-draw box to upper right point" E-exit
[/HORIZONTAL]		Annotation text horizontal
[/VERTICAL]		Annotation text vertical

example:

```
# Display a cursor near the expected position
# of a peak in world coordinates.
>> Cursor/Horizontal 10000.0 50.0
```

Cursor

This command is used mainly for annotating a plot whilst it is on the screen. An obvious use is for tagging peaks with the data value in the X or Y direction at the point selected by the cursor.

The command enters a sub-system where only certain keystrokes are recognized. Annotation operations as listed above can be carried out repeatedly until the E key is pressed to terminate the command.

Parameters:

/Vertical [default]

Sets the default for any annotation text to be drawn vertically (from bottom to top).

/Horizontal

Sets the default for any annotation text to be drawn horizontally, this is the default if no qualifiers are given.

Xval, Yval (Real)

Optional starting values for the cursor position. If a peak is known to be somewhere near the middle of the screen, for example, it can save unnecessary cursor movement. The default position for the cursor to appear is at (0,0) in the current window.

Display()

Display a graphical plot of spectra in a workspace.

DISPLAY	[**wksp**=*Workspace*] [**xmin**=*Real*] [**xmax**=*Real*] [**ymin**=*Real*] [**ymax**=*Real*] [**linecolour**=*Colour*] [**linewidth**=*Real*] [**textcolour**=*Colour*] [**textheight**=*Real*]	Display the workspace within the given range.
[/HISTOGRAM]		Display as a histogram
[/LINE]		Display as a line plot
[/MARKERS]		Display as markers
[/ERRORS]		Display as error bars

example:

```
# Display a spectrum but change the title first
>> wm = s(10)
>> wm.title = "This is my Nobel plot!"
>> Display wm xmin=1.0e4 xmax=7.0e4
```

Note: Other display parameters may be altered using the Alter() and Toggle() commands.

Display

The display command is the high level command used to plot one or more spectra from a workspace in a single operation. By default, the Display() command will assume that the plot is to be done as a histogram. The plot will be automatically scaled and titled with other relevant information from the workspace. Most features of the Display() command may be customized either by using the command line parameters on Display() itself or by modifying characteristics of the plot using the *Alter()*, *Toggle()* or *Limits()* commands.

Parameters:

/Histogram

This is the default for the Display() command when given bin-mode (histogrammed) data. It will plot a spectrum making the assumption that it is a valid histogram (i.e. binned data with one more X value than Y value). The data values are represented by the height of the corresponding bin. Remember that several commands can actually affect the height of bins unexpectedly by changing the binning method and/or data. Examples of these are *Alter/Binning* and *Units/Channel*.

If point-mode data is supplied with the /Histogram qualifier an error will be reported as it is not possible for Open GENIE to guess the assumptions you may want to make about the conversion (an extra X value would have to be generated by interpolation).

/Line

This is the default for the Display() command when given point-mode data. It will plot a spectrum as if it is a numerically calculated function (i.e. with the same number of X-values as Y-values) and join the points with a straight lines. If histogram data is displayed with the /Line qualifier, the data will undergo a transformation to take the bin centre positions as the X-values for the data points (see the *Centre_bins()* function). There are situations where you may not wish this behaviour to be the default (e.g. for logarithmic binning). Generally, it is safer to transform the data yourself into point-mode and then use Display to plot it.

/Markers

This is identical to the /Line qualifier in operation except that instead of joining the data points with lines, marker symbols are used instead. See *Marker Types* for a list of available styles.

/Errors

This is identical to the /Line qualifier in operation except that instead of joining the data points with lines, error bars are generated. These are centered on the data points and have a size scaled from the values of the data error values contained within the workspace E array.

Wksp (Workspace)

This gives the workspace to be plotted. As long as the workspace is a valid Open GENIE workspace containing one or more spectra this command will produce a display. In later version of Open GENIE this will also take multi-spectra workspaces to produce multiplots (see *Multiplot()*).

For convenience, if the Display() command is used without a workspace being specified, the last workspace to be plotted will be used.

Xmin, Xmax, Ymin, Ymax (Real)

These parameters can be used to set fixed limits for the display, for more details see the *Limits()* command.

Linecolour (Colour)

This parameter changes the colour for drawing the axes and surrounding boxes, see *Alter()* for details.

Linewidth (Real)

This parameter changes the line width used for drawing line plots, see *Alter()* for details.

Textcolour (colour)

This parameter changes the colour used for drawing text, see *Alter()* for details.

Textheight (Real)

This parameter changes the text height used for drawing text, see *Alter()* for details.

Hardcopy()

Makes a copy of picture on a hardcopy device.

HARDCOPY	[**filename**=*String*] [**devtype**=*String*] [**picture**=*Gobject*]	Creates a hardcopy of a previously drawn picture

example:

```
# Save a postscript copy of the screen
>> Hardcopy
# Save a copy of the last but one picture as
# colour postscript (found with Redraw/Info)
>> Hardcopy filename="test.cps" picture=my_pict
```

The Hardcopy command allows a copy to be made of either the currently displayed window, or an earlier picture.

The supported hardcopy devices on most machines include "PS" Postscript, "CPS" Colour Postscript and "HPGL" HPGL driver for 7475A For full details of supported devices on each machine type see *Supported Graphics Devices*.

The "filename" and "devtype" parameters also set the default for further calls to the Hardcopy command so normally just typing Hardcopy is sufficient to make a hardcopy of the current screen.

Parameters:

filename (String)

The name of a file into which to put the hardcopy, the default is "GENIE.PS".

devtype (String)

One of the supported hardcopy graphics devices.

picture (Gobject)

The picture to be copied. By default picture 0 is selected which is the last picture to be created.

Limits()

Sets axis limits for the Display() and Plot() commands.

LIMITS [/DEFAULT]		Sets X and Y axes to autoscale
LIMITS/X	[**xmin**=*Real*] [**xmax**=*Real*]	Sets X axis limits
LIMITS/Y	[**ymin**=*Real*] [**ymax**=*Real*]	Sets Y axis limits
LIMITS/NO_AUTO	[**xmin**=*Real*] [**xmax**=*Real*] [**ymin**=*Real*] [**ymax**=*Real*]	Sets X and Y axis limits

example:

```
# Set the limits before doing a display
>> Limits/X xmin=10.0E4 xmax=50.0E4
>> Display s(1)
```

Note: If no data appears on a Display(), typing "Limits" may help find it!

Limits

The Limits() command is used to gain independent control over the scaling of the axes used in the *Display()* and *Plot()* commands. Settings made using the command remain in effect until another Limits() command is issued.

Parameters:

/Default

This qualifier enables automatic limit scaling for the Display() command. If for some reason the data has disappeared off the plot, using Limits/Default will ensure any data is plotted in the viewable area. This is the default qualifier so just typing "Limits" will perform a Limits/Default.

Limits/X

Sets explicit X limits for the *Display()* and *Plot()* commands to adhere to. Note that the actual setting of the limits will still be dependent on whether rounding is switched on for the X axis (see *Alter()*)

Parameters:

Xmin, **Xmax** (Real)

Real values for the lower and upper bounds of the visible data region in data co-ordinates.

Limits/Y

Sets explicit Y limits for the *Display()* and *Plot()* commands to adhere to. Note that the actual setting of the limits will still be dependent on whether rounding is switched on for the Y axis (see *Alter()*)

Parameters:

Ymin, **Ymax** (Real)

Real values for the lower and upper bounds of the visible data region (in data co-ordinates).

Limits/No_auto

Sets explicit X and Y limits for the *Display()* and *Plot()* commands to adhere to. Note that the actual setting of the limits will still be dependent on whether rounding is switched on for either or both axes (see *Alter()*)

Parameters:

Xmin, **Xmax, Ymin, Ymax** (Real)

Real values for the lower and upper bounds of the visible data region in data co-ordinates.

Multiplot()

Plots multiple spectra on one set of axes using a hidden line removal

MULTIPLOT	**spectra**=*Interval or IntegerArray* [**ygap**=*Real*] [**file**=*String*] [**linecolour**=*Colour*] [**linetype**=*LineStyle*] [**linewidth**=*Real*] [**textcolour**=*Colour*] [**textheight**=*Real*]	plot the spectra specified from a file.
MULTIPLOT	**wa**=*WorkspaceArray* [**ygap**=*Real*] [**file**=*String*] [**linecolour**=*Colour*] [**linetype**=*LineStyle*] [**linewidth**=*Real*] [**textcolour**=*Colour*] [**textheight**=*Real*]	plot the spectra in the workspace array.

example:

```
# Do multiplots from a file
>> multiplot 1:20 file="mydata.raw" ygap=10.0
>> multiplot 1:300@3          # plot every third spectrum
# Make and plot a workspace array
>> ww = Get:Array(1:20)       >> multiplot wa=ww linecolour=$green
```

Note: multiplot from a file is more efficient in terms of memory

---

Multiplot

The multiplot command provides an easy to interpret display of multiple spectra on the same X-axis. It is useful for comparing groups of spectra which although essentially similar may show a trend across the group (e.g. a phonon effect with temperature). The command works by producing a pseudo Y-axis where subsequent plots are placed at an artificial Y offset value (ygap). Individual spectra are overplotted using hidden line removal. Currently, multiplot only plots as a histogram, the binning is controlled with the *Alter()* command.

Parameters:

Spectra (Range)

When the Multiplot() command is used with this parameter specified, it is assumed that the data source for the multiplot has already been specified as a default input source, see *Set/File*, or is given with the "file" parameter to this command.

Open GENIE also supports unique data types called *Interval* and *Range,* a Range or Interval can be used to specify multiple indices or a range of indices respectively to access multi-dimensional data. For more information on specifying intervals, please see a description of the *Interval syntax*. For the most general ability to specify a group of arbitrary spectra an Integer array of spectrum identifiers can be given as the the "Spectra" parameter. The spectra will be plotted in the order specified.

Wa (WorkspaceArray)

If this form of the command is used, the data to be plotted is supplied in the form of multiple spectra stored in a Workspace array. The spectra must all be binned identically on the same X-axis for the multiplot command to be able to work.

Ygap (Real)

This parameter sets the spacing artificially added to separate the multiple spectra on the Y-Axis. It is specified in the same coordinates as the data and is calculated automatically by the Multiplot() command if it is not specified. The real Y height of a point in any spectrum on a multiplot is given by *yvalue = measured-height - ygap * spectrum-number* where spectrum-number is the nth spectrum in the order of plotting.

file (String)

Specifies a data file other than the default set by *Set/File*.

Linecolour (Colour)

This parameter changes the colour for drawing the axes and surrounding boxes, see *Alter()* for details.

Linetype (LineStyle)

This parameter changes the line style for drawing the axes and surrounding boxes, see *Alter()* for details.

Linewidth (Real)

This parameter changes the line width used for drawing line plots, see *Alter()* for details.

Textcolour (colour)

This parameter changes the colour used for drawing text, see *Alter()* for details.

Textheight (Real)

This parameter changes the text height used for drawing text, see *Alter()* for details.

Peak()

Performs interactive peak fitting

PEAK [**w**=*Workspace*] Fit peaks in a workspace

example:

```
# Fit peaks from a genie intermediate file
>> fit = peak( get(3,"mydata.in3") )
>> printn fit
```

Note: See the Peakfit() and Peakgen() commands for more detail.

Peak

The peak command is a high level command built on the powerful Open GENIE peak fitting primitive commands. It operates interactively to allow selection of the bounds of the peak and the form of fit to use. Once fitted the peak parameters of the fit are displayed and made available as a result so that they can be used later. For more control over the fitting process see the *Peakgen()* and *Peakfit()* primitive commands.

Parameters:

W (Workspace)

Any valid single spectrum in a workspace.

RESULT = (Workspace)

The result of the Peak() command is a workspace containing the fitting parameters used as well as a description of the type of fit and an algebraic description of the function fitted.

Plot()

Overplots an existing graph produced by Display()

PLOT	[**wksp**=*Workspace*] [**binning**=*Integer*]	Overplots with new data
[/HISTOGRAM]		Plots as a histogram (default)
[/LINE]		Plots as a line
[/MARKERS]		Plots as markers
[/ERRORS]		Plots as error bars

example:

```
# Plot a vanadium spectrum on top of
# the data as a comparison
>> Display s(5) xmin=20000.0
>> Plot/Line get("vanadium1","normal.in3")
```

Note: Plot does not allow any change of scaling.

Plot

The Plot() command is really an adjunct to the *Display()* command. It is used to plot just the data plot of a spectrum such that it can be used to superimpose new data onto existing plots made with the *Display()* command. The Plot() command itself has no ability to create axes or scale the data.

Parameters:

/Histogram

This is the default as for the *Display()* command when given bin-mode (histogrammed) data. It will plot a spectrum making the assumption that it is a valid histogram (i.e. binned data with one more X value than Y value). The data values are represented by the height of the corresponding bin. Remember that several commands can actually affect the height of bins unexpectedly by changing the binning method and/or data. Examples of these are *Alter/Binning* and *Units/Channel*.

If point-mode data is supplied with the /Histogram qualifier an error will be reported as it is not possible for Open GENIE to guess the assumptions you may want to make about the conversion (an extra X value would have to be generated by interpolation).

/Line

This is the default for the Plot() command when given point-mode data. It will plot a spectrum as if it is a numerically calculated function (i.e. with the same number of X-values as Y-values) and join the points with a straight lines. If histogram data is displayed with the /Line qualifier, the data will undergo a transformation to take the bin centre positions as the X-values for the data points (see the *Centre_bins()* function). There are situations where you may not wish this behaviour to be the default (e.g. for logarithmic binning). Generally, it is safer to transform the data yourself into point-mode and then use Display to plot it.

/Markers

This is identical to the /Line qualifier in operation except that instead of joining the data points with lines, marker symbols are used. See *Marker Types* for a list of available styles.

/Errors

This is identical to the /Line qualifier in operation except that instead of joining the data points with lines, error bars are generated. These are centered on the data points and have a size scaled from the values of the data error values contained within the workspace E array.

Wksp (Workspace)

The workspace containing the data to overplot with.

Binning (Integer)

Binning for this plot only. See *Alter/Binning* for more details on binning. Note that by binning differently with the Plot() command, data may appear quite differently. This is because the implicit micro-averaging in the binning may substantially reduce the height of sharp peaks or noise.

Toggle()

Toggles between common settings, mainly graphics.

TOGGLE/STATUS	Reports the values of all Toggled settings.
TOGGLE/LOGX	X axis log/linear
TOGGLE/LOGY	Y axis log/linear
TOGGLE/CLEAR	Screen clearing between plots
TOGGLE/GRATICULE	Toggle graticule off/on
TOGGLE/HEADER	Header plotted on Display() command.
TOGGLE/RX	X axis rounding off/on
TOGGLE/RY	Y axis rounding off/on
TOGGLE/INFO	Informational message printing on/off
[/ON]	Toggles item on
[/OFF]	Toggles item off

example:

```
# Switch off informational messages and display
# a data plot with no banner heading.
>> Toggle/info/off
>> Toggle/header
>> Display Spectrum(22)
```

Note: See the Alter() command for changing other graphics settings.

Toggle/Status

The Toggle command acts like an on/off switch for the settings described below. All the toggle commands switch the setting to its opposite value unless /Off or /On is specified in which case the value of the setting is forced on or off.

Parameters:

/On

Applies to all Toggle() commands and forces the value of the parameter to the On state.

/Off

Applies to all Toggle() commands and forces the value of the parameter to the Off state.

Toggle/LogX

Set the default mode for plotting with the *Display()* and *Plot()* commands to plot logs of the X data against a log X Axis. On startup, the X-Axis mode is linear.

Toggle/LogY

Set the default mode for plotting with the *Display()* and *Plot()* commands to plot logs of the Y data against a log Y Axis. On startup, the Y-Axis mode is linear.

Toggle/Clear

Alters the behaviour of the *Display()* and *Multiplot()* commands to produce plots *without* clearing the display device first. This can be used with the *Alter/Plot* command to put multiple pictures on one display, see also the Graphics primitive window creation commands. On startup, both the commands clear the display device by default.

Toggle/Graticule

Switches a graticule or crosshatch on for subsequent invocations of the *Display()* command. The frequency of the graticule may be altered by the *Alter()* command.

Toggle/Header

Switches the default for whether to display the standard *Display()* command banner or not. The banner can be toggled off to make more room to display the plot on a small screen. On startup the banner is enabled.

Toggle/RoundX

Enables rounding on the X axis. By default this is switched on, but for accuracy it may be desirable to let the absolute values of the axis be fixed to the exact limits of the plot.

Toggle/RoundY

Enables rounding on the Y axis. By default this is switched on, but for accuracy it may be desirable to let the absolute values of the axis be fixed to the exact limits of the plot.

Toggle/Info

Switches the display of informational messages on/off. Normally Open GENIE operations report warning or informational messages as ANSI style **blue** coloured text using the *Printin()* command. For example, Genie command language procedures using a lot of data I/O may wish to suppress messages about the data being read.

Zoom()

Interactively choose an area of a plot to magnify.

ZOOM	Redraw a plot scaling to the cursor selected box.
[/ERRORS]	Redraw using error bars
[/HISTOGRAM]	Redraw as a histogram
[/LINE]	Redraw as a line plot
[/MARKERS]	Redraw as a marker plot

example:

```
# Zoom in on the current plot
# and re-display with error bars
>> zoom/errors
```

Note: A zoom can be undone by a Limits/Default command with no arguments

Zoom

The Zoom() command is an interactive command which helps examine a plot already on the screen in more detail. After running the zoom command two points are selected with the cursor to mark the lower and upper X and Y boundaries of the area to be zoomed. The zoom command works by re-displaying the selected region of data. As a result of this approach, the resolution of a zoom is only limited to the resolution set by the data or the current bin setting (see *Alter/Binning*).

Parameters:

/Errors

Display the resulting plot in the form of error bars.

/Histogram

Display the resulting plot in the form of a histogram (the default for bin-mode data).

/Line

Display the resulting plot in the form of a line plot (the default for point-mode data).

/Markers

Display the resulting plot in the form of marker symbols

Primitive Graphics Command Reference

Delete()

Deletes unwanted graphics objects from memory

DELETE **item**=*Gobject* Deletes an object, window or a picture.

example:

```
# Delete object 10 in window 2, picture 4.
>> delete obj(4,2,10)
# Create and delete a picture
>> mypict = picture()
>> delete item=mypict
```

Note: Once something is deleted then there is no going back.

Delete

Deletes a DrawableObject from the window it is found in. There is no default object chosen by Delete(), as the operation is permanent, an object must be specified. Delete() will change the numbering order of pictures, windows and objects.

Parameters:

Item (Gobject)

Deletes the specified object. If the object refers to a window or picture, all contained objects are deleted too.

Pic_add()

Copy graphical objects from picture to picture.

PIC_ADD	**item**=*Gobject* [**device**=*Gobject*]	Copy item to current picture/window
PIC_ADD	**item**=*Gobject* [**dest**=*Gobject*] [**device**=*Gobject*]	Copy item to picture or window

example:

```
# Draw a line object, then copy it into
# the last open window on picture 4
# redraw the updated picture on device 3.
>> my_line = Line:Draw(0.0,1.0,0.0,1.0,$green)
>> Pic_add item=my_line dest=pic(4) device=dev(3)
```

Note: item can only ever be a primitive object or a window.

Pic_add()

This command allows graphical objects to be re-show more than once, moved from display to display or duplicated. Open GENIE contains a copy of every graphical object that has been created and is still in use (they can be removed permanently by the *Delete()* command).

Objects can either be referred to using the *Pic()*, *Win()*, *Dev()* or *Obj()* functions or obtained directly as an object reference when an object is created (as in the example above).

Note that the default destination for Pic_add() is the current Picture when "item" is a window, and is the Current Window when "item" is a primitive object.

Parameters:

Item (Gobject)

This must be an object reference to either a drawing primitive (e.g. a line) or a window. This parameter cannot be a device or picture reference. This is the item which is to be copied and redrawn at a new destination. Probably the most useful form of the command is to copy windows which already have several things drawn in them.

Dest (Gobject) [default = current window]

Specifies a destination picture or window. If a picture is specified and the "item" parameter is a primitive object the window used will be the last one in the picture. When the "item" parameter is a window, the window is simply added at the end of the specified picture.

Device (Gobject) [default = current device]

After adding an object to a picture, the object will be re-drawn, by default this will be on the current device. If this is not the device required, the device in which to re-draw the newly copied primitive can be specified explicitly.

RESULT = (Restype)

Returns a reference to the copied item. This allows "/alter" commands to be applied to the object without having to find it again.

Select()

Select a new current device, window or picture

SELECT **item**=*Gobject* Sets this to be the current object

example:

```
# Select an old window to draw in
>> Select win(picnum=3,winnum=4)
>> keep_dev = Select(dev(3))
>> draw/text 0.2 0.3 "Fudge this data"
# back to previous device & current window
>> select keep_dev
>> select win(0,0)
```

Note: Drawing is always to the current window.

Select

The Open GENIE graphics system always maintains a record of a current device, picture and window. This is done so that drawing commands can all occur without having to specify where to draw. Normally Open GENIE looks after the setting of these defaults in an intuitive way but the Select() command makes it possible to change the currently selected objects.

The most likely use for the Select() command is when programming a complex system involving several windows, pictures and open devices. It is also the only way to go back to draw in a window which was created earlier on and which has been superceded by subsequent window creating commands. To specify which objects to select, see the *Dev()*, *Pic()* and *Win()* commands.

Parameters:

Item (Gobject)

This specifies either a picture, window or device object and the object specified then becomes the current device, picture or window. Generally if you wish to draw into a previous picture or window, the object to select is the window; drawing commands can only be into a window. The only time to select a picture explicitly is when adding a new window to an old picture.

RESULT = (Gobject or Undefined)

Returns the Gobject selected as the new current device, window or picture.

Redraw()

Redisplay current or previous pictures or objects.

REDRAW	[**item**=*Gobject*] [**device=***Gobject***]**	Redraws all or part of the specified picture

example:

```
# Store an HPGL copy of the last but one picture
>> device/close
>> device/open "HPGL" file="OUT.HPGL"
>> redraw pic(-2)
```

Note: The default is to redraw the current picture on the current device

Redraw

This command allows a complete redraw of a picture (either one currently displayed on the device or one which was previously displayed and still remains in the picture list). As pictures are stored independently of the device on which they are displayed, it is possible to close the currently open device, open another, and then redraw one or more pictures to the new device. This is how the high level command *Hardcopy()* works. Windows and objects may also be redrawn specifically where the underlying graphics driver allows updates to specific parts of the display.

Parameters:

Item (Gobject) [default = current picture]

The graphics object to be redrawn, this will default to the current picture and effectively refresh the whole picture on the current device. If a window or even an individual primitive object is specified instead, just the window or object will be redrawn.

Device (Gobject) [default = current device]

The device on which to redraw the item specified. The default is the currently selected device.

Undraw()

Undraw a primitive graphics operation.

UNDRAW [**item**=*Gobject*] Temporarily remove a graphics primitive

example:

```
>> Draw/Text 0.5 0.4 "Special data point"
# oops, too far to the right!
>> Undraw
>> Draw/Text 0.4 0.4 "Special data point"
```

Note: See Obj(), Win() and Pic() commands for specifying objects.

Undraw

Undraw allows a temporary removal of a graphics item; either a primitive object, window or picture. Normally it is used interactively when annotating a plot as a means of undoing some annotation which was inappropriate. An undrawn item remains in the display list and can be redrawn if desired with the *Redraw()* command. If something is undrawn by mistake, it can be redrawn using the *Redraw()* command.

Parameters:

item (Gobject) [default = last graphical object drawn]

This parameter specifies the the primitive to remove. Objects can be referred to either by an object reference saved when the object was created or by one of the object locating functions *Pic()*, *Win()*, *Obj()*.

Device()

Selects and controls graphics devices

DEVICE/OPEN	[**devtype**=*String*] [**width**=*Real*] [**height**=*Real*]	Opens a graphics device for plotting on
DEVICE/CLEAR		Clears the current graphics device
DEVICE/CLOSE		Close the current graphics device
DEVICE:STATUS()		Returns $TRUE if there is a device open

example:

```
# Open two X-Windows display devices
# and close one.
>> device/open "XW" height=5.0
>> dev2 = device:open("XWINDOW",height=3.0)
>> select dev2
>> device/close
```

Note: For windowing systems, multiple device may be open on the screen at once.

Device/Open

The supported interactive devices on most machines include "XWINDOWS" X-Windows and "TEK4010" Tek 4010 compatible. The supported non-interactive devices include "PS" Postscript, "CPS" Colour Postscript and "HPGL" HPGL driver for 7475A For full details of supported devices on each machine see *Supported Graphics Devices*.

Parameters:

Devtype (String) [default = "XWINDOW"]

The device name string, this can be any supported device, interactive or hardcopy.

Width (Real) [default = 8.0 inches]

Width of the graphics display in inches.

Height (Real) [default = 8.0 inches]

Height of the graphics display in inches.

RESULT = (Device)

Returns the device object for later reference.

Device/Clear

Clears the display surface on the currently open device or advances to a new sheet of paper, a new picture is created automatically and will become the current picture. The older pictures will still be available by using the *Redraw()* command (Redraw pic(-1) will redraw the picture before the device was cleared).

Device/Close

Deactivates the currently open graphics device. This command flushes output to a file when using a hardcopy device and must be called before the file can be used. Once the device is closed, it is not possible to draw anything without re-opening the device. A different device can be used when the device is re-opened.

Parameters:

Object (Gobject)

Optional device to close, otherwise the default is to close the current device.

Device:Status()

Returns true if there is at least one graphics device open. This can be useful where another procedure may or may not have initialized the graphics system previously and it would be unnecessary or unhelpful to open a second device.

Picture()

Creates a picture

PICTURE	Creates and displays a picture on the graphics device

example:

```
# Opening a device then
# creating a picture
>> device/open
>> picture
```

Note: To create a pictures without opening a real device, open a "/NULL" device.

Picture

Create and display a picture on the current device. After a picture command the current picture will be reset to the new picture and the previous picture can be accessed via the *Pic()* command (i.e. Redraw pic(-1)). After creating a picture you will need to create windows in which to draw objects.**Parameters:**

RESULT = (Picture)

This returns all of the properties of a Picture as a picture object reference.

Window()

Obsolete command

WINDOW/AUTOSCALED	Autoscaled window, use WIN_AUTOSCALED
WINDOW/MULTIPLOT	multiplot window, use WIN_MULTIPLOT
WINDOW/SCALED	Scaled window, use WIN_SCALED
WINDOW/TWOD	2-D window, use WIN_TWOD
WINDOW/UNSCALED	Unscaled window, use WIN_UNSCALED

Window()

This command is now obsolete. The new "Win_" commands provide the same functionality.

Win_autoscaled()

Creates a scaled window automatically from supplied data.

WIN_AUTOSCALED	**dxmin**=*Real* **dxmax**=*Real* **dymin**=*Real* **dymax**=*Real* **x**=*RealArray* **y**=*RealArray* [**e**=*RealArray*] [**colour**=*Colour*] [**minmode**=*Integer*] [**maxmode**=*Integer*]	Creates an autoscaled window.

example:

```
# Creating and displaying an autoscaled
# window, where yarray, xarray and earray
# have already been defined
>> win_autoscaled 0.1 0.9 0.1 0.9 xarray yarray earray
```

Note: d min/max values give fraction of device window to cover

---

Win_Autoscaled

Creates and displays a scaled window, without the knowledge of the exact co-ordinates needed for the scaling of the window. The scaling is fairly intelligent and can be controlled by the "maxmode" and "minmode" parameters. Obviously for this routine to work, the data must be available already, if not the *Win_scaled()* command can be used.

Parameters:

Dxmin (Real)

Sets the minimum x device co-ordinate.

Dxmax (Real)

Sets the maximum x device co-ordinate.

Dymin (Real)

Sets the minimum y device co-ordinate.

Dymax (Real)

Sets the maximum y device co-ordinate.

X (RealArray)

Array to use for X axis limits.

Y (RealArray)

Array to use for Y axis limits.

E (RealArray)

Optional array to use for Y axis limits.

Minmode (Integer)

Specify how the axes are scaled by setting the minimum mode where, 0=absolute values, 1 or 3 = +10%, 2 or 3 = round down, and 4 = force zero.

Maxmode (Integer)

Specify how the axes are scaled by setting the maximum mode where, 0=absolute values, 1 or 3 = +10%, 2 or 3 = round down, and 4 = force zero.

Colour (Colour)

Sets the colour of the window. By default the colour of the window is the background colour.

RESULT = (WindowScaled)

Returns all of the properties of the newly created window.

Win_multiplot()

Controls and creates multi_plot windows for plotting

WIN_MULTIPLOT	**dxmin**=*Real* **dxmax**=*Real* **dymin**=*Real* **dymax**=*Real* **object**=*Gobject* [**minmode**=*Integer*] [**maxmode**=*Integer*] [**colour**=*Colour*]	Creates a multiplot window.

example:

```
# Creating and displaying a multiplot
# window
>> win_multiplot 0.1 0.9 0.1 0.9
```

Note: d means the device co-ordinate.

Win_MultiPlot

Creates and displays a window which enables multiplots to be drawn. This window can not be created without a completed multi_plot object from which to scale the axes.

Parameters:

Dxmin (Real)

Sets the minimum x device co-ordinate.

Dxmax (Real)

Sets the maximum x device co-ordinate.

Dymin (Real)

Sets the minimum y device co-ordinate.

Dymax (Real)

Sets the maximum y device co-ordinate.

Object (Gobject)

An already created multi_plot object with spectra added so that this window can scale from it.

Minmode (Integer)

Specify how the axes are scaled by setting the minimum mode where, 0=absolute values, 1 or 3 = +10%, 2 or 3 = round down, and 4 = force zero.

Maxmode (Integer)

Specify how the axes are scaled by setting the maximum mode where, 0=absolute values, 1 or 3 = +10%, 2 or 3 = round down, and 4 = force zero.

Colour (Colour) [= $background]

Sets the colour of the window.

RESULT = (MultiplotWindow)

Returns all of the properties of the newly created window.

Win_scaled()

Controls and creates pre-scaled windows for plotting data.

WIN_SCALED	**dxmin**=*Real* **dxmax**=*Real* **dymin**=*Real* **dymax**=*Real* **wxmin**=*Real* **wxmax**=*Real* **wymin**=*Real* **wymax**=*Real* [**colour**=*Colour*] [**minmode**=*Integer*] [**maxmode**=*Integer*]	Creates a scaled window.

example:

```
# Creating and displaying a scaled
# window
>> win_scaled 0.1 0.9 0.1 0.9 -100.0 10.0 0.0 10.0
>>
```

Note: d means the device co-ordinate, w means the world co-ordinate.

Win_Scaled

Creates and displays a scaled window. A scaled window is an area in which it is possible to specify any range of data or window co-ordinates to map onto an area of the display device (specified in device co-ordinates). Normally a scaled window should be created for every data plot. For drawing simple graphics which do not require an extra scaling to match the data you can use the *Win_unscaled()* command instead.

Parameters:

Dxmin (Real)

Sets the minimum x device co-ordinate.

Dxmax (Real)

Sets the maximum x device co-ordinate.

Dymin (Real)

Sets the minimum y device co-ordinate.

Dymax (Real)

Sets the maximum y device co-ordinate.

Wxmin (Real)

Sets the minimum x world co-ordinate.

Wxmax (Real)

Sets the maximum x world co-ordinate.

Wymin (Real)

Sets the minimum y world co-ordinate.

Wymax (Real)

Sets the maximum y world co-ordinate.

Colour (Colour) [= $background]

Sets the colour of the window.

Minmode (Integer)

Specify how the axes are scaled by setting the minimum mode where, 0=absolute values, 1 or 3 = +10%, 2 or 3 = round down, and 4 = force zero.

Maxmode (Integer)

Specify how the axes are scaled by setting the maximum mode where, 0=absolute values, 1 or 3 = +10%, 2 or 3 = round down, and 4 = force zero.

RESULT = (WindowScaled)

Returns all of the properties of the newly created window.

Win_twod()

Controls and creates two dimensional windows for plotting

WIN_TWOD	**dxmin**=*Real* **dxmax**=*Real* **dymin**=*Real* **dymax**=*Real* ***x***=*RealArray* **y**=*RealArray* [**colour**=*Colour*]	Creates a Two dimensional window.

example:

```
# Creating and displaying a Twod window,
# where yarray and xarray have already
# been defined
>> win_twod 0.1 0.9 0.1 0.9 y=yarray x=xarray
```

Note: d means the device co-ordinate. Giving the x/yarray the window is scaled automatically.

Win_Twod

Creates and displays a Two dimensional window ready for plotting a colour cell or contour plot. The area of the screen to use for the window is specified with the four device co-ordinates and the supplied X and Y arrays are used to auto-scale the axes in the X and Y directions respectively. If the X and Y scaling is known the *Win_scaled()* command my be used instead.

Parameters:

Dxmin (Real)

Sets the minimum x device co-ordinate.

Dxmax (Real)

Sets the maximum x device co-ordinate.

Dymin (Real)

Sets the minimum y device co-ordinate.

Dymax (Real)

Sets the maximum y device co-ordinate.

X (RealArray)

Scales the two dimensional window.

Y (RealArray)

Scales the two dimensional window.

Colour (Colour) [= $background]

Sets the colour of the window

RESULT = (TwoDWindow)

Returns all of the properties of the newly created window.

Win_unscaled()

Controls and creates unscaled windows for plotting.

WIN_UNSCALED	**xmin**=*Real* **xmax**=*Real* **ymin**=*Real* **ymax**=*Real* [**colour**=*Colour*]	Sets the device co-ordinates and creates an unscaled window.

example:

```
# Creating and displaying an
# unscaled window
>> win_unscaled 0.1 0.9 0.1 0.9
```

Note: d means the device co-ordinate.

Win_Unscaled

Creates and displays an unscaled window. The unscaled window is the simplest form of graphics window. By default every new picture has an unscaled window associated with it and they can be created as desired. Normally things such as plot annotation, text and boxes are drawn in the default unscaled window to avoid scaling the simple graphics primitives in the same coordinates as the plot. The co-ordinates within an unscaled window always go from 0 to 1 in the X and Y directions.

Parameters:

Xmin (Real)

Sets the minimum x device co-ordinate.

Xmax (Real)

Sets the maximum x device co-ordinate.

Ymin (Real)

Sets the minimum y device co-ordinate.

Ymax (Real)

Sets the maximum y device co-ordinate.

Colour (Colour) [= $background]

Sets the colour of the window

RESULT = (Window)

Returns all of the properties of the newly created window.

New_zoom()

Zooms in on a graphics device

NEW_ZOOM	Magnifies specified areas of the current picture

example:

```
# Using the zoom command
new_zoom
```

Note: This is only partially implemented.

New_Zoom

The left mouse button specifies the region to be zoomed.
The Right mouse button will:

- go back to the original picture (clicked once and device is showing zoomed area)
- exits the command (clicked twice - if originally showing
a zoomed area, otherwise click once)

Getcursor()

Cursor input routine for a window or a picture.

GETCURSOR()	[**x**=*Real*] [**y**=*Real*]	Returns co-ordinates of the point the mouse was clicked

example:

```
# Shows how to use the getcursor command
>> printn getcursor(0.5, 0.5)
```

Note: If there is more than one window where the pointer clicked, it will return the last window drawn.

Getcursor()

Returns the point where the mouse clicked in device co-ordinates (d_x, d_y), window co-ordinates (w_x, w_y), and a character to represent the mouse button clicked ("A" for left "X" for right, "D" for middle).

X, Y (Real) [default = (0, 0)]

Specifies an optional starting place to put the cross hairs on the screen. This is specified in device co-ordinates.

= (Workspace)

At the point the mouse clicked, it returns all of the possible co-ordinates and window number in a single workspace

RESULT = (Workspace)

At the point the mouse clicked, it returns all of the possible co-ordinates and window number in a single workspace.

Draw()

Obsolete command

DRAW/AXES	Displays axes on the graphics device, use AXES
DRAW/ERRORS	Displays error-bars on a plot, use ERRORS
DRAW/PLOT	Plots a Graph on the graphics device, use GRAPH
DRAW/GRATICULE	Displays a Graticule on the graphics device, use GRATICULE
DRAW/HISTOGRAM	Plots a histogram on the graphics device, use HISTOGRAM
DRAW/LABELS	Displays x/y labels on the graphics device, use LABELS
DRAW/LINE	Displays a line on the graphics device, use LINE
DRAW/MARKERS	Displays markers on the graphics device, use MARKERS
DRAW/POLYGON	Displays a polygon on the graphics device, use POLYGON
DRAW/TEXT	Displays some text on the graphics device, use TEXT
DRAW/TITLE	Displays a Title on the graphics device, use TITLE

Draw()

This command is now obsolete. The alternatives listed above provide the same functionality.

Axes()

Displays an axes on the graphics device

AXES/DRAW [/XLOG] [/YLOG] [/YHORIZ] [/NOXNUM] [/NOYNUM] [/XLINEAR] [/YLINEAR] [/YVERTICAL] [/XNUM] [/YNUM]	[**colour**=*Colour*] [**size**=*Real*] [**font**=*Font*] [**line_thickness**=*Real*] [**line_type**=*LineStyle*]	Draws an Axis on a graphics device
AXES/ALTER [/XLOG] [/YLOG] [/YHORIZ] [/NOXNUM] [/NOYNUM] [/XLINEAR] [/YLINEAR] [/YVERTICAL] [/XNUM] [/YNUM]	[**colour**=*Colour*] [**size**=*Real*] [**font**=*Font*] [**line_thickness**=*Real*] [**line_type**=*LineStyle*] **object**=*Gobject*	Alters an axis already drawn on a graphics device

example:

```
# Draw green Axes on a device with a window already
# displayed, then alters the colour.
>> a = axes:draw(colour=$green)
>> axes/alter colour=$red object=a
```

Note: By default the axes are linear, x is labeled horizontally and y is labeled vertically

---

Axes/Draw

Draws an axes on a graphics device when a window is displayed previously.

Parameters:

/Xlog

Draws a logarithmic x axis.

/Ylog

Draws a logarithmic y axis.

/Yhoriz

Labels the numbers horizontally on the y axis..

/Noxnum

Does not put numbers on the x axis.

/Noynum

Does not put numbers on the y axis.

/Xlinear

Draws a linear x axis.

/Ylinear

Draws a linear y axis.

/Yvertical

Draws the numbers vertically on the y axis.

/Xnum

Draws numbers on the x axis.

/Ynum

Draws numbers on the y axis.

Colour (Colour)

Defines the colour of the axis.

Size (Real)

Defines the size of the numbers.

Font (Font)

Defines the font, $Normal, $Roman, $Italic, $Script.

Line_thickness (Real)

Defines the thickness of the axis.

Line_type (LineStyle)

Defines the line_type of the axes, $Full, $Dash, $Dot_Dash, $Dot.

RESULT = (OneDAxes)

This returns all of the properties of the newly created axes.

Axes/Alter

Alters a specified axes, the parameter object MUST be specified.

Parameters:

/Xlog

Draws a logarithmic x axis.

/Ylog

Draws a logarithmic y axis.

/Yhoriz

Labels the numbers horizontally on the y axis..

/Noxnum

Does not put numbers on the x axis.

/Noynum

Does not put numbers on the y axis.

/Xlinear

Draws a linear x axis.

/Ylinear

Draws a linear y axis.

/Yvertical

Draws the numbers vertically on the y axis.

/Xnum

Draws numbers on the x axis.

/Ynum

Draws numbers on the y axis.

Colour (Colour)

Defines the colour of the axis.

Size (Real)

Defines the size of the numbers.

Font (Font)

Defines the font, $Normal, $Roman, $Italic, $Script.

Line_thickness (Real)

Defines the thickness of the axis.

Line_type (LineStyle)

Defines the line_type of the axes, $Full, $Dash, $Dot-Dash, $Dot.

Object (Gobject)

Shows which axes is to be altered.

RESULT = (OneDAxes)

This returns all of the properties of the altered axes

Errors()

Displays error-bars on a plot

ERRORS/DRAW [/HORIZONTAL] [/VERTICAL]	**xarray**=*RealArray* **yarray**=*RealArray* **earray**=*RealArray* [**colour**=*Colour*] [**line_thickness**=*Real*]	Draw error-bars at the x/yarray points, with the size of the earray values
ERRORS/ALTER [/HORIZONTAL] [/VERTICAL]	[**xarray**=*RealArray*] [**yarray**=*RealArray*] [**earray**=*RealArray*] [**colour**=*Colour*] [**line_thickness**=*Real*] **object**=*Gobject*	Alter the error-bars

example:

```
# Add error bars to a plot
>> e = Errors:draw(colour=$blue)
>> Errors/alter colour=$red object=e
```

Note: e array can only default like if it has been given originally.

Errors/Draw

Draws error-bars on a graphics device. If the xarray and yarray have already been defined, by either graph, histogram or markers, then it is not necessary to define them again.

Parameters:

/Vertical

Draw vertical error-bars.

/Horizontal

Draw horizontal error-bars.

Xarray (RealArray)

Defines the x points.

Yarray (RealArray)

Defines the y points

Earray (RealArray)

Defines the size of the error-bars.

Colour (Colour)

Defines the error-bars colour.

Line_thickness (Real)

Defines the error-bars width.

RESULT = (OneDVErrors or OneDHErrors)

Returns the properties of the newly created error-bars.

Errors/Alter

Alters an already created set of error bars.

Parameters:

/Vertical

Draw vertical error-bars.

/Horizontal

Draw horizontal error-bars.

Xarray (RealArray)

Not yet implemented.

Yarray (RealArray)

Not Yet Implemented.

Earray (RealArray)

Not Yet Implemented.

Colour (Colour)

Defines the error-bars colour.

Line_thickness (Real)

Defines the error-bars width.

Object (Integer)

Shows which set of error-bars are to be altered, in any one window.

RESULT = (OneDVErrors or OneDHErrors)

Returns the properties of the altered error-bars.

Graph()

Displays a graph (or line plot) on a graphics device.

GRAPH/DRAW	**xarray**=*RealArray* **yarray**=*RealArray* [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Draw a graph at the x/yarray points.
GRAPH/ALTER	[**xarray**=*RealArray*] [**yarray**=*RealArray*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Gobject*	Alter the graph

example:

```
# Add graph to a plot
>> g = Graph:draw(colour=$blue)
>> Graph/alter colour=$red object=g
```

Note: For graph/alter yarray and xarray are not yet implemented.

Graph/Draw

Draws a graph on a graphics device. If the xarray and yarray have already been defined, by either errors, histogram or markers, then it is not necessary to define them again.

Parameters:

Xarray (RealArray)

Defines the x points.

Yarray (RealArray)

Defines the y points

Colour (Colour)

Defines the graphs colour.

Line_type (LineStyle)

Defines the line type of the graph. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Defines the graphs line width.

RESULT = (OneDGraph)

Returns the properties of the newly created graph.

Graph/Alter

Alters a graph. The xarray and yarray are not able to be altered at this point in time.

Parameters:

Xarray (RealArray)

Not yet implemented.

Yarray (RealArray)

Not Yet Implemented.

Colour (Colour)

Defines the graph colour.

Line_type (LineStyle)

Defines the line type of the graph. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Defines the graphs line width.

Object (Gobject)

Shows which graph is to be altered.

RESULT = (OneDGraph)

Returns the properties of the altered graph.

Graticule()

Displays a graticule axis on the graphics device

GRATICULE/DRAW [/XLOG] [/YLOG] [/YHORIZ] [/NOXNUM] [/NOYNUM] [/XLINEAR] [/YLINEAR] [/YVERTICAL] [/XNUM] [/NUM]	[**colour**=*Colour*] [**size**=*Real*] [**font**=*Font*] [**line_thickness**=*Real*] [**line_type**=*LineStyle*]	Draws a Graticule on a graphics device
GRATICULE/ALTER [/XLOG] [/YLOG] [/YHORIZ] [/NOXNUM] [/NOYNUM] [/XLINEAR] [/YLINEAR] [/YVERTICAL] [/XNUM] [/NUM]	[**colour**=*Colour*] [**size**=*Real*] [**font**=*Font*] [**line_thickness**=*Real*] [**line_type**=*LineStyle*] **object**=*Gobject*	Alters a graticule already drawn on a graphics device

example:

```
# Draws a green Graticule on a device
# then alters the colour.
>> g = graticule:draw(colour=$green)
>> graticule/alter colour=$red object=g
```

Note: By default the graticule is linear.

Graticule/Draw

Draws a graticule on a graphics device when a window is displayed previously.

Parameters:

/Xlog

Draws with a logarithmic x axis.

/Ylog

Draws with a logarithmic y axis.

/Yhoriz

Labels the numbers horizontally on the y axis..

/Noxnum

Does not put numbers on the x axis.

/Noynum

Does not put numbers on the y axis.

/Xlinear

Draws a linear x axis.

/Ylinear

Draws a linear y axis.

/Yvertical

Draws the numbers vertically on the y axis.

/Xnum

Draws numbers on the x axis.

/Ynum

Draws numbers on the y axis.

Colour (Colour)

Defines the colour of the graticule.

Size (Real)

Defines the size of the numbers.

Font (Font)

Defines the font, $Normal, $Roman, $Italic, $Script.

Line_thickness (Real)

Defines the thickness of the graticule.

Line_type (LineStyle)

Defines the line_type of the graticule, $Full, $Dash, $Dot-Dash, $Dot.

RESULT = (OneDGraticule)

This returns all of the properties of the newly created graticule.

Graticule/Alter

Alters a specified graticule, the parameter object MUST be specified.

Parameters:

/Xlog

Draws a logarithmic x axis.

/Ylog

Draws a logarithmic y axis.

/Yhoriz

Labels the numbers horizontally on the y axis..

/Noxnum

Does not put numbers on the x axis.

/Noynum

Does not put numbers on the y axis.

/Xlinear

Draws a linear x axis.

/Ylinear

Draws a linear y axis.

/Yvertical

Draws the numbers vertically on the y axis.

/Xnum

Draws numbers on the x axis.

/Ynum

Draws numbers on the y axis.

Colour (Integer)

Defines the colour of the graticule.

Size (Real)

Defines the size of the numbers.

Font (Font)

Defines the font, $Normal, $Roman, $Italic, $Script.

Line_thickness (Real)

Defines the thickness of the graticule.

Line_type (LineStyle)

Defines the line_type of the graticule, $Full, $Dash.

Object (Gobject)

Defines which graticule is to be altered, in any one window.

RESULT = (OneDGraticule)

This returns all of the properties of the altered graticule

Histogram()

Displays a histogram on a graphics device.

HISTOGRAM/DRAW [/CENTRE] [/NOTCENTRED]	**xarray**=*RealArray* **yarray**=*RealArray* [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Draw a Histogram at the x/yarray points.
HISTOGRAM/ALTER [/CENTRE] [/NOTCENTRED]	[**xarray**=*RealArray*] [**yarray**=*RealArray*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Gobject*	Alter the Histogram

example:

```
# Draws a histogram
# then alters the colour.
>> h = histogram:draw(colour=$green)
>> histogram/alter colour=$red object=h
```

Note: By default the points are not centred.

Histogram/Draw

Draws a histogram on a graphics device. If the xarray and yarray have already been defined, by histogram, then it is not necessary to define them again.

Parameters:

/Centre

Plots the points at the centre of the bin.

/Notcentred

Plots the points at the beginning of the bin.

Xarray (RealArray)

Defines the x points.

Yarray (RealArray)

Defines the y points.

Colour (Colour)

Defines the histogram's colour.

Line_type (LineStyle)

Defines the line type of the graph. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Defines the histogram's line width.

RESULT = (OneDHistogram)

Returns the properties of the newly created histogram.

Histogram/Alter

Alters a histogram. The xarray and yarray are not able to be altered at this point in time.

Parameters:

/Centre

Plots the points at the centre of the bin.

/Notcentred

Plots the points at the beginning of the bin.

Xarray (RealArray)

Not yet implemented.

Yarray (RealArray)

Not Yet Implemented.

Colour (Colour)

Defines the histogram colour.

Line_type (LineStyle)

Defines the line type of the graph. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Defines the histogram's line width.

Object (Gobject)

Shows which histogram is to be altered, in any one window.

RESULT = (OneDHistogram)

Returns the properties of the altered histogram.

Labels()

Displays x/y labels on the graphics device

LABELS/DRAW [/XHORIZONTAL] [/YVERTICAL] [/XVERTICAL] [/YHORIZONTAL]	[**xlabel**=*String*] [**ylabel**=*String*] [**size**=*Real*] [**colour**=*Colour*] [**font**=*Font*]	Draws x and y labels.
LABELS/ALTER [/XHORIZONTAL] [/YVERTICAL] [/XVERTICAL] [/YHORIZONTAL]	[**xlabel**=*String*] [**ylabel**=*String*] [**size**=*Real*] [**colour**=*Colour*] [**font**=*Font*] **object**=*Gobject*	Alters the x and y labels

example:

```
# Draw some labels
>> a = labels:draw("X Axis", "Y Axis")
# Wanted capitals
>> labels/alter "X AXIS" "Y AXIS" object=a
```

Note: By default xlabel is labeled horizontally, and ylabel is labeled vertically.

Labels/Draw

Draws text just below the x-axis and just to the left of the y-axis.

Parameters:

/Xhorizontal

Sets the xlabels to draw the text horizontally.

/Xvertical

Sets the xlabels to draw the text vertically.

/Yhorizontal

Sets the ylabels to draw the text horizontally.

/Yvertical

Sets the ylabels to draw the text vertically.

Xlabel (String)

Sets the xlabel text.

Ylabel (String)

Sets the ylabel text.

Size (Real)

Sets the size of the text.

Colour (Colour)

Sets the colour of the text.

Font (Font)

Sets the font of the text. Such as, $normal, $roman, $italic, $script

RESULT = (OneDLabels)

Returns all of the properties of the newly created object.

Labels/Alter

Alters any drawableObject with the datatype of OneDLabels.

Parameters:

/Xhorizontal

Sets the xlabels to draw the text horizontally.

/Xvertical

Sets the xlabels to draw the text vertically.

/Yhorizontal

Sets the ylabels to draw the text horizontally.

/Yvertical

Sets the ylabels to draw the text vertically.

Xlabel (String)

Sets the xlabel text.

Ylabel (String)

Sets the ylabel text.

Size (Real)

Sets the size of the text.

Colour (Colour)

Sets the colour of the text.

Font (Font)

Sets the font of the text. Such as, $normal, $roman, $italic, $script

Object (Gobject)

Shows which labels are to be altered, in any one window.

RESULT = (OneDLabels)

Returns all of the properties of the altered object.

Line()

Displays a line on the graphics device

LINE/DRAW	**xstart**=*Real* **ystart**=*Real* **xend**=*Real* **yend**=*Real* [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Draws a line on a graphics device.
LINE/ALTER	[**xstart**=*Real*] [**ystart**=*Real*] [**xend**=*Real*] [**yend**=*Real*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Gobject*	Alters a previously drawn line.

example:

```
#Draw a line
>> aline = line:draw(0.0, 0.0, 1.0, 1.0, line_type=$dash)
# Want the line type to be dot-dash
>> line/alter line_type=$dot-dash object=aline
```

Line/Draw

Draws a line in a graphics device.

Parameters:

Xstart (Real)

x co-ordinate of the starting point.

Xend (Real)

x co-ordinate of the ending point.

Ystart (Real)

y co-ordinate of the starting point.

Yend (Real)

y co-ordinate of the ending point.

Colour (Colour)

Sets the colour of the line

Line_thickness (Real)

Sets the width of the line.

RESULT = (Polyline)

Returns the properties of the newly created line.

Line/Alter

Alters a line in a graphics device.

Parameters:

Xstart (Real)

x co-ordinate of the starting point.

Xend (Real)

x co-ordinate of the ending point.

Ystart (Real)

y co-ordinate of the starting point.

Yend (Real)

y co-ordinate of the ending point.

Colour (Colour)

Sets the colour of the line

Line_thickness (Real)

Sets the width of the line.

Object (Gobject)

Shows which line is to be altered, in any one window.

RESULT = (Polyline)

Returns the properties of the altered line.

Markers()

Displays markers on a plot

MARKERS/DRAW	**xarray**=*RealArray* **yarray**=*RealArray* [**colour**=*Colour*] [**symbol**=*Marker*] [**size**=*Real*]	Draw markers at the x/y array points.
MARKERS/ALTER	[**xarray**=*RealArray*] [**yarray**=*RealArray*] [**colour**=*Colour*] [**symbol**=*Marker*] [**size**=*Real*] **object**=*Gobject*	Alter the markers.

example:

```
# Draws a marker plot with crosses
# then alters the marker symbol.
>> mp = Markers:draw(symbol=$cross)
>> Markers/alter object=mp symbol=$box
```

Note: For markers/alter yarray, xarray are not implemented.

Markers/Draw

Draws markers on a graphics device. If the xarray and yarray have already been defined, by either graph, histogram or markers, then it is not necessary to define them again.

Parameters:

Xarray (RealArray)

Defines the x points.

Yarray (RealArray)

Defines the y points

Colour (Colour)

Defines the markers colour.

Symbol (Marker)

Defines the style of markers. Such as $point, $plus, $star, $circle, $cross, $box.

Size (Real)

Defines the markers size.

RESULT = (OneDMarkers)

Returns the properties of the newly created markers.

Markers/Alter

Alters a set of markers. The xarray and yarray are not able to be altered at this point in time.

Parameters:

Xarray (RealArray)

Not yet implemented.

Yarray (RealArray)

Not Yet Implemented.

Colour (Colour)

Defines the markers colour.

Symbol (Marker)

Defines the style of markers; such as $point, $plus, $star, $circle, $cross, $box.

Size (Real)

Defines the markers size.

Object (Gobject)

Shows which set of markers are to be altered, in any one window.

RESULT = (OneDMarkers)

Returns the properties of the altered markers.

Polygon()

Displays a polygon on the graphics device

POLYGON/DRAW [/FILLED] [/OUTLINE]	**xarray**=*RealArray* **yarray**=*RealArray* [**colour**=*Colour*] [**line_thickness**=*Real*]	Draws a polygon onto a graphics device.
POLYGON/ALTER [/FILLED] [/OUTLINE]	[**xarray**=*RealArray]* [**yarray**=*RealArray*] [**colour**=*Colour*] [**line_thickness**=*Real*] **object**=*Gobject*	Alters a polygon on a graphics device

example:

```
# Draws a polygon plot with crosses
# then alters the marker symbol.
>> Polygon/draw x_points y_points
>> Polygon/alter line_thickness=10.0 object=obj(0,0,0)
```

Note: By default the polygon is outlined.

Polygon/Draw

Draws a polygon onto a graphics device

Parameters:

/Filled

Colours the whole of the polygon.

/Outline

Colours the perimeter of the polygon.

Xarray (RealArray)

Specifies the x co-ordinates of the polygon.

Yarray (RealArray)

Specifies the y co-ordinates of the polygon.

Colour (Colour)

Sets the colour of the polygon.

Line_thickness (Real)

Sets the line thickness of the polygon.

RESULT = (Polygon)

Returns all of the properties of the newly created polygon.

Polygon/Alter

Alters a polygon previously drawn on a graphics device.

Parameters:

/Filled

Colours the whole of the polygon.

/Outline

Colours the perimeter of the polygon.

Xarray (RealArray)

Not Yet Implemented. Specifies the x co-ordinates of the polygon.

Yarray (RealArray)

Not Yet Implemented. Specifies the y co-ordinates of the polygon.

Colour (Colour)

Sets the colour of the polygon.

Line_thickness (Real)

Sets the line thickness of the polygon.

Object (Gobject)

Shows which polygon is to be altered, in any one window.

RESULT = (Polygon)

Returns all of the properties of the altered polygon.

Text()

Displays some text on the graphics device

TEXT/DRAW	**xcoord**=*Real* **ycoord**=*Real* **text**=*String* [**colour**=*Colour*] [**size**=*Real*] [**angle**=*Real*] [**font**=*Font*]	Draws some text onto a graphics device
[/CENTRE]		Centres the text on the given coordinate
TEXT/ALTER	[**xcoord**=*Real*] [**ycoord**=*Real*] [**text**=*String*] [**colour**=*Colour*] [**size**=*Real*] [**angle**=*Real*] [**font**=*Font*] **object**=*Gobject*	Alters some text, on a graphics device.

example:

```
>> text/draw 0.0 0.0 "some text" size=3.0
# The text is a bit to big
>> text/alter size=1.0 colour=$yellow object=obj(0,0,0)
```

Text/Draw

Draws some text onto a graphics device.

Parameters:

/Centre

Draw text so that the point specified is at the centre of the string.

Xcoord (Real)

X co-ordinate of the position of the text.

Ycoord (Real)

Y co-ordinate of the position of the text.

Text (String)

Text to be put onto a graphics device.

Colour (Colour)

Sets the colour of the text.

Size (Real)

Sets the size of the text.

Angle (Real)

Sets the angle for the text to be written.

Font (Font)

Sets the font of the text. Such as, $normal, $roman, $italic, $script

RESULT = (Text)

Returns all of the properties of the newly created text.

Text/Alter

Alters some text already on a graphics device.

Parameters:

Xcoord (Real)

X co-ordinate of the position of the text.

Ycoord (Real)

Y co-ordinate of the position of the text.

Text (String)

Text to be put onto a graphics device.

Colour (Colour)

Sets the colour of the text.

Size (Real)

Sets the size of the text.

Angle (Real)

Sets the angle for the text to be written.

Font (Font)

Sets the font of the text. Such as, $normal, $roman, $italic, $script

Object (Gobject)

Shows which text is to be altered, in any one window.

RESULT = (Text)

Returns all of the properties of the altered text.

Title()

Displays a Title on the graphics device

TITLE/DRAW	**text**=*String* [**colour**=*Colour*] [**size**=*Real*] [**font**=*Font*]	Draw a title onto a graphics device
TITLE/ALTER	[**text**=*String*] [**colour**=*Colour*] [**size**=*Real*] [**font**=*Font*] **object**=*Gobject*	Alters a title on a graphics device

example:

```
>> title/draw "A Title"
# Want the font to be italic
>> title/alter font=$italic
```

Title/Draw

Draws a title onto a graphics device.

Parameters:

Text (String)

Text to be put onto a graphics device.

Colour (Colour)

Sets the colour of the text.

Size (Real)

Sets the size of the text.

Font (Font)

Sets the font of the text. Such as, $normal, $roman, $italic, $script

RESULT = (OneDTitle)

Returns all the properties of the newly created title

Title/Alter

Alters a title on a graphics device.

Parameters:

Text (String)

Text to be put onto a graphics device.

Colour (Colour)

Sets the colour of the text.

Size (Real)

Sets the size of the text.

Font (Font)

Sets the font of the text. Such as, $normal, $roman, $italic, $script

Object (Gobject)

Shows which title is to be altered, in any one window.

RESULT = (OneDTitle)

Returns all the properties of the altered title.

Cell()

Colour cell plot for gridded data.

CELL/DRAW [/SMOOTH] [/LOG] [/SQRT]	**values**=*RealArray* **table**=*ColourTable* [**tr**=*RealArray*] [**valmax**=*Real*] [**valmin**=*Real*]	Plots a cell on a graphics device
CELL/ALTER [/SMOOTH] [/LOG] [/SQRT]	**values**=*RealArray* **table**=*ColourTable* [**tr**=*RealArray*] [**valmax**=*Real*] [**valmin**=*Real*] **object**=*Gobject*	Alters a cell already drawn on a graphics device

example:

```
# Produce a cell plot of 2-D data
>> w = get(1:10)
>> yindex=dimensions(10)
>> fill yindex 1.0 1.0
>> win_twod 0.1 0.9 0.1 0.9 x=w.x y=yindex
>> t = colourtable:rainvow(80)
>> Cell/Draw values=w.y table=t
```

Cell/Draw

Draws a colour cell plot. The cell plot will transform with a transformation matrix if it is given or it will assume that the data is linear and scale to fit the axes. For non-linear grids or non-gridded data see *Cell_array()* and *Cell_function()* respectively.

Parameters:

/Smooth

Interpolate colour across each cell.

/Log

Use a logarithmic scale for cell plot colours.

/Sqrt

Use a square root scale for cell plot colours.

Values (RealArray)

Defines the cell values of the cell plot. Values is a two dimensional array with sufficient data to fit the grid points.

Table (Colourtable)

Defines the colour table for the cell plot.

Tr (RealArray)

Array defining a transformation between the i, j grid of the data array and the window coordinates. The window coordinates of the array point a[i,j] are given by:

x = tr[1] + tr[2] * i + tr[3] * j
y = tr[4] + tr[5] * i + tr[6] * j

Usually tr[3] and tr[5] are zero - unless the coordinate transformation involves a rotation or shear.

The default transformation if this parameter is not specified is to map linearly onto the window to fit the plot.

Valmax (Real)

Cut off maximum above which the colour table will not be mapped.

Valmin (Real)

Cut off minimum below which the colour table will not be mapped.

RESULT = (Cell)

This returns all of the properties of the Cell plot.

Cell/Alter

Alters an already drawn cell plot.

Parameters:

/Smooth

Interpolate colour across each cell.

/Log

Draws a logarithmic cell plot.

/Sqrt

Draws a square root cell plot.

Values (RealArray)

Defines the cell values of the cell plot, values is a two dimensional array with sufficient data to fit the grid points.

Table (Colourtable)

Defines the colour table for the cell plot.

Tr (RealArray)

Array defining a transformation between the i, j grid of the data array and the window coordinates. The window coordinates of the array point a[i,j] are given by:

x = tr[1] + tr[2] * i + tr[3] * j
y = tr[4] + tr[5] * i + tr[6] * j

Usually tr[3] and tr[5] are zero - unless the coordinate transformation involves a rotation or shear.

The default transformation if this parameter is not specified is to map linearly onto the window to fit the plot.

Valmax (Real)

Cut off maximum above which the colour table will not be mapped.

Valmin (Real)

Cut off minimum below which the colour table will not be mapped.

Object (Gobject)

The cell plot to alter.

RESULT = (Cell)

This returns all of the properties of the altered Cell plot.

Cell_array()

Colour cell plot for gridded data with explicit gridding.

CELL_ARRAY/DRAW [/SMOOTH] [/LOG] [/SQRT]	**values**=*RealArray* **table**=*ColourTable* **xarray**=*RealArray* **yarray**=*RealArray* [**valmax**=*Real*] [**valmin**=*Real*]	Plots a cell_array on a graphics device
CELL_ARRAY/ALTER [/SMOOTH] [/LOG] [/SQRT]	**values**=*RealArray* **table**=*ColourTable* **xarray**=*RealArray* **yarray**=*RealArray* [**valmax**=*Real*] [**valmin**=*Real*] **object**=*Gobject*	Alters a cell_array already drawn on a graphics device

example:

```
# Produce a cell plot of 2-D data on a non-linear grid
>> Cell_array/Draw values=my_points &
      xarray=non_linear_x yarray=non_linear_y &
      table=colourtable:rainbow(80)
```

Cell_Array/Draw

Draws a colour cell plot where the grid is non-linear but can be still be specified by two arrays. For non-gridded data see *Cell_function()*, for linearly gridded data you may wish to use the *Cell()* primitive which automatically produces a grid.

Parameters:

/Smooth

Interpolate colour across each cell.

/Log

Draws a cell plot with logarithmic colour scales.

/Sqrt

Draws a cell plot with square root colour scales.

Values (RealArray)

Defines the cell values of the cell plot, values is a two dimensional array with sufficient data to fit the grid points.

Table (Colourtable)

Defines the colour table for the cell plot.

Xarray (RealArray)

Defines the x gridding of the cell_array.

Yarray (RealArray)

Defines the y gridding of the cell_array.

Valmax (Real)

Cut off maximum above which the colour table will not be mapped.

Valmin (Real)

Cut off minimum below which the colour table will not be mapped.

RESULT = (CellArray)

This returns all of the properties of the Cell_array.

Cell_Array/Alter

Alters the already created cell_array.

Parameters:

/Smooth

Interpolate colour across each cell.

/Log

Draws a logarithmic cell plot.

/Sqrt

Draws a square root cell plot.

Values (RealArray)

Defines the cell values of the cell plot, values is a two dimensional array with sufficient data to fit the grid points.

Table (Colourtable)

Defines the colour table for the cell plot.

Xarray (RealArray)

Defines the x gridding of the cell_array.

Yarray (RealArray)

Defines the y gridding of the cell_array.

Valmax (Real)

Cut off maximum above which the colour table will not be mapped.

Valmin (Real)

Cut off minimum below which the colour table will not be mapped.

Object (Gobject)

The cell_array to alter.

RESULT = (CellArray)

This returns all of the properties of the altered Cell_array.

Cell_function()

Coloured cell plot for ungridded data.

CELL_FUNCTION/DRAW [/SMOOTH] [/LOG] [/SQRT]	**values**=*RealArray* **table**=*ColourTable* **function_name**=*String* [**valmax**=*Real*] [**valmin**=*Real*]	Plots a cell_function on a graphics device
CELL_FUNCTION/ALTER [/SMOOTH] [/LOG] [/SQRT]	**values**=*RealArray* **table**=*ColourTable* **function_name**=*String* [**valmax**=*Real*] [**valmin**=*Real*] **object**=*Gobject*	Alters a cell_function already drawn on a graphics device

example:

```
# Plot some ungridded data with a
# a user specified module function
>> Cell_function/Draw values=my_points &
     function_name="my_gridding_function"
```

Note: Need to load user function with the Module/Load command before use.

Cell_function/Draw

Draws a colour cell plot using a user written module function to supply the location of the data points, for example a non-uniform grid for S(Q) plots. This is the most general of the colour cell plotting functions and requires that the user writes or uses a pre-existing FORTRAN module to calculate the grid for the data. For more information on writing modules, see the *Module()* command and also the "*user notes*" section of this manual. If a the function can be plotted on a linear or non-linear grid, you may be able to use the *Cell()* or *Cell_array()* commands to avoid the need to use a module.

The FORTRAN function supplied to the cell function must be declared in the module in a similar form to the function below with the parameters as specified.

```
SUBROUTINE MY_CELL_GRID_FUNC(FLAGS, VAL_IN, VAL_OUT)

INTEGER FLAGS(*)
DOUBLE PRECISION VAL_IN(*), VAL_OUT(*)

VAL_OUT(1) = (VAL_IN(1) + 10.0) / 11.0 ! X gridding modifier
VAL_OUT(2) = (VAL_IN(2) + 10.0) / 11.0 ! Y gridding modifier

END
```

For cell gridding, the function is supplied with X and Y World coordinates in the VAL_IN(1) and VAL_IN(2) parameters respectively. For examples, if A(10,10) represents values on a grid of 1.0 → 100.0 in X and Y then VAL_IN will contain values in the region 1.0 → 100.0 and you should set VAL_OUT to values in the region 1.0 → 10.0 to represent where they are in the array grid. Any values outside this region will be clipped because they will not actually be on the plotted grid.

Note: this function is the inverse of that supplied for the *Contour_function()* command for efficiency reasons.

Parameters:

/Smooth

Interpolate colour across each cell.

/Log

Draws a logarithmic cell plot.

/Sqrt

Draws a square root cell plot.

Values (RealArray)

Defines the cell values of the cell plot, values is a two dimensional array with sufficient data to fit the grid points.

Table (Colourtable)

Defines the colour table for the cell plot.

Function_name (String)

This is the function name of a function previously written and compiled in fortran.

Valmax (Real)

Cut off maximum above which the colour table will not be mapped.

Valmin (Real)

Cut off minimum below which the colour table will not be mapped.

RESULT = (CellFunction)

This returns all of the properties of the Cell_function.

Cell_function/Alter

Alters an already created cell function.

Parameters:

/Smooth

Interpolate colour across each cell.

/Log

Draws a logarithmic cell plot.

/Sqrt

Draws a square root cell plot.

Values (RealArray)

Defines the cell values of the cell plot, values is a two dimensional array with sufficient data to fit the grid points.

Table (Colourtable)

Defines the colour table for the cell plot.

Function_name (String)

This is the function name of a function previously written and compiled in FORTRAN.

Valmax (Real)

Cut off maximum above which the colour table will not be mapped.

Valmin (Real)

Cut off minimum below which the colour table will not be mapped.

Object (Gobject)

The cell function plot to be altered.

RESULT = (CellFunction)

This returns all of the properties of the Cell_function.

Cell_wedge()

Displays a cell_wedge (colour key) on the graphics device.

CELL_WEDGE/DRAW [/HORIZONTAL] [/VERTICAL]	**object**=*Gobject*	Draws a cell_wedge onto a graphics device.
CELL_WEDGE/ALTER [/HORIZONTAL] [/VERTICAL]	**object**=*Gobject*	Alters a cell_wedge already drawn on a graphics device.

example:

```
# Annotate an already drawn cell plot
>> my_cell = Cell:Draw(w.y, colourtable:heat(80))
>> Cell_Wedge/draw/horizontal  my_cell
```

Note: Must be in a separate window to the cell plot. By default the wedge is vertical.

Cell_wedge/Draw

Provides colour key annotation to a cell plot if required. The cell wedge draws either a vertical or horizontal colour key using the same colour table as the cell plot it is annotating. The cell object (either a Cell, Cell_array or Cell_function) must already exist and the object reference of the Cell plot must be supplied for the wedge to be drawn.

Parameters:

/Horizontal

The cell_wedge will be drawn horizontally.

/Vertical

The cell_wedge will be drawn vertically.

Object (Gobject)

Give a reference to the colour cell plot for which the key is required.

RESULT = (CellWedge)

This returns all of the properties of the cell_wedge.

Cell_wedge/Alter

Alters an already created wedge.

Parameters:

/Horizontal

The cell_wedge will be drawn horizontally.

/Vertical

The cell_wedge will be drawn vertically.

Object (Gobject)

Give an object reference to the cell_wedge to be altered.

RESULT = (CellWedge)

This returns all of the properties of the altered cell_wedge.

Contour()

Makes a contour plot on the graphics device.

CONTOUR/DRAW [/LOG] [/SQRT]	**values**=*RealArray* [**contours**=*RealArray*] [**ncont**=*Integer*] [**tr**=*RealArray*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Draw a contour plot onto a graphics device.
CONTOUR/ALTER [/LOG] [/SQRT]	[**values**=*RealArray*] [**contours**=*RealArray*] [**ncont**=*Integer*] [**tr**=*RealArray*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Contour*	Alters a previously drawn contour plot on a graphics device.

example:

```
# Produce a contour plot of 2-D data
>> cp = Contour:Draw(my_points)
```

Note: By default the contour is linear. Use contours or ncont, not both.

Contour/Draw

Draws a contour plot of the data supplied. The contour plot should be drawn in a two-D window (see *Win_twod()*). The Contour() function is the simplest of the contouring functions and draws a contour from linearly gridded data or gridding specified by a transformation matrix. For contouring non-linearly gridded data or non-gridded data use *Contour_array()* and *Contour_function()* respectively.

If no transformation matrix is specified, the contour assumes linear gridding based on the window in which the data is being contoured.

Parameters:

/Log

Draws with logarithmic contour spacing.

/Sqrt

Draws with square root contour spacing.

Values (RealArray)

Defines the level values of the contour plot, values is a two dimensional array with sufficient data to fit the grid points.

Contours (RealArray)

Sets the contours explicitly by specifying the heights (in raw data values) of the contours. If this is specified, the number of contours will be taken from the data array and Ncont is ignored.

Ncont (Integer)

Draws ncont contours, by default, evenly spaced between Valmin and Valmax, but modified by /log or /sqrt.

Tr (RealArray)

Array defining a transformation between the i, j grid of the data array and the window coordinates. The window coordinates of the array point a[i,j] are given by:

x = tr[1] + tr[2] * i + tr[3] * j
y = tr[4] + tr[5] * i + tr[6] * j

Usually tr[3] and tr[5] are zero - unless the coordinate transformation involves a rotation or shear.

The default transformation if this parameter is not specified is to map linearly onto the window to fit the plot.

Colour (Colour)

Sets the contour plots colour.

Line_type (LineStyle)

Sets the line type of the contour plot. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Sets the contour plots line width.

RESULT = (Contour)

Returns all of the properties of the newly created contour plot.

Contour/Alter

Alters a contour plot previously drawn on a graphics device.

Parameters:

/Log

Draws with logarithmic contour spacing.

/Sqrt

Draws with square root contour spacing.

Values (RealArray)

Defines the cell values of the cell plot, values is a two dimensional array with sufficient data to fit the grid points.

Contours (RealArray)

Sets the contours explicitly by specifying the heights (in raw data values) of the contours. If this is specified, the number of contours will be taken from the data array and Ncont is ignored.

Ncont (Integer)

Draws ncont contours, by default, evenly spaced between Valmin and Valmax, but modified by /log or /sqrt.

Tr (RealArray)

Array defining a transformation between the i, j grid of the data array and the window coordinates. The window coordinates of the array point a[i,j] are given by:

x = tr[1] + tr[2] * i + tr[3] * j
y = tr[4] + tr[5] * i + tr[6] * j

Usually tr[3] and tr[5] are zero - unless the coordinate transformation involves a rotation or shear.

The default transformation if this parameter is not specified is to map linearly onto the window to fit the plot.

Colour (Colour)

Sets the contour plots colour.

Line_type (LineStyle)

Sets the line type of the contour plot. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Sets the contour plots line width.

Object (Contour)

Shows which contour plot is to be altered.

RESULT = (Contour)

Returns all of the properties of the altered contour plot.

Contour_array()

Plots a contour_array on the graphics device

CONTOUR_ARRAY/DRAW [/LOG] [/SQRT]	**values**=*RealArray* **contours**=*RealArray* **ncont**=*Integer* **xarray**=*RealArray* **yarray**=*RealArray* [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Draw a contour array onto a graphics device.
CONTOUR_ARRAY/ALTER [/LOG] [/SQRT]	[**values**=*RealArray*] [**contours**=*RealArray*] [**ncont**=*Integer*] [**xarray**=*RealArray*] [**yarray**=*RealArray*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Gobject*	Alters a previously drawn contour array on a graphics device.

example:

```
# Produce a contour plot of 2-D data on a non-linear grid
>> Contour_array/Draw values=my_points &
      xarray=non_linear_x yarray=non_linear_y
```

Note: Use contours or ncont, not both.

Contour_Array/Draw

Draws a contour plot of the data supplied. The contour plot should be drawn in a two-D window (see *Win_twod()*). The Contour_array() function is for contouring with non-linearly gridded data. For contours with linearly gridded data or non-uniform gridding use the *Contour()* or *Contour_function()* commands respectively.

Parameters:

/Log

Draws logarithmically spaced contours.

/Sqrt

Draws square root spaced contours.

/Linear

Draws a linear contour array.

Values (RealArray)

Defines the levels of the contour plot, values is a two dimensional array with sufficient data to fit the grid points.

Contours (RealArray)

Sets the contours explicitly by specifying the heights (in raw data values) of the contours. If this is specified, the number of contours will be taken from the data array and Ncont is ignored.

Ncont (Integer)

Draws ncont contours, by default, evenly spaced between Valmin and Valmax, but modified by /log or /sqrt.

Xarray (RealArray)

Defines the x gridding of the contours.

Yarray (RealArray)

Defines the y gridding of the contours.

Colour (Colour)

Sets the contour colour.

Line_type (LineStyle)

Sets the line type of the contours. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Sets the contour line width.

RESULT = (ContourArray)

Returns all of the properties of the newly created contour array.

Contour_Array/Alter

Alters a contour array previously drawn on a graphics device.

Parameters:

/Log

Draws logarithmically spaced contours.

/Sqrt

Draws square root spaced contours.

/Linear

Draws a linear contour array.

Values (RealArray)

Defines the level values of the contour plot, values is a two dimensional array with sufficient data to fit the grid points.

Contours (RealArray)

Sets the contours explicitly by specifying the heights (in raw data values) of the contours. If this is specified, the number of contours will be taken from the data array and Ncont is ignored.

Ncont (Integer)

Draws ncont contours, by default, evenly spaced between Valmin and Valmax

Xarray (RealArray)

Defines the x gridding of the contours.

Yarray (RealArray)

Defines the y gridding of the contours.

Colour (Colour)

Sets the contour colour.

Line_type (LineStyle)

Sets the line type of the contours. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Sets the contour line width.

Object (Integer)

Shows which contour array is to be altered, in any one window.

RESULT = (ContourArray)

Returns all of the properties of the altered contour array.

Contour_function()

Plots contours for ungridded data.

CONTOUR_FUNCTION/DRAW [/LOG] [/SQRT]	**values**=*RealArray* **contours**=*RealArray* **ncont**=*Integer* **function_name**=*String* [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Draw a contour function onto a graphics device.
CONTOUR_FUNCTION/ALTER [/LOG] [/SQRT]	[**values**=*RealArray*] [**contours**=*RealArray*] [**ncont**=*Integer*] [**function_name**=*String*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Integer*	Alter a previously drawn contour function on a graphics device.

example:

```
# Plot some ungridded data with a
# a user specified module function
>> Contour_function/Draw values=my_points &
      function_name="my_gridding_function"
```

Note: Use contours or ncont, not both.

Contour_Function/Draw

Draws a contour plot using a user written module function to supply the location of the data points, for example a non-uniform grid for S(Q,E) plots. This is the most general of the contour plotting functions and requires that the user writes or uses a pre-existing FORTRAN module to calculate the grid for the data. For more information on writing modules, see the *Module()* command and also the "*user notes*" section of this manual. If a the function can be plotted on a linear or non-linear grid, you may be able to use the *Contour()* or *Contour_array()* commands to avoid the need to use a module.

The FORTRAN function supplied to the contour function must be declared in the module in a similar form to the function below with the parameters as specified.

```
SUBROUTINE MY_CONT_GRID_FUNC(FLAGS, VAL_IN, VAL_OUT)
INTEGER FLAGS(*)
DOUBLE PRECISION VAL_IN(*), VAL_OUT(*)

VAL_OUT(1) = 11.0 * VAL_IN(1) - 10.0    ! X gridding modifier
```

```
VAL_OUT(2) = 11.0 * VAL_IN(2) - 10.0    ! Y gridding modifier

END
```

For contour gridding, the function is supplied with array grid coordinates in the VAL_IN(1) and VAL_IN(2) parameters respectively and must return world co-ordinates. For examples, if A(10,10) represents values on a grid of 1.0 → 100.0 in X and Y then VAL_IN will contain values in the region 1.0 → 10.0 and you should set VAL_OUT to values in the region 1.0 → 100.0 to represent where they are in world co-ordinates. Any values outside this region will be clipped by the current window.

The FLAGS(1) parameter is a visibility flag. If set to 0 the contour will not be drawn, if set to 1 the contour will be drawn. VAL_IN(3) also contains the value a(i,j) should it be required. An example of the use of the last two parameters described might be to provided a high or low cutoff below or above which contours are not plotted.

Note: this function is the inverse of that supplied for the *Cell_function()* command for efficiency reasons.

Parameters:

/Log

Draws logarithmic spaced contours.

/Sqrt

Draws square root spaced contours.

Values (RealArray)

Defines the values of the contour function, values=(xarray,yarray).

Contours (RealArray)

Sets the contours explicitly by specifying the heights (in raw data values) of the contours. If this is specified, the number of contours will be taken from the data array and Ncont is ignored.

Ncont (Integer)

Draws ncont contours, by default, evenly spaced between Valmin and Valmax, values are modified by /log or /sqrt.

Function_name (String)

This is the function name of a function previously written and compiled in FORTRAN.

Colour (Integer)

Sets the contour functions colour.

Line_type (Integer)

Sets the line type of the contour function. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Sets the contour functions line width.

RESULT = (ContourFunction)

Returns all of the properties of the newly created contour function.

Contour_Function/Alter

Alters a contour function previously drawn on a graphics device.

Parameters:

/Log

Draws logarithmically spaced contours.

/Sqrt

Draws square root spaced contours.

Values (RealArray)

Defines the values of the contour function, values=(xarray,yarray).

Contours (RealArray)

Sets the contours explicitly by specifying the heights (in raw data values) of the contours. If this is specified, the number of contours will be taken from the data array and Ncont is ignored.

Ncont (Integer)

Draws ncont contours, by default, evenly spaced between Valmin and Valmax, unless modified by /log or /sqrt.

Function_name (String)

This is the function name of a function previously written and compiled in FORTRAN.

Colour (Integer)

Sets the contour functions colour.

Line_type (Integer)

Sets the line type of the contour function. Such as $full, $dash, $dot_dash and $dot.

Line_thickness (Real)

Sets the contour functions line width.

Object (Gobject)

The contour function plot to be altered

RESULT = (ContourFunction)

Returns all of the properties of the altered contour function.

Contour_label()

Labels the contours in the contour plot

CONTOUR_LABEL/DRAW	[**text**=*String*] [**start**=*Integer*] [**end**=*Integer*] [**st**=*Integer*] [**intval**=*Integer*] [**minint**=*Integer*] [**font**=*Font*] [**size**=*Real*] [**colour**=*Colour*] **object**=*Gobject*	Draw labels on a contour plot.
CONTOUR_LABEL/ALTER	[**text**=*String*] [**start**=*Integer*] [**end**=*Integer*] [**st**=*Integer*] [**intval**=*Integer*] [**minint**=*Integer*] [**font**=*Font*] [**size**=*Real*] [**colour**=*Colour*] **object**=*Gobject*	Alter labels on a contour plot.

example:

```
# put labels on a contour plot, use
# label formatting to add units.
>> my_cont = Contour:Draw(w.y)
>> Contour_label object=my_cont &
     text="%gmEV"
```

Note: Default for text is just to give the values.

Contour_Label/Draw

Add labels to the contours of the specified contour plot. The labels may be formatted by the optional "Text" parameter if required.

Parameters:

Text (String)

Sets the format of the labels if required. The default is just to print the numbers but by adding a format string it is possible to gain full control over the label format. A "%g" in the formatting string converts numbers in general format (G format in FORTRAN). A "%f" formats in floating point (F) format and "%e" formats in scientific (E) format. As an example, a string formatting in floating point format in a field of 10 with four decimal places would be "%10.4f meV".

Start (Integer)

Contour level of first label.

End (Integer)

Contour level of last label.

St (Integer)

The number of steps between labelled contours.

Intval (Integer)

The number of cells crossed by a contour before another label is drawn. Gives control over the number of labels drawn on any one contour. The default is 20.

Minint (Integer)

The minimum number of cells to be crossed by a contour to justify adding a label. Contours crossing less than this number of cells will remain unlabelled. The default is 10.

Font (Font)

Sets the labels font.

Size (Real)

Sets the label size.

Colour (Colour)

Sets the label colour.

Object (Gobject)

Specifies the contour plot for the labels to draw on.

RESULT = (ContourLabels)

Returns all of the properties of the newly created contour labels.

Contour_Label/Alter

Alters previously drawn contour labels.

Parameters:

Text (String)

Sets the format of the labels if required. The default is just to print the numbers but by adding a format string it is possible to gain full control over the label format. A "%g" in the formatting string converts numbers in general format (G format in FORTRAN). A "%f" formats in floating point (F) format and "%e" formats in scientific (E) format.

Start (Integer)

Contour level of first label.

End (Integer)

Contour level of last label.

St (Integer)

Allows label values to be specified explicitly, the steps between labels here.

Intval (Integer)

The number of cells crossed by a contour before another label is drawn. Gives control over the number of labels drawn on any one contour. The default is 20.

Minint (Integer)

The minimum number of cells to be crossed by a contour to justify adding a label. Contours crossing less than this number of cells will remain unlabelled. The default is 10.

Font (Font)

Sets the labels font.

Size (Real)

Sets the label size.

Colour (Colour)

Sets the label colour.

Object (Gobject)

Specifies the contour label to alter.

RESULT = (ContourLabels)

Returns all of the properties of the altered contour labels.

Multi_plot()

Controls multiplot creation and plotting

MULTI_PLOT/CREATE [/BINCENTRE] [/NOTCENTRED]	**data**=*RealArray* [**numhist**=*Integer*] [**chmin**=*Integer*] [**chmax**=*Integer*] [**ygap**=*Real*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*]	Creates the initial parameters of the multiplot.
MULTI_PLOT/SPECTRA	**data**=*Range or RealArray*	Adds the final parameter (the y array's) needed to draw a multiplot.
MULTI_PLOT/DRAW		Draws a multiplot onto a graphics device.
MULTI_PLOT/ALTER [/BINCENTRE] [/NOTCENTRED]	[**numhist**=*Integer*] [**chmin**=*Integer*] [**chmax**=*Integer*] [**ygap**=*Real*] [**colour**=*Colour*] [**line_type**=*LineStyle*] [**line_thickness**=*Real*] **object**=*Gobject*	Alter the multiplot previously drawn on a graphics device.

example:

```
# Create a multiplot and display it
>> w=get(1:10)
>> multi_plot/create data=w.x
>> multi_plot/spectra data=w.y
>> win_multiplot 0.1 0.9 0.1 0.9
>> mp = multi_plot:draw()
```

Note: Multiplot window must be created after the multi_plot but before drawing.

Multi_Plot/Create

Creates a multiplot without adding the data.

Parameters:

/Bincentre

Centres the data before plotting (for point mode data)

/Notcentred

Plots the data as it is. (by default the data is not centred).

Data (RealArray)

Sets the x data points, these must be identical for all Y data

Numhist (Integer)

Sets the number of histograms wanted, otherwise defaults to all the Y arrays

Chmin (Integer)

Sets an X data minimum cutoff.

Chmax (Integer)

Sets an X data maximum cutoff.

Ygap (Real)

Sets an array of gap values between the data points

Colour (Colour)

Sets the colour of the multiplot.

Line_type (LineStyle)

Sets the line type of the multiplot.

Line_thickness (Integer)

Sets the line thickness of the multiplot.

RESULT = (Multiplot)

Returns all of the properties of the newly created multiplot.

Multi_plot/Spectra

Adds all of the y data to the already created multiplot.

Parameters:

Data (RealArray or Range)

Sets the Y data. A two dimensional array with the second dimension the same as the X-array (or one less for bin-mode data). For example, x[4000] will require y[n, 4000] or y[n,3999] for histogram mode for n spectra.

For an already open data file, a Range may be specified as an integer array of spectrum numbers to be read from the raw data file. See *Range* data type. This is used where it would not be possible to hold all of the data for the multiplot in memory.

Multi_plot/Draw

Draws the multiplot. Note that a window must be created before the Multiplot/Draw command is given but after the multiplot has been created and scaled with the Multiplot/Create and Multiplot/Spectra commands.

Multi_plot/Alter

Alters the multiplot. The ydata and xdata cannot be altered.

Parameters:

/Bincentre

Centres the data.

/Notcentred

Plots the data as it is. (by default the data is not centred).

Numhist (Integer)

Sets the number of histograms wanted.

Chmin (Integer)

Sets the data point to start from..

Chmax (Integer)

Sets the data point to finish at.

Ygap (RealArray)

Sets an array of gap values between the data points

Colour (Colour)

Sets the colour of the multiplot.

Line_type (Integer)

Sets the line type of the multiplot.

Line_thickness (Integer)

Sets the line thickness of the multiplot.

Object (Gobject)

Shows which multiplot is to be altered, in any one window.

RESULT = (Multiplot)

Returns all of the properties of the altered multiplot.

Colour()

Define a new graphics colour based on one of several models

COLOUR:RGB()	**r**=*Real* **g**=*Real* **b**=*Real*	Create new colour using RGB model
COLOUR:HLS()	**h**=*Real* **l**=*Real* **s**=*Real*	Create new colour using HLS model
COLOUR:NAMED()	**name**=*String*	Access a named palette colour

example:

```
# Draw lines in colours
# created from two different models
>> Line/Draw 0.0 0.0 1.0 1.0 colour:rgb(0.5, 0.6, 0.7)
>> Line/Draw 0.0 0.0 1.0 1.0 colour:named("SteelBlue")
```

Note: The number of simultaneous colours available depends on the display hardware.

Colour:Rgb()

Define a new colour index value given a set of RGB (Red, Green, Blue) values. The values of R, G and B are real numbers in the range 0.0-1.0.

Parameters:

r (Real)

Colour intensity value for red

g (Real)

Colour intensity value for green

b (Real)

Colour intensity value for blue

RESULT = (Colour)

Returns a colour index value which may be used instead of the fixed colours (e.g. $RED, $GREEN) anywhere a colour parameter is required in the Open GENIE graphics system. The index is only valid in the current Open GENIE session.

Colour:Hls()

Define a new colour index value given a set of HLS (Hue, Lightness, Saturation) values.

Parameters:

h (Real)

H is an angle in degrees with Blue=0.0 (or 360.0), Red=120.0, Green=240.0

l (Real)

L ranges from 0.0 (black) to 1.0 (white), for example red is (120, 0.5, 1.0).

s (Real)

S ranges from 0.0 (gray) to 1.0 (pure colour), for example black is (any, 0.0, 0.0).

RESULT = (Colour)

Returns a colour index value which may be used instead of the fixed colours (e.g. $RED, $GREEN) anywhere a colour parameter is required in the Open GENIE graphics system. The index is only valid in the current Open GENIE session.

Colour:Named()

Look up and instantiate a colour by name. Generally this will only be supported on a windowing system which supports a standard set of colours by name, for example X-Windows.

Parameters:

Name (String)

Looks up the named colour on the display colour palette. This may change depending on the windowing system and graphics hardware so care should be taken or code using this may become device dependent. Names for X-Windows colours may be found in the "rgb.txt" file which comes with Open GENIE.

RESULT = (Colour)

Returns a colour index value which may be used instead of the fixed colours (e.g. $RED, $GREEN) anywhere a colour parameter is required in the Open GENIE graphics system. The index is only valid in the current Open GENIE session.

Colourtable()

Manipulate graphics colour tables (used with Cell() commands).

COLOURTABLE:CREATE()	**ncolours**=*Integer* **red**=*RealArray* **green**=*RealArray* **blue**=*RealArray* **contrast**=*Real*	Creates and returns a new colour table.
[/NOSHARE]		Avoid sharing colours
COLOURTABLE/DELETE	**table**=*ColourTable*	Deletes and frees up colours from an existing colourtable.

example:

```
# Create, use in a cell plot and then
# delete a colourtable.
>> ct1 = Colourtable:create(80,ra,ga,ba, 1.0,1.0)
>> Cell/alter object=obj(1,2,3) table=ct1
>> Colourtable/delete ct1
```

Note: The data in the real arrays will be interpolated if ncolours exceeds the number of elements in the ra, ga and ba arrays.

Colourtable:Create()

This command creates a colourtable given three arrays of colour values, one each for red, green and blue. The created colourtable will normally be used to provide a colour scale for a cell array style plot and will provide "ncolours" different colours for the plot using it. Note that the colour arrays need not have ncolours elements, the colourtable command will interpolate to provide intermediate colours if necessary.

Where possible the Colourtable() command will allocate colours which are already being used to save on a potentially limited number of colour indices. As colour slots can be relatively short on some devices, it is usually a good idea to deallocate colourtables immediately after they have been used.

Parameters:

/Noshare

If this qualifiers is specified, the colourtable command avoids sharing similar colours between colourtables. The most likely use of this is if one plot is assigning colour tables with the intention of colour cycling (it would not be desirable to colour cycle any shared colours!). Note that colour slot usage will be much higher with this qualifier.

Red (RealArray)

This array specifies how much red goes into the new colour, values must be between 0.0 and 1.0.

Green (RealArray)

This array specifies how much green goes into the new colour, values must be between 0.0 and 1.0.

Blue (RealArray)

This array specifies how much blue goes into the new colour, values must be between 0.0 and 1.0.

Ncolours (Real)

Number of colours in the colour table.

Contrast (Real)

Allows an adjustment of the contrast by scaling each colour value by a power (default 1.0).

RESULT = (ColourTable)

Returns the newly created colour table.

Colourtable/Delete

Deallocates a previously allocated colourTable, this allows new colourTables to be allocated with the correct number of colours (only allows 84 colours in total unless changed using an X-Windows resource).

Parameters:

Table (ColourTable)

Colour table to free up.

Dev()

Locates and returns device object given its index.

DEV() **devnum**=*Integer* Absolute index of the object.

example:

```
# Close the third device
# and the current device
>> device/close dev(3)
>> device/close dev(devnum=0)
```

Dev

This is one of four functions which return a reference to a graphical object when given index information about the location of the object in the graphics object subsystem (*Dev(), Pic(), Win()* and *Obj()*).

Positive numbers give the number of the object in its list, starting at 1. Negative numbers give the number of the object starting from the last item in the list and counting backwards, starting at -1 (this is often the easiest way to find objects). Zero represents the last object created, normally the current object.

All open genie devices are stored in a single device list so it is only necessary to specify from a small number of devices. By default, device windows are labeled with the number of the device.

Parameters:

Devnum (Integer)

The position within the device list of the required device.

RESULT = (Gobject)

Returns the device specified.

Obj()

Locates and returns a graphics primitive object given its location.

OBJ()	**picnum**=*Integer* **winnum**=*Integer* **objnum**=*Integer*	Absolute index of the object.

example:

```
# Undraw the third object in the
# current picture and window
>> undraw obj(0,0,3)
```

Note: Picture and window indices of 0 represent the current values.

Obj

This is one of four functions which return a reference to a graphical object when given index information about the location of the object in the graphics object subsystem (*Dev(), Pic(), Win()* and *Obj()*).

Positive numbers give the number of the object in its list, starting at 1. Negative numbers give the number of the object starting from the last item in the list and counting backwards, starting at -1 (this is often the easiest way to find objects). Zero represents the last object created, normally the current object.

All Open GENIE primitive graphical objects are stored within a window which itself is stored within a picture, this means that to refer to a primitive object, say a line, it is necessary to specify the index of the picture and the window in which the line was drawn. Normally references are within the current picture and window so these values can be set to zero.

Parameters:

Picnum (Integer)

The position within the picture list of the required picture.

Winnum (Integer)

The position within the window list of the required window.

Objnum (Integer)

The position within the primitive object list of the required primitive.

RESULT = (Gobject)

Returns the graphics primitive specified.

Pic()

Locates and returns a picture object given its index.

PIC() **picnum**=*Integer* Absolute index of the object.

example:

```
# Redraw the third picture
>> redraw pic(3) device=dev(2)
```

Pic

This is one of four functions which return a reference to a graphical object when given index information about the location of the object in the graphics object subsystem (*Dev()*, *Pic()*, *Win()* and *Obj()*).

Positive numbers give the number of the object in its list, starting at 1. Negative numbers give the number of the object starting from the last item in the list and counting backwards, starting at -1 (this is often the easiest way to find objects). Zero represents the last object created, normally the current object.

All Open GENIE pictures are stored in a single picture list, regardless of the number of devices opened (pictures may be redrawn to one or more devices using the *Redraw()* command).

Parameters:

Picnum (Integer)

The position within the picture list of the required picture.

RESULT = (Gobject)

Returns the picture object specified.

Pic()

Locates and returns a picture object given its index.

PIC() **picnum**=*Integer* Absolute index of the object.

example:

```
# Redraw the third picture
>> redraw pic(3) device=dev(2)
```

Pic

This is one of four functions which return a reference to a graphical object when given index information about the location of the object in the graphics object subsystem (*Dev()*, *Pic()*, *Win()* and *Obj()*).

Positive numbers give the number of the object in its list, starting at 1. Negative numbers give the number of the object starting from the last item in the list and counting backwards, starting at -1 (this is often the easiest way to find objects). Zero represents the last object created, normally the current object.

All Open GENIE pictures are stored in a single picture list, regardless of the number of devices opened (pictures may be redrawn to one or more devices using the *Redraw()* command).

Parameters:

Picnum (Integer)

The position within the picture list of the required picture.

RESULT = (Gobject)

Returns the picture object specified.

Win()

Locates and returns a graphics window object given its location.

WIN() **picnum**=*Integer* **winnum**=*Integer* Absolute index of the object.

example:

```
# Copy the 2nd window in the
# last but one picture into a
# new picture
>> pic_add item=win(-2,2) picture=picture()
```

Note: A picture value of 0 specifies the current picture.

Win

This is one of four functions which return a reference to a graphical object when given index information about the location of the object in the graphics object subsystem (*Dev()*, *Pic()*, *Win()* and *Obj()*).

Positive numbers give the number of the object in its list, starting at 1. Negative numbers give the number of the object starting from the last item in the list and counting backwards, starting at -1 (this is often the easiest way to find objects). Zero represents the last object created, normally the current object.

All Open GENIE window objects are stored within a picture, this means that to refer to a window, it is necessary to specify the index of the picture as well. If the window is in the current picture, zero can be used for the "picnum" parameter.

Parameters:

Picnum (Integer)

The position within the picture list of the required picture.

Winnum (Integer)

The position within the window list of the required window.

RESULT = (Gobject)

Returns the graphics window specified.

Chapter 3

GENIE-V2 Emulation and Data I/O

This section of the reference manual documents routines and facilities which have been added to Open GENIE for previous users of the GENIE-V2 program These commands are in general not essential to using Open GENIE but may be a convenient starting point for accessing single spectra/scan data. Several of the standard Open GENIE commands also work in concert with these commands to aid compatibility. For example, the *Get()* command will still take notice of a default input file selected using the GENIE-V2 disk, directory, instrument and extension defaults.

The Genie I/O system maintains several internal variables that control:

- The current input file
- The current output file
- The current disk
- The current directory
- The current instrument
- The current file extension

Current values may be viewed by using *Show()* commands, and changed by using the *Set()* commands.

The GENIE-V2 compatibility commands are listed below.

Assign	Select a run for input by giving the ISIS run number
Cfn	Construct the full filename from run number and other defaults
Groupbins	Groups histogram bins by averaging contents

Jump	Similar to the *System()* command.
Keep	Similar to the *Hardcopy()* command.
S	Identical to the *Get()* command.
Scatmode	Return GENIE-V2 Energy mode description
Set	Set user defined defaults
Setpar	GENIE-V2 style parameter setting for the *Units()* command
Show	Show user defined defaults and system variables

GENIE-V2 Emulation and Data I/O Command Reference

Assign()

Select a run for input by giving the ISIS run number

ASSIGN **p1**=*Integer* Selects a default run for input with the S() command.

example:

```
# Select run 2004 and read in a spectrum
>> Assign 2004
>> w=s(2)        # 2nd Spectrum
```

Note: Assign nnn is equivalent to Set/File/Input Cfn(nnn)

Assign

The assign command is designed to allow a shorthand of specifying the ISIS run number to select the input file to read from. Essentially it concatenates the defaults for disk, directory, instrument and extension and then makes the resultant file the default file. It is based on the *Cfn()* command.

Parameters:

P1 (Integer)

An existing ISIS run number. Before this can make sense, the disk, directory, instrument and extension defaults must have been set up with the appropriate *Set()* commands.

Cfn()

Construct the full filename from run number and defaults

CFN	[**number**=*Integer*]	Construct a full path name from the given run number.
CFN	[**name**=*String*]	Construct a full path name from the given file name.

example:

```
>> Set/Disk "IRIS$DISK0:"
>> Set/Dir "[IRSMGR.DATA]"
>> Set/Inst "IRS"
>> Set/Ext ".RAW"
>> printn Cfn(345)
IRIS$DISK0:[IRSMGR.DATA]IRS00345.RAW
>> printn Cfn("special.dat")
IRIS$DISK0:[IRSMGR.DATA]SPECIAL.DAT
```

Note: Cfn() adds the appropriate number of 0s for ISIS raw file names.

Cfn

The Cfn() function provides the formatting necessary to convert between integer run numbers and full file names by making use of previously set defaults. The run number conversion is ISIS file name specific however using Cfn() with a file name is general (note that on Unix systems, there is no need to specify the Disk default).

Parameters:

Number (Integer)

Integer run number which is automatically converted to a zero padded string before concatenation into a file name.

Name (String)

A complete filename but without any path (i.e. Disk/Directory) specification.

RESULT = (String)

A fully constructed file name which can be used to access the data.

Groupbins()

Groups histogram bins by averaging the bin contents.

GROUPBINS	**wksp**=*Workspace* **nbin**=*Integer* [**undef**=*Real*]	Group bins in a GENIE-V2 format workspace.
GROUPBINS()	**xarray**=*Realarray* **yarray**=*Realarray* **earray**=*Realarray* **nbin**=*Integer* [**undef**=*Real*]	Group bins in a spectrum given X, X, and E arrays.

example:

```
# Group the bins in a workspace by tens.
>> work = s(1)
>> Groupbins work 10 undef=0.0
```

Note: Undef gives a value to assume when undefined elements are met during the averaging process.

Groupbins

The Groupbins() function provides a way of reducing (in a fairly simple manner) a spectrum with large number of bins. The process averages every "nbin" bins by replacing the group of bins with a single new bin. This process is less sophisticated than that used by the Rebin() command and is usually used before displaying a large spectrum graphically to achieve a degree of smoothing.

The first form of the command takes a workspace and modifies the binning to return a re-binned result or to change the original workspace (depending on whether the command is specified as a *function* or *keyword* command).

The second form of the command takes X, Y, and E data arrays where the E and Y arrays are of the same length and the X array is one element longer (for the extra bin boundary). The result is a workspace containing just the rebinned X, Y and E arrays. The original arrays are unaffected.

Parameters:

Wksp (Workspace)

Any workspace W containing a histogram stored in arrays in W.X, W.Y and W.E. Bin grouping does not work on a two dimensional workspace.

Nbin (Integer)

Number of bins to group together.

Undef (Real) [default = 0.0]

Because Open GENIE allows arrays to contain undefined elements, it is possible that the user may wish to set any undefined elements to a specific value before the bin grouping is carried out (usually zero).

Xarray, Yarray, Earray (Realarray)

These are the three arrays representing the histogram to be grouped. The Xarray parameter specifies the histogram bin boundaries, the Yarray parameter the data values and the E array parameter the Error values for the Y array.

RESULT = (Workspace)

Either the bin-grouped input workspace or a new workspace created to hold just the histogram arrays. Which ever form of the command is used, the resultant workspace will have X, Y and E fields.

Jump()

Similar to the System() command.

JUMP/S	**acommand**=*String*	Execute one operating system command
JUMP/P		Create an operating system session within Open GENIE

example:

```
# Use Jump/s to print a file
>> Jump/S "print/q=sys$lsr0 genie.ps"
```

Note: For new programs use the System() command.

Jump/S

This executes a single operating system command (eg "print") from within Open GENIE. Jump does not return a status so it is preferable to use the *System()* command.

Parameters:

Acommand (String)

The command and parameters (if any) to execute.

Jump/P

Jumps out of Open GENIE into a terminal session but keeps Open GENIE running in the background. To return to Open GENIE type "exit" on Unix or "logout" on VMS.

Keep()

The keep command is no longer supported. Use Hardcopy() for saving graphics and the Getcursor() and Asciifile() commands for retrieving and storing peak location data.

Keep

The Keep/Hardcopy command has been superceded by the *Hardcopy()* command for saving graphics.

The other keep commands (originally used for storing peak information) can now be specified as simple procedures using the *Getcursor()* command to get the co-ordinate data and the *Asciifile()* command to write the data in any format into an ASCII formatted file.

S()

Get data, now identical to the Get() command. See Get/Help

S

The S() command now behaves identically to the *Get()* command but it is kept as an alias for consistency.

Scatmode()

Return GENIE-V2 Energy mode description (emode in a GENIE-V2 workspace)

SCATMODE	[**emode**=*Integer*]	Returns a string giving the meaning of the emode value.

example:

```
# Get a description from an emode value of 1
>> printn Scatmode(1)
inelastic (direct geometry)
```

Scatmode

Simply provides a textual description to translate the numeric value of this parameter. This parameter says how to interpret the efixed value during a units operation.

Parameters:

Emode (Integer)

Emode value (0, 1 or 2)

RESULT = (String)

String description of the instrument energy mode.

Set()

Set user defined defaults.

SET/DISK	**value**=*String*	Select the default disk
SET/DIR	**value**=*String*	Select the default directory
SET/INST	**value**=*String*	Select the default instrument short name
SET/EXT	**value**=*String*	Select the default file extension
SET/FILE [/INPUT] [/OUTPUT]	**value**=*String*	Set the default input or output files

example:

```
# Change the file for saving data to
>> Set/file/output "Mydat.in3"
```

Note: Set/File/Input is the default for Set/File

Set/File

Allows selection of the default file for input or output. This new Open GENIE command overrides the older commands listed below which set parts of the path individually. The full pathname for the file must be supplied when Set/File is being used, but once set both the *Get()* and *Put()* commands will accept the Input/Output files as defaults.

Parameters:

/Input

Sets the default file for input

/Output

Sets the default file for output

Value (String)

The full pathname for the default input or output file.

Set/Disk

Sets the default disk for the input file. This is not needed on Unix systems but can be useful on VMS or Windows/NT.

Parameters:

Value (String)

The full disk name (or disk logical name on VMS - e.g. "USER$DISK:")

Set/Dir

Sets the default directory for the input file.

Parameters:

Value (String)

The full directory path for the file. (e.g. [TFXA.DATA] on VMS or /home/user/bob/ on Unix - the trailing / is needed on Unix)

Set/Inst

Sets the default instrument short name. This is an ISIS specific three letter abbreviation for the instrument name which constitutes part of the file name.

Parameters:

Value (String)

The instrument short name (e.g. IRS)

Set/Ext

Sets the default file extension for the input file.

Parameters:

Value (String)

The full file extension (e.g. ".RAW")

Setpar()

GENIE-V2 style T-O-F parameter setting for the Units() command.

SETPAR	**wksp**=*Workspace* [**l1**=*Real*] [**l2**=*Real*] [**tt**=*Real*] [**em**=*Integer*] [**ef**=*Real*] [**d**=*Real*] [**t**=*String*]	Set time of flight parameters
[/R]		Take angles to be values in radians

example:

```
# Set a value of two-theta that was not in the raw data file
>> w = s(1)
>> Setpar w tt=50.0
>> wd = Units:d(w)
```

Note: Could probably be done more easily by assignment (i.e. w.twotheta = 50.0)

Setpar

The Setpar() command emulates the GENIE-V2 command which set Time-of-Flight parameters for a workspace so that the *Units()* command could be used. Open GENIE behaves differently, in that it will read the parameters from a data file automatically when reading spectra. Setpar() can be used however to fix parameters which had an incorrect or default value in the raw file. Note that the problem is trivial in Open GENIE anyway because the correct values can simply be assigned to the appropriate workspace field by using GCL (as in the note above).

Parameters:

/R

Interpret the two theta angle as a value given in radians.

L1 (Real)

Primary flight path (m)

L2 (Real)

Secondary flight path (m)

Tt (Real)

Two theta angle (in degrees by default but see the /R qualifier).

Em (Integer)

Selects the energy mode of the instrument which affects how the "Ef" fixed energy parameter is interpreted by the *Units()* command.

0 = Elastic
1 = Inelastic (Direct geometry)
2 = Inelastic (Indirect geometry)

Ef (Real)

Fixed energy value (meV)

D (Real)

The hold off time (delta) in microseconds.

T (String)

Workspace title string.

Show()

Set user defined defaults and system variables.

SHOW/DEFAULTS		Show all I/O defaults
SHOW/PAR	**wksp**=*Workspace*	Show parameters set by Setpar()
SHOW/DATA	**wksp**=*Workspace*	Emulate GENIE-V2 show data command
SHOW/CONST		Show all the constants in Open GENIE
SHOW/PROC		Show all the procedures in Open GENIE
SHOW/TYPE		Show all the types in Open GENIE
SHOW/VAR		Show all the variables in Open GENIE
[/SYS]		Show system variables as well

example:

```
# Check the I/O defaults
>> Show/defaults
Current default disk = EVS$DISK0:
Current default directory = [EVSDATA]
Current default instrument = EVS
Current default extension = .RAW
Current default input = EVS$DISK0:[EVSDATA]EVS00456.RAW
Current default output =
```

Note: The current default input file can be altered by Assign() or Set/File/Input

Show/Defaults

Shows all the current I/O defaults. Note that the GENIE-V2 disk, directory, instrument and extension defaults will only change the default input file when the Assign() command is used. The *Get()* and *S()* commands both use whatever the default input file specifies.

Show/Par

Prints the values set in a workspace by the *Setpar()* command. This is only kept for compatibility with GENIE-V2, in Open GENIE all values can be printed directly out of the workspace (eg printn work.twotheta).

Parameters:

Wksp (Workspace)

The workspace being inspected.

Show/Data

Emulates the GENIE-V2 "show data" command by printing *all* the data from the workspace to the terminal screen. If a long workspace is printed to the screen by mistake Control-C can be used to stop the printout! If similar data is required in a file, use the Open GENIE *Asciifile/Writefree* or *Put/Ascii* commands.

Parameters:

Wksp (Workspace)

The workspace being inspected.

Show/Const

Display all the constants currently defined within the Open GENIE session (including pre-defined constants such as colours).

Parameters:

/Sys

Show system constants as well (all these begin with an "_")

Show/Proc

Display all the procedures currently defined within the Open GENIE session (including their help text)

Parameters:

/Sys

Show system procedures as well (all these begin with an "_")

Show/Type

Display all the workspace types currently defined within the Open GENIE session.

Parameters:

/Sys

Show system types as well (all these begin with an "_")

Show/Var

Display all the variables currently defined within the Open GENIE session.

Parameters:

/Sys

Show system variables as well (all these begin with an "_")

Chapter 4

I/O Commands

This section details all the commands used in Open GENIE for input and output of data. The commands can be grouped as show below.

- Storage and retrieval in one of the *supported data file formats*

Filetype()	Returns information about the type of a data file
Get()	Reads data into GENIE from a file
List()	Lists the contents of a data file
Nblocks()	Returns the number of readable blocks in a file
Put()	Outputs Open GENIE data into a file

- Free format ASCII input and output of GENIE data

Asciifile()	Read or write any format of ASCII data

- Data input from a user written program in FORTRAN or C.

Module()	Call user written code as part of Open GENIE

- Printing information to and reading information from the user.

Print()	Print information to the user
Inquire()	Prompt the user for terminal input and read back an input variable
Read_terminal()	Read a string back from the input terminal into a variable

Open GENIE maintains a default "Current input file" and "Current output file". For input and output the *Set/File/Input* and *Set/File/Output* may be used explicitly to set the files to be used as a default for later commands. For input files, defaults may also be set up in a similar fashion to those used in GENIE-V2 with separate defaults for disk, directory, instrument name and file extension using the appropriate *Set()* commands. The current defaults may be viewed with the *Show/Defaults* command.

One of the major strengths of Open GENIE is the flexibility available in accessing data. For all the supported file formats it is possible to use the lower level *Get()* and *Put()* commands to access named items of data directly.

The commands reading whole spectra (See the section on *GENIE-V2 emulation*) are built using these lower level commands. Whole spectra may also be saved/retrieved using *Put()* and *Get()* by treating workspaces as single Open GENIE variables.

I/O Command Reference

Filetype()

Finds out the type of a data file (if recognized by Open GENIE).

FILETYPE [**file**=*String*] Return a string giving the data file type.

example:

```
# Check the type of a file
printn Filetype("HRP00389.RAW")
raw
```

Note: "file" defaults to the type of the default input file if there is one.

Filetype

The filetype command makes the process Open GENIE uses to determine file types available to the user. By using the Filetype() command it is possible to write Open GENIE procedures which, from the outside appear independent of input file type, but internally take account of different data formats.

Parameters:

File (String) [default = currently selected input file (via the *Set()* command)]

File which is being tested.

RESULT = (String)

One of the following strings depending on the type of the file.

"<unknown>"	Filetype unknown to Open GENIE
"raw"	ISIS original format raw data file
"G2 intermediate"	GENIE-V2 intermediate file
"G3 intermediate"	Open GENIE intermediate file
"CRPT"	ISIS Current Run Parameter Table
"HDF"	HDF format data file
"ASCII"	Plain text (ASCII) file
"<directory>"	A directory file on the host system
"G3 ASCII"	Open GENIE ASCII dump formatted file

this is only a minimum set of file types. If you have a file reading routine please mail *genie@isise.rl.ac.uk* and we will investigate adding your data file as one recognized by Open GENIE.

Get()

Read binary spectrum data into GENIE from any valid data source (e.g. ISIS Raw files, DAE, intermediate files etc.)

GET()	**item**=*String* [**file**=*String*]	Read binary data by name
GET()	**item**=*Integer* [**file**=*String*]	Read binary data by number
GET()	**item**=*Interval* [**file**=*String*]	Read binary data by interval
GET()	**item**=*IntegerArray* [**file**=*String*]	Read binary data by range
/ARRAY		Read data back as a workspace array instead of a two-d workspace

example:

```
# print the user name
# then read in all the spectra in the file,
# finally read every third spectrum from a file
>> printn get("USER", "Isisfile.raw")
Professor G. Offar
>> 2d = get( 1 : get("NSP1") )
>> spin_up = get( 1:50@3 , "CSP4589.RAW" )
```

Note: Use the List() command to find out what data can be read from the file.

Get()

This function is a fully generic way of accessing binary data from Open GENIE from any valid data format. Open GENIE is always able to detect automatically what format the data source is in so there is no need to do anything differently when accessing an intermediate file, an ISIS raw file or even the Data Acquisition Electronics (DAE).

It should be remembered however that even if Open GENIE is able to detect the type of the data source, it cannot guarantee what items are available in the file, for example, asking for user information will work fine from an ISIS raw file, but not from the DAE! It is wise to check any unknown file with the *List()* command to see what information is available.

Parameters:

/Array

This optional qualifier specifies that when getting multiple spectra, the routine should return a workspace array of single spectrum workspaces, rather than a single multi-dimensional workspace.

Item (String or Integer or Range)

This specifies the name (String) or index (Integer) of the primary data item being requested from the data file. The assumption here is that all data items in the file are either named explicitly or numbered. For example, "NSP1" is the name used in ISIS RAW files to get the total number of spectra in the first time regime. In the same file, specifying the integer 5 would get the fifth spectrum within the file. To find the names of items which may be specified for a particular file, the *List()* command may be used.

Open GENIE also supports unique data types called *Interval* and *Range,* a Range or Interval can be used to specify multiple indices or a range of indices respectively to access multi-dimensional data. For more information on specifying intervals, please see a description of the *Interval syntax*.

For the most general ability to specify a group of arbitrary spectra an Integer array of spectrum identifiers can be given as the the "Item" parameter.

File (String) [default = Current input file]

This parameter gives the source from which Open GENIE is reading the data specified. Normally this will be a file on disk or the data acquisition electronics but in future could also be a network address of a data source at a remote site.

If this parameter is not given, a file specified previously using the *Set/File/Input* command is used.

RESULT = (Any Open GENIE type)

Because the Get() command can return any sort of data from a file, the result of this function will the the closest available Open GENIE data type to the item being read. For example, if a string is being read from a file, the return type will be a *String*, alternatively, if a spectrum is being read, the return type will normally be an Open GENIE workspace.

List()

Lists the items available within any datafile with a format Open GENIE understands.

LIST	**file**=*String*	Catalogues the contents of a data file
LIST/IN		Lists the contents of the default input file
LIST/OUT		Lists the contents of the default output file
[/FULL]		List the file contents with full details

example:

```
>> List "dat.out"
Genie intermediate file version 1.0
Block    Label          Type
1       vanadium        GXWorkspace
2       Run 1   GXWorkspace
3       Run 2   GXWorkspace
```

Note: To see comments use the /Full qualifier.

List

This command gives a means of listing out pertinent information about the contents of a data file. The block number and/or label string identify how to request the data when using the *Get()* command (either the block number or the label string can be given as the "Item" parameter).

Parameters:

/Full

Give a full listing of information, eg the full listing for the example above.

```
t/full "dat.out"
termediate file version 1.0
l, Label = vanadium, Type = GXWorkspace
 Tue 01 Jul at 14:02:36 BST by Chris <cmt@spudtek>
ıt: Calibrating run

2, Label = Run 1, Type = GXWorkspace
 Tue 01 Jul at 14:03:36 BST by Chris <cmt@spudtek>
ıt: 15/3/98 - Nobel data (first cut)

3, Label = Run 2, Type = GXWorkspace
 Tue 01 Jul at 14:16:58 BST by Chris <cmt@spudtek>
ıt: 15/3/98 - Nobel data (what I wrote up)
```

File (String)

The input file to be listed

Parname2 (Type2)

As above

RESULT = (Workspace)

Returns the result of the command as a workspace of strings and string arrays so that the information can be used in a program. The example below shows the same listing returned in a workspace.

```
ıtn list:full("dat.out")
```

ıce []

= "Genie intermediate file version 1.0"
Chris <cmt@spudtek> Chris <cmt@spudtek> Chris <cmt@spudtek>] Array(3)
GXWorkspace GXWorkspace GXWorkspace] Array(3)
[vanadium Run 1 Run 2] Array(3)
ıt = [Calib. run 15/3/98 - Nobel dat (first cut) 15/3/98 - Nobel dat (written up)] Array(3)
Tue 01 Jul at 14:02:36 BST Tue 01 Jul at 14:03:36 BST Tue 01 Jul at 14:16:58 BST] Array(3)

List/In

As for the List() but lists the contents of the default *input* file as specified with one or more of the *Set()* commands.

List/Out

As for the List() but lists the contents of the default *output* file as specified with one or more of the *Set()* commands.

Nblocks()

Returns the number of readable blocks in an Open GENIE recognized file.

NBLOCKS()	[**file**=*String*]	Return number of blocks in an intermediate file, or number of spectra in a raw file

example:

```
# Check the number of spectra in a raw
# file then do a multiplot
>> Set/File/Input "hrp00345.raw"
>> n = nblocks()
>> multiplot 1:(n)
```

Note: For an ISIS raw file the above use of Nblocks() is equivalent to Get("NSP1").

Nblocks

Returns the number of blocks the Get() command would be able to read when reading by block number. Put in another way for a file "d.dat" the command Get(Nblocks("d.dat"), "d.dat") will access the last block in the file.

Parameters:

File (String) [default = default input file]

Any file of a type which can be recognised by Open GENIE.

RESULT = (Integer)

Returns the number of the highest block in the file.

Put()

Outputs Open GENIE data to a known file type.

PUT	**gv**=*Any* [**label**=*String*] [**comment**=*String*] [**file**=*String*]	Appends the specified variable to the end of the data file, by default in Open GENIE intermediate file format..
[/NEW]		Create a new data file or overwrite existing file.
[/ASCII]		Write the data file in formatted ASCII.
[/HDF]		Write in HDF format.

example:

```
# Copy a whole multidimensional spectrum from a raw
# file into a single item in an intermediate file
>> Set/file/Input "hrp00273.raw"
>> Set/file/Output "example.in3"
>> put get(1:20) label="bigspec" comment="demo"
```

Put

The Put() command provides a mechanism for saving Open GENIE variables of arbitrary complexity in an intermediate binary file. Items are written sequentially and appended to any already existing file - if the file does not exist or needs re-creating it can be created or overwritten by specifying the "/New" qualifier.

Variables being put into a file may optionally be tagged with a label string. This makes direct access using the *Get()* command easier as the label can be used by *Get()* instead of the block number. Comments may also be added to annotate individual data items in the file to enable identification later with the *List()* command.

Parameters:

/New

Used to force file creation when no file exists or when it is necessary to overwrite and existing file rather than appending a data item to it (the default).

/ASCII

Specifies that data will be written in Open GENIE ASCII file format. This is an alternative to using the normal Open GENIE intermediate file format but it actually behaves in the same way both on input and output. It allows ASCII output of all types of data including workspaces.

/HDF

Identical behaviour, but specifies that data will be written in HDF format rather than in Open GENIE intermediate file format.

Gv (Any)

The Open GENIE variable to be written to the data file. Any genie variable of any complexity can be written directly into an intermediate file and retrieved with the *Get()* command.

Asciifile()

Read or write any format of ASCII data.

ASCIIFILE:OPEN()	**file**=*String* [**comment**=*String*] [**delimiter**=*String*]	Open an ASCII file
ASCIIFILE/CLOSE	**handle**=*AsciiFile*	Close an open file
ASCIIFILE:DATA() [:RESET]	**handle**=*AsciiFile*	Return all data currently accumulated in a workspace.
ASCIIFILE:LINES()	**handle**=*AsciiFile*	Return the number of data lines left to read
ASCIIFILE/READFIXED	**handle**=*AsciiFile* [**fields**=*String*] **format**=*String* [**count**=*Integer*]	Read fixed format data
ASCIIFILE/READFREE	**handle**=*AsciiFile* [**fields**=*String*] [**separator**=*String*] [**count**=*Integer*]	Read free format data
ASCIIFILE/SKIP	**handle**=*AsciiFile* [**nlines**=*Integer*]	Skip lines on input
ASCIIFILE/WRITEFREE	**handle**=*AsciiFile* **comment**=*String* **separator**=*String* **gv1**=*Any* [**gv2**=*Any*] [**gv3**=*Any*] [**gv4**=*Any*]	Write free format data
ASCIIFILE/REWIND	**handle**=*AsciiFile*	Close and re-open a file at the start

example:

```
# Read three data arrays (in columns separated by whitespace)
# and assign to arrays in fields X, Y, and E in the new workspace.
Handle = Asciifile:Open("myilldata.dat")
wk1 = Asciifile:Readfree(Handle, "X,Y,E", $WHITESPACE, 8)
printin wk1
  Workspace []
  (
    x = [1.0 1.1 1.2 1.3 1.4 ...] Array(8 )
    y = [2.1 2.1 2.2 2.0 2.0 ...] Array(8 )
    e = [0.0 0.1 0.2 0.0 0.1 ...] Array(8 )
  )
```

Asciifile/Open

Open a file and return a handle for use with other Asciifile() commands. This command must be called before any other operations can be carried out

Parameters:

File (String)

Name of the ASCII data file to be opened, or if it does not exist, to be created.

Comment (String) [default = ""]

When an ASCII file is opened by Open GENIE for reading, it is possible to specify a single character which is being used in the input file as a comment character. The result of this will be to effectively ignore all lines beginning with this character up to the last instance of "Delimiter" (the Asciifile:lines() command will report only the number of non-commented lines).

Delimiter (String) [default = "\n" (new line)]

A single character that indicates the end of a line of input and this is configurable because different systems may terminate lines in different ways. Normally this will only be necessary if the ASCII file has come from a different operating system (for example, files from a Macintosh may require "\r" as the delimiter).

RESULT = (Asciifile)

The result of this command is a file handle variable which must be stored and passed as a parameter to any future operations on the file.

Asciifile/Close

Close the file, flushing any buffers to disk.

Parameters:

Handle (Asciifile)

Specifies the file to close. Once a file has been closed, the "Handle" value becomes undefined and should not be used.

Asciifile:Data()

It is possible to read data in two ways with the Asciifile() command. The example at the beginning of this command description shows how data may be read into a workspace in a single go with the /Readfree qualifier. An alternative way of reading the same data is shown below using the Asciifile:Data() command.

```
Asciifile/Readfree Handle "X1,Y1,E1" $WHITESPACE 4
Asciifile/Readfree Handle "X2,Y2,E2" $WHITESPACE 4
wk1 = Asciifile:data(Handle)
printn wk1
  Workspace []
  (
    x1 = [1.0 1.1 1.2 1.3 ] Array(4 )
    y1 = [2.1 2.1 2.2 2.0 ] Array(4 )
    e1 = [0.0 0.1 0.2 0.0 ] Array(4 )
    x2 = [1.4 1.5 1.6 1.7 ] Array(4 )
    y2 = [2.0 2.0 2.0 2.0 ] Array(4 )
    e2 = [0.1 0.1 0.2 0.0 ] Array(4 )
  )
```

Using this method several different items of data may be accumulated into the workspace before it is returned.

Parameters:

/Reset

Clears the workspace which was previously being accumulated of all fields and gives a fresh start. Note that this does not reset any file pointers; to reset a file on reading use the /Rewind option.

Handle (Asciifile)

Specifies the file with which all the data is associated. A separate workspace may be accumulated for every open file handle.

RESULT = (Workspace)

Returns the accumulated workspace of data items.

Asciifile:Lines()

Returns the number of data lines (i.e. non comment lines) in the file left to read. To return the total number of data lines in a file, the Asciifile:lines() command must be called before any lines of data are read in.

Parameters:

Handle (Asciifile)

Specifies the file to count the remaining lines in.

RESULT = (Integer)

Count of remaining lines to read.

Asciifile/Readfixed

Reads lines from an ASCII formatted file in "Fixed" format. This is similar to FORTRAN fixed format reads where data is identified exactly by position and length and there are not necessarily any spaces separating the data being read in. For example, the line below reads 50 lines out of a fixed format file consisting of two columns of real numbers in fields of width 10 each. The first character on each line is a space to be skipped before the numbers start.

```
wk = Asciifile:Readfixed(Handle, "x,y", " %10f%10f", 50)
```

Parameters:

Handle (Asciifile)

Specifies the file to read data from. The current position in the file is also remembered with the file handle.

Fields (String) [default = "field01, field02, field03, field04, field05, field06"]

Gives a list of names, separated by commas, which will be assigned to the fields of the resultant data workspace. The names will be associated, in order, with the columns of data as they are read from the input file.

If the empty string ("") is passed and the data being read is either a single value or a single column of data, a single variable or array will be returned instead of a whole workspace.

Format (String)

This is a C style sscanf expression (similar in a FORTRAN FORMAT statement in meaning). The "%" character is used to indicate the position and specific format for each data item in the line. Some examples of common formatting possibilities are given below.

%6f	Matches a floating point number of total width 6
%s	Matches a string of non-whitespace characters
%10c	Matches ten characters, including white space
%d	Matches an integer
%4d	Matches an integer of width 4
%*f	Skips a float in the line
%%	Matches a % character
" "	(space) Skips one or more spaces

Other sscanf formatting conventions may be used if they are supported on the host operating system.

Count (Integer) [default = 1]

Number of lines to read in one operation. This effectively defines the size of the data arrays being returned for each column of data. To read to the end of the file specify -1.

RESULT = (Workspace, array or single value)

The result of an ASCII read operation depends on the amount and format of the data read, normally a workspace is returned, but for a single column of data or a single value in the file, an array or single item can be returned respectively (see the "Fields" parameter description).

Asciifile/Readfree

Reads lines from an ASCII formatted file in "Free" format. This is similar to FORTRAN list directed read statements where data is separated, usually by spaces or commas. For example, the line below reads back a workspace with two array fields X and Y which are read from two columns in the input file where the columns are delimited by one or more "!" characters.

wk = Asciifile:Readfree(Handle, "x,y", "[!]+", 40)

Parameters:

Handle (Asciifile)

Specifies the file to read data from. The current position in the file is also remembered with the file handle.

Fields (String) [default = "field01, field02, field03, field04, field05, field06"]

Gives a list of names, separated by commas, which will be assigned to the fields of the resultant data workspace. The names will be associated, in order, with the columns of data as they are read from the input file.

If the empty string ("") is passed and the data being read is either a single value or a single column of data, a single variable or array will be returned instead of a whole workspace.

Separator (String) [default = $WHITESPACE (one or more tabs or spaces)]

A single character or regular expression specifying what to expect between each item of data on a line. The definition of the default $WHITESPACE is "[\t]+" meaning one or more space or tab characters. This default will be suitable for many cases but for example, an expression like "," will delimit comma separated data. For more details of how to construct regular expressions see *regular expressions* in the Appendices to this manual.

Count (Integer) [default = 1]

Number of lines to read in one operation. This effectively defines the size of the data arrays being returned for each column of data.

RESULT = (Workspace, array or single value)

The result of an ASCII read operation depends on the amount and format of the data read, normally a workspace is returned, but for a single column of data or a single value in the file, an array or single item can be returned respectively (see the "Fields" parameter description).

Asciifile/Skip

This skips "Nlines" of input data (i.e. non comment) lines in an input file.

Parameters:

Handle (Asciifile)

Specifies the file in which to skip the lines.

Nlines (Integer) [default = skip 1 line]

Number on lines in the input file to skip.

Asciifile/Writefree

Writes Open GENIE variables into up to four free formatted columns of ASCII data. Data cannot be written to an existing file already opened for reading.

Handle (Asciifile)

Specifies the opened file to write data to.

Comment (String) [default = ""]

Optional user comment which will be preceded by the comment character specified with the Asciifile/Open command. If a comment character was specified for the file and nothing is specified for this parameter, Open GENIE will add its own default header.

Separator (String) [default = " "]

Specifies the default character(s) to separate columns of output data with.

Gv1-Gv4 (Any valid Open GENIE type except workspaces)

Up to four separate GENIE data items to write out at one time. The values are written out in up to four columns. The longest data item of the four determines the length of the data added to the file. The shorter items are repeated to fill their columns.

Module()

Manage the dynamic loading and execution of user written C or FORTRAN code.

MODULE/COMPILE [/C]	**file**=*String* [**comp_flags**=*String*] [**link_flags**=*String*] [**symbols**=*String*]	Compile a user written module
MODULE/LOAD	**file**=*String* [**desc**=*String*]	(Re-) load the routine into memory
MODULE/EXECUTE [/C]	**func**=*String* **pars**=*Workspace*	Run a loaded module
MODULE/LIST		Show loaded modules

example:

```
# Compile a convolution module, load then run it on a 2-D workspace
>> Module/Compile "conv.for" comp_flags="/check=all" symbols="conv_sub"
>> Module/Load "conv.so"
>> w1 = get(1:30)
>> w2 = Module:Execute("conv_sub", w1)
```

Note: A compiled module only needs loading once per Open GENIE session.

Module/Compile

Open GENIE looks after the details of compiling a dynamically loadable program module whatever system it is running on. This greatly simplifies what on some systems is quite a complex process. The end result of the compilation of either a C or FORTRAN program is a dynamically loadable library which can contain one or more subroutines callable by Open GENIE once the module has been loaded.

The Compiled module may be written in FORTRAN or C and needs to adhere to a simple set of conventions to ensure its correct communication with the running Open GENIE. The structure of a module and the facilities available to it are described in the *FORTRAN Module Interface* and the *C Module Interface* sections of this manual.

Parameters:

/C

Invoke the C compiler instead of the FORTRAN compiler (the default).

File (String)

The name of the FORTRAN or C source file (including the .f, .c, or .for part of the filename). The compilation will produce a shareable library of the same file name but with the file extension of ".so" for the dynamic library. Once created, this can be kept and used with the Module/Load command.

Comp_flags (String) [default = ""]

This string takes any compiler flags or switches which may be needed by your program. Open GENIE already will have already added anything that is needed for compiling modules.

Link_flags (String) [default = ""]

This string takes any linking flags or switches which may be needed to link the compiled code. Open GENIE will have already added all the flags needed to build a shareable library.

Symbols (String) [default = ""]

This is a comma separated list of the subroutine or C function names which you want open GENIE to be able to call from your module when you run a loaded module. This parameter supplies all the names you can use when specifying the "func" parameter in the Module/Execute command.

Note that if you do not list a name here, on most operating systems, the Module/Execute command will not be able to find your subroutine in the dynamic library and you will get an error.

Module/Load

The Module/Load command takes an existing compiled shareable module library and activates it into Open GENIE so that the routines in the module may be called. This only needs to be done once for any module file in an Open GENIE session. From then on it is possible to call routines within the module using the Module/Execute command. It is possible to check which modules are loaded with the Module/List command.

Parameters:

File (String)

The filename of the shareable library module to be loaded (it will always be a file ending in ".so"). Open GENIE searches for files in the current directory, its own area and any other directories specified by the GENIE_USER_LIBRARY_DIR environment variable/logical name.

Desc (String) [default = "<no description>"]

An optional description string. This is useful when several different modules are being used to identify their function. The description is show when using the Module/List command.

Module/Execute

This command runs a FORTRAN subroutine or C function from within a user written module. Open GENIE's advanced data access mechanism means that complex data can easily be transferred to and from a user written program without complex programming on the part of the user. The Module/Execute command can be called either as a function (x = Module:Execute(...)) or as a keyword command (Module/Execute ...). When called as a function a workspace is returned which may contain complex multi-dimensional data and/or simple strings and individual parameters.

Parameters:

/C

Invoke a C program instead of a FORTRAN program (the default).

Func (String)

The name of the subroutine or C function in the compiled module to call.

Pars (Workspace)

A workspace containing all the information for the external routine to process. This may be a workspace assembled just to package a few parameters and a data array needed by the external routine or it can be a complete n-dimensional workspace as read in by Open GENIE from data file (see the example above). See the sections on *C Module Interface* or *FORTRAN Module* Interface for details about the method of communicating parameters with the shareable library.

RESULT = (Workspace)

Contains the workspace given in the "Pars" parameter but as modified by the external subroutine.

Module/List

Lists all the currently loaded user written modules.

Print()

Print to the users terminal (no newline).

PRINT [**p1-p20**=*Any*] Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> print "Value of variable $PI=" $PI
Value of variable $PI=3.14159265358979>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Print(), Printn(), Printi(), Printin(), Printe(), Printen(), Printd(), Printdn(),

The generic print command takes up to 20 parameters, these are converted into a text form suitable for display on an interactive terminal and then displayed to the terminal. Depending on the suffix of the print command the text will be sent via different output streams and will be coloured differently using ANSI escape code sequences. If the command ends with an "n" a newline will be added automatically after the text has been printed. If the print conversion is required (i.e. from variable to string) but without writing to the terminal, it is possible to use the *As_string()* function.

There are four possible output streams with the following attributes:

Stream type	**Print Command**	**Characteristics**
Normal output	Print() or *Printn()*	Normal output in (typically) black text
Informational output	*Printi()* or *Printin()*	Blue coloured, can be switched on and off (see *Toggle/Info*)
Error output	*Printe()* or *Printen()*	Red coloured, includes line number and procedure where error occurred
Debugging output	*Printd()* or *Printdn()*	Mauve coloured, separates debugging output from other types of output (not usually visible during an ordinary user session).

Parameters:

P1-P20 (Any)

Variables to be printed. The strings produced by each variable are concatenated by the print command without adding any white space to allow maximum control over formatting.

Printn()

Print to the users terminal with a terminating newline.

PRINTN [**p1-p20**=*Any*] Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printn "Value of variable $PI=" $PI
Value of variable $PI=3.14159265358979
>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printn()

See the generic *Print()* command for a full description.

Printi()

Print an informational message to the users terminal (no newline).

PRINTI	[**p1-p20**=*Any*]	Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printi "Value of variable $PI=" $PI
```
Value of variable $PI=3.14159265358979>>

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printi()

See the generic *Print()* command for a full description.

Printin()

Print an informational message to the users terminal with a terminating newline.

PRINTIN	[**p1-p20**=*Any*]	Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printin "Value of variable $PI=" $PI
Value of variable $PI=3.14159265358979
>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printin()

See the generic *Print()* command for a full description.

Printe()

Print an error message to the users terminal (no newline).

PRINTE	[**p1-p20**=*Any*]	Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printe "Value of variable $PI=" $PI
0/(none)/Value of variable $PI=3.14159265358979>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printe()

See the generic *Print()* command for a full description.

Printen()

Print an error message to the users terminal with a terminating newline.

PRINTEN [**p1-p20**=*Any*] Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printen "Value of variable $PI=" $PI
0/(none)/Value of variable $PI=3.14159265358979
>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printen()

See the generic *Print()* command for a full description.

Printd()

Print an debugging message to the users terminal (no newline).

PRINTD [**p1-p20**=*Any*] Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printd "Value of variable $PI=" $PI
Value of variable $PI=3.14159265358979>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printd()

See the generic *Print()* command for a full description.

Printdn()

Print a debugging message to the users terminal with a terminating newline.

PRINTDN [**p1-p20**=*Any*] Converts all parameters for printing to the terminal screen

example:

```
# print out a variable with some text
>> printdn "Value of variable $PI=" $PI
Value of variable $PI=3.14159265358979
>>
```

Note: See also the other print commands in the group ("print" - "n, i, in, e, en, d, dn" e.g. "printdn")

Printdn()

See the generic *Print()* command for a full description.

Inquire()

Prompts the user for terminal input and reads back an input variable.

INQUIR E [**aprompt**=*String*]   Inquire information from the user interactively

example:

```
# Read in a lower limit from the user
>> llimit = inquire("Enter lower limit")
Enter lower limit: 45.678
>> printn llimit
45.6779999999999
>> printn Is_a(llimit, "Real")
$TRUE
```

Note: To read a string without any conversions or prompting use Read_terminal()

Inquire

This command reads input from the users terminal. It also supplies a prompt string to obtain input from the user. The Inquire() command attempts to read the input variable back into the correct type of Open GENIE internal variable. Currently, this is only the case for single Integer or Real values. Any unrecognized type is returned as a string.

Parameters:

Aprompt (String) [default = " :"]

A string which tells the user what information is required, a ":" is appended to the prompt.

RESULT = (Any)

Either a string or the value of the variable as a Real or Integer. If no input is given "" is returned.

Read_terminal()

Read a string back from the input terminal into a variable

READ_TERMINAL()   Return a string typed in at the terminal

example:

```
# Read a string
>> printn "(" read_terminal() ")"
abcd efg
(abcd efg)
>>
```

Note: To read numbers as well as strings you may wish to use the Inquire() command.

Read_terminal()

This command reads a string (terminated by a carriage return) from the terminal. A numeric string can be converted by using *As_variable()* on the string returned by Read_terminal(). This is the way the *Inquire()* command is written.

Parameters:

RESULT = (String)

The string read from the terminal, not including the return character which terminated the input.

Chapter 5

Syntax

Open GENIE commands fall into two obvious groupings, those used primarily in interactive mode and which are complete within a single line, and those used mainly to build procedures and whose effect spans several lines. The former are termed "single-line" and the latter "multi-line" commands. Single line commands also fall into a further division between commands which are expressed in an algebraic or functional notation and commands which are accessed as a simple keyword type command.

- *Micro-syntax* (some preliminaries about Open GENIE syntax)
    - *Comments*
    - *Intervals*
    - *Line extending*
- *Single-line commands*
    - *Keyword Commands* (of the form **Display w**)
    - *Function Commands* (of the form **x = sin(y)**)
- *Multi-line commands*
    - *IF - ELSE - ENDIF*
    - *CASE - ENDCASE*
    - *LOOP - ENDLOOP*
    - *PROCEDURE - ENDPROCEDURE*

Micro-syntax

This section describes a few minor but important parts of the syntax of Open GENIE command language.

Comments

Comments in Open GENIE may be placed anywhere on a line as long as they are preceded by a "#". The rest of the line is then treated as a comment and ignored. There is no mechanism for commenting out multiple lines in Open GENIE without putting a "#" at the beginning of each line, this avoids the risk of commenting out sections of code inadvertently.

Intervals (and Ranges)

Intervals are a means by which groups of indices can be specified within Open GENIE. The most common usage for intervals is in slicing parts out of arrays and in specifying groups of spectra. The basic syntax is

j **:** k [**@** m]   (for example 1:5@2)

where "j" is an integer start value, "k" is an integer end value and "m" is an optional step value. An interval such as 1:4 specifies all the numbers between 1 and 4 i.e. 1 2 3 4. If the step value is used, the interval selects only those numbers which fall in multiples of step from the starting value, so 2:12@3 will specify the numbers 2 5 8 11. If required, the values specified in an interval can be variables but it is necessary to surround each variable name in parentheses so instead of writing start:stop@step it is necessary to write (start):(stop)@(step).

Note that some spectrum manipulation functions in Open GENIE take a "Range" data type which is either an Integer array or an Interval. The Integer array form is more general than the interval syntax but the interval syntax is more convenient.

Examples

```
# reads every third spectrum into a multi-dimensional workspace
# starting from the first spectrum.
>> maxspec = get("NSP1")
>> a = get( 1:(maxspec)@3 )
# copy selected elements of the array "myarray" to make a new array called "slice"
>> slice = myarray(34:50)
# Create a Range in an integer array equivalent to
# the interval 4:20@2
>> my_range=Dimensions(9)
>> fill my_range 4 2
>> set/file "myfile.raw"
>> printn get(my_range) = get(4:20@2)
true
```

Line extending

Open GENIE uses similar rules to FORTRAN in that single line statements are assumed to terminate at the end of the physical line. Sometimes however, it is useful to be able to extend a single logical line over several physical lines. In Open GENIE the line extension character is an "&", it must be used at the end of the line being extended. All white space following the "&" is ignored. If necessary, strings can also be extended in an unbroken fashion. There is no limit to the number of continuation lines which may be created in this way.

Conversely, several logically separate single-lines may be placed on one physical line by using the ";" character as a separator. This is just provided as a convenience which can be used to make a program more readable.

Examples

```
>> long_string = "My very long string is being broken&
in two"
IF a = b; printn "yes, a = b"; ELSE; printn "no, a !=b"; ENDIF
```

Note that in the first example, any spaces before the start of the following line WILL get included in the string.

Single-line commands

The term "single-line" refers to a single logical line containing one keyword command or assignment.

Single-line commands consist of keyword commands and assignments. Any Open GENIE procedure may be invoked as a keyword command. Any Open GENIE procedure which returns a result can be invoked either as a keyword command (in which case the value is ignored) or as an expression using the function syntax.

In some cases it is useful to know in which way a procedure has been invoked

whilst within the procedure, for this the *Called_as_function()* command can be used.

Keyword commands

Open GENIE keyword commands start off with a keyword followed by a list of parameters and qualifiers. As a general rule, the keyword specifies the basic command, the qualifiers modify the action of the command and the parameters supply the values which the command operates on. An example of a keyword command is

```
>> Display/Line w1
>>
```

The *Display()* command is used to plot a histogram on the graphics screen. The */Line* qualifier modifies the command to do the plot as a polyline rather than a histogram, the parameter w1 is the workspace being displayed.

Specifying parameters

Parameters are generally specified by the position on the command line which they occupy. For example, with the *Display()* command the first parameter is the workspace to be displayed, the next two are the low and high X limits for the plot and the final two the low and high Y limits. An example is

```
>> Display w1 0 100 -35 70
>>
```

All the parameters except for the workspace are optional and where omitted a default value will apply. Unfortunately, identifying parameters by position alone is not sufficient. If the Y limits need specifying but the X limits should be defaulted it is necessary to tell Open GENIE that the parameters on the line are the Y limits and not X limits. This is done by giving each parameter a keyword which uniquely identifies it amongst other parameters on the line. The command with defaulted X values could then be specified

```
>> Display w1 ymin=-35 ymax=70
>>
```

The parameters to a command may be expressions or function invocations themselves so commands such as

```
>> Display w1-background 0 xmax/100.0 ymin=Min(w1.y) ymax=Max(w1.y)
>>
```

or

```
>> Display sqrt(w1)
>>
```

are perfectly legal. Occasionally it may be necessary to bracket an expression to make the meaning obvious.

Anonymous placeholders

Another mechanism to avoid the problem referred to above is to use anonymous placeholders. In Open GENIE the "_" character is used. The example given above could also be written.

```
>> Display w1 _ _ -35 70
>>
```

Although a little less readable it is quicker to type!

Qualifiers

Qualifiers in Open GENIE are defined as switches which in some way modify the action of a command. Taking the *Display()* command as an example, the default action is to display a spectrum as a histogram rather than a line plot. It wouldn't be sensible to change the name of the command completely to get a line plot because in all other senses the command is the same. Qualifiers allow this modification to be achieved in an obvious way. The command to display a line plot is *Display/Line*. Further qualifiers may be added to the command to modify the action further. For example *Display/Errors* would display the spectrum as a line plot with error bars. Sometimes a qualifier will require some modification to the parameter list, this is permissible but qualifiers which require this are mutually exclusive.

Abbreviations

Many keywords can be abbreviated, as long as there exists an acceptable unique abbreviation by using the *Alias()* command. The native commands are allocated abbreviations as they are defined in Open GENIE. This means that in general, user defined commands will have longer abbreviations than the native commands. Qualifiers may be abbreviated as much as is allowed by other qualifiers used with the command. Note, it is quite possible to obscure a command by abbreviating something else to it!

Assignments

Apart from keyword commands there is one other sort of single-line command, the assignment.

<destination-location> = <expression>

Assignments are made using the "=" symbol. The left hand side of an assignment specifies a destination variable or possibly one element of a compound variable. Any value already in <destination-location> is replaced by a copy of the value of the expression on the right hand side.

Expressions

Expressions can be used anywhere a value is required, for example, on the RHS of an assignment or in the parameter list of a function call. An expression is made up from one or more sub-expressions built from the parts listed below.

- A literal value. e.g. 5.6 or "Hello".
- A variable, or part of a variable. e.g. TwoTheta or w1.npts.
- A bracketed expression e.g. (x - 7).
- A function invocation e.g. Sin(Pi). See *Function commands*.
- An unary expression e.g. -value or NOT logAxes. The unary operators are " - + NOT" and are evaluated before any binary operators.
- A binary expression e.g. 4.5 + 8.0 or 5 <= 6. The binary operators are given below, the operators with the highest precedence first. Where two operators with the same precedence are used at the same bracketing level in an expression the leftmost sub-expression will be evaluated first.

  ^
  AND * / |
  OR + -
  &
  > < = >= <= !=

For details of how these operators are interpreted for different Open GENIE variable types and combinations of types (see *Storage*).

Function commands

The Open GENIE function syntax provides a method for calling Open GENIE procedures as functions. The function/procedure result can then be assigned to a variable. Commands which return a value can be used interchangeably as statements or as value returning functions in an expression. For example

```
>> integral = Integrate( W1, 100.0, 200.0 )
>>
```

or

```
>> Integrate w1 100.0 200.0
>>
```

are both valid single-line commands.

When the *Integrate()* command is used here as a function, the assignment places the result of the integration in the user defined variable "integral". When used with the keyword syntax, the result of the integration is lost.

Specifying parameters

The parameters to an Open GENIE command invoked as a function are supplied in a similar format to that used in the keyword command syntax. Default values for parameters will be supplied if the parameter is omitted. Parameter names may also be used as in a keyword command if desired. For example

```
>> V6 = Integrate( w1, High=200.0 )
>>
```

An important point to note is that Open GENIE commands being invoked as functions must always specify a parameter list, even if it is empty e.g. " look_no_parameters()". This allows Open GENIE to distinguish between function commands and ordinary variables.

Qualifiers

Qualifiers may be specified in the functional notation by placing a colon separated list of qualifiers after the function name and before the parentheses delimiting the parameter list. For example,

```
>> W2 = Asciifile:Open( "My_ascii_file.txt" )
>>
```

Here the equivalent keyword command would be *Asciifile/Open* but there would be no way to return the workspace without the assignment.

Abbreviations

Abbreviations are allowed in function commands as with keyword commands and the same rules apply.

Multi-line commands

These statements provide all the program flow control in Open GENIE. The keywords beginning the constructs may not be abbreviated and must be written in *uppercase* to distinguish them from ordinary commands. Open GENIE procedures should avoid names which clash with these keywords.

IF statement

The IF statement provides for a choice between two actions. A boolean condition is tested and the clause following the IF is executed if the condition is true. If the condition is false the clause following the ELSE statement is executed. In the absence of the ELSE part, no action is taken on a false condition.

```
  IF <Condition1>
      <Statement1>
[ ELSEIF <Condition2>
      <Statement2> ]
[ ELSE
      <Statement3> ]
  ENDIF
```

In the example <Condition1> and <Condition2> are expressions evaluating to true or false. If <Condition1> evaluates to true <Statement1> is executed. If <Condition1> and <Condition2> are false then <Statement3> is executed. <Statement1> and <Statement2> may consist of several Single-line or Multi-line Open GENIE commands. In multiply nested IF statements, one ENDIF is required for each opening IF statement.

Examples

```
# Initialise and zero an array that may or may not exist yet.
IF NOT Defined(A4)
    A4 = Dimensions(1000)   # Create a 1000 element long array
    Fill A4 0.0             # initialises each element to 0.0
ELSE
    A4=A4*0.0               # Shorthand way of zeroing array
ENDIF
# Multi line commands can be reduced to a
# single physical line by using semicolons
IF a >= 0; Printn "a=" sqrt(a); ELSE; Printn "complex"; ENDIF
```

CASE statement

The CASE statement allows a selection of alternative actions to be specified depending on the value of an expression. This is similar in operation to the FORTRAN computed GOTO statement.

```
    CASE <expression>
        IS <value1> [ TO <Range1> ]
            <Statement1>
        ...
        IS <valueN> [ TO <RangeN> ]
            <StatementN>
        OTHERWISE
            <StatementN+1>
    ENDCASE
```

The expression after the CASE statement is evaluated and then compared with the expression or optional range after each IS statement. The statement after the first matching IS statement is executed and all others are ignored. If no values or ranges match, the statement following the OTHERWISE statement is executed. In the absence of an OTHERWISE statement no action is taken.

Examples

```
#CASE My_mode()                        # User heck for batch mode operation
    IS "BATCH"
        Hardcopy devtype="PS"                     # Save postscript plots if in batch
    IS "INTERACTIVE"
        Device/Open devtype="XW"        # X-windows if interactive
    OTHERWISE
        Printn "Invalid mode - No plot device open"
ENDCASE
CASE energy                            # a real number
    IS 0.0
        Printn "No data"                # energy = 0.0
    IS 0.0 TO 500.0
        Myplot/positive                 # 0.0 < energy <= 500.0
    IS -500.0 TO 0.0
        Myplot/negative                 # -500 <= energy < 0.0
    OTHERWISE
        Printn "Energy out of range"
ENDCASE
```

LOOP statement

The LOOP statement provides a means of controlled loop execution in an Open GENIE program. The LOOP statement provides for conditional and/or counted loops.

```
    LOOP [ <variable> FROM <Start> TO <End> [ STEP <Step-size> ] ]
```

```
        <Statements>
      [ EXITIF <Condition> ]
        <Statements>
    ENDLOOP
```

Statements in the loop are executed until `<condition>` after the EXITIF statement becomes true, or until the optional counting variable has reached or exceeded the final value. If the loop has neither a counter nor an EXITIF statement it will continue to loop indefinitely.

Only one EXITIF statement may be directly contained in any loop. It must not be used inside any other form of control construct within the loop (doing this will generate a syntax error). In the case of nested loops, the exit is to the end of the loop directly containing the EXITIF statement.

For counted loops, the values of `<start>` and `<end>` can be real or integer values and they may be positive or negative. Any sensible combination of start and end values will work as long as `<step>` has the correct sign to progress the counter to (or past) the end value. If it does not, the loop is guaranteed not to execute any statements. If the STEP part is omitted the step interval defaults to one.

The counter for the loop does NOT need to be declared anywhere else. It only has a valid value within the body of the loop and will shadow the value of any other variable of the same name for the duration of the loop.

Examples

```
# print some squares
LOOP i FROM 1 TO 10 STEP 1
    Print i^2 " "                   # print squares
    IF i = 10; Printn; ENDIF        # print a newline
ENDLOOP
# something a little more complicated
cubic=0; square=0; y=0
LOOP i FROM -10.0 TO 10.0 STEP 0.01
    y = i
    cubic = y^3 - 5
    square = y^2
    EXITIF cubic > square           # stop at +ve intersection
ENDLOOP
Printn "cubic=" cubic ", square=" square " for y=" y
# Dimension an array of strings
position = 0
sr = Dimensions(10)
sr[8] = "Find me"           # and hide a string in it
sr[5] = "Decoy"
LOOP i FROM 1 TO Length(sr)
    position = i
    EXITIF sr[i] = "Find me"
ENDLOOP
Printn "Found at position " position
```

PROCEDURE statement

The PROCEDURE statement provides a method for writing Open GENIE programs. Most of Open GENIE is built using procedures which are loaded each time it is run, usually this is transparent to the user. By writing procedures it is possible to customise Open GENIE for a specific task or for a specific group of data manipulation and display operations.

```
  [ FORWARD <name> ]

    PROCEDURE <name>
  [ QUALIFIERS /<Qname1> /<Qname2> ... /<QnameN> ]
  [ PARAMETERS <parname1>=<partype1> ... ]
```

```
[ RESULT <resname>=<default-value> ]

[ GLOBAL <gloabalvar1> [ = <initial-value> ] ... ]
[ LOCAL <localvar1> [ = <initial-value> ] ... ]

    <Statements>
  [ RETURN ]

ENDPROCEDURE
```

The PROCEDURE statement defines a command to be known to the whole of Open GENIE. Once defined, the procedure may be invoked at the " >>" prompt using either the keyword or function syntax (see *Single-line commands*). This section describes the individual parts of a procedure definition as shown above.

Occasionally it is necessary to use a procedure before it can be defined, this is likely if any routines use recursion, or when groups of commands stored in multiple GCL command files are interdependent. In this circumstance, a procedure can be forward declared with the FORWARD statement, generally this will not be needed if files are loaded so that procedures are declared before they are used.

`<Statements>` consists of any number of single-line or multi-line Open GENIE commands just as would be typed at the terminal in an interactive session. An optional RETURN statement can also be used to allow the procedure to return to the calling procedure before control reaches the ENDPROCEDURE line.

The major difference from typing commands interactively is that all variables used in the procedure must be defined before they can be used, this is the purpose of the five optional declaration statements at the start of each procedure.

These declarations are used in two groups QPR (QUALIFIERS, PARAMETERS and RESULT) and GL (GLOBAL and LOCAL). Within these groups the order of the declarations is unimportant and all (except for the RESULT declaration --- for obvious reasons) can be used as often as necessary. A valid set of declarations could be ordered RQQPPPLLGLG. The only major restriction is that all members of the QPR group must come before any of the GL group, if not, a syntax error will result. The format and purpose of the individual declarations is given below.

- QUALIFIERS /<Qname> ...

  The QUALIFIERS statement declares any qualifiers that the procedure should respond to. In the procedure, qualifiers can be accessed by treating them as variables with a truth value i.e. true if the qualifier was specified on the command line, false otherwise. Qualifier variables can be tested with IF statements.

  Examples

```
.ine /Markers

-s
/ithout markers"
```

- PARAMETERS <Parname>=<Partype> ...

  This PARAMETERS statement declares all the parameters which can be passed to the procedure. <Parname> specifies the name by which the parameter will be known within the procedure and also the name by which it can be specified explicitly on the command line. <Partype> specifies the type of the parameter expected by the procedure. Valid types are "Real", "Integer", "String", "Workspace" and arrays of these "RealArray", "IntegerArray", "StringArray", "WorkspaceArray". If the type fails to match the actual parameter given by the user of the procedure an error message will be generated. Within the procedure, the parameter can be used just like a normal variable. For the simple types (Real, Integer and String), the variable in the procedure is just a copy of the original parameter and cannot be passed back. For all types of arrays and workspaces however, the procedure may modify the contents of the parameters passed to it.

Examples

```
ıta hacking
‹_results
ıd_point=Integer yarray=RealArray
 bad point
ıd_point] = 0.0

l to this procedure could be
:s  bad_point=100  yarray=MyThesisData
```

- RESULT <Resname> [= <default-value>]

The RESULT statement declares a special variable which will be returned at the end of the procedure as a function result. This result variable may be used normally during procedure execution and assigned any value. An optional default value can be specified such that the procedure will return this as the result even if the variable is never assigned another value. Any procedure which is to be called using the function syntax should return a result. Any procedure called using the function syntax but which does not contain a RESULT statement will return the value "Undefined."

Examples

```
 the standard functions with a new procedure
ıD         # sind(x) for angle 0 <= x <= 360
=real
 999.0     # Sentinel silly value
           # Global (defined elsewhere)

‹ ) AND ( x <= 360.0 )
ı( x * (2.0*$PI/360.0) )
```

- LOCAL <Localvar> [= <initial-value>] ...
- GLOBAL <Globalvar> [= <initial-value>] ...

The LOCAL and GLOBAL statements declare variables for use in the procedure.

Local variables are only accessible within the procedure, they are created on entry to the procedure and deleted on exit. If the procedure is called recursively, a new set of variables is created for each level of recursion.

Global variables may be shared with other procedures which have declared global variables of the same name. Unlike local variables, global variables last as long as the

Open GENIE session. There is only ever one global variable corresponding to one name (This is in contrast to local variables where there may be several different variables under the same name in different procedures).

Both global and local variables may be initialised with a value. For local variables, this will be reset each time the declaring procedure is entered. For global variables, this will be set only when the first procedure giving an initial value is loaded. The initial value is only assigned if the global variable does not already exist.

All variables created interactively are global variables.

Examples

```
bal & local variables
stG
hing = 45.0  anything = 50.0
ng = 90.0

hing " " anything


 1

.000000

sion 2
 = 88.0              # Override the GLOBAL
= 22.0               # a new global variable

.000000
```

Chapter 6

Storage

Although not a very explanatory title "Storage" is probably the best umbrella term to describe the ways and forms in which Open GENIE holds the data which it manipulates. This chapter gives an overview of the way in which *variables* are held in GENIE, some of the *conventions* used, and a detailed list of the variable types available and intrinsic operations which can be applied to them. The standard Open GENIE variable types are:

Real
Integer
String
Workspace
Arrays of the above types.

The variable pool

Each Open GENIE session can be viewed as having a pool of global variables associated with it. Every variable in the pool can be referred to by name. This allows the values of variables to be used or changed. When using Open GENIE interactively, assigning a value to an unused variable name will create a new variable and add that variable to the pool automatically. Variables can also be added to the pool from procedures by using the GLOBAL statement (local variables, parameters and qualifiers in procedures are slightly different in that they belong in a local pool for that procedure, just while it is running). The *lifetime* of a global variable - that is the time for which it retains its value and is accessible - is the entire Open GENIE session. By default, all variables created during a session will be lost when *Exit()* is typed.

This "variable pool" which forms the basis of Open GENIE is also used to store procedures, some of which are defined by the user and others which are predefined by Open GENIE during start-up. It is possible to see what variables are defined and the values they hold by using the *Show/Var* command. The *Show/Proc* command will list currently defined procedures. Normally, memory reclamation is looked after by the Open GENIE system but in circumstances where memory is short it is possible to explicitly free up the storage and remove a variable with the *Free()* command.

It is possible to preserve all the variables from one Open GENIE session to another. This is done using the *Save()* command to take a snapshot of all the variables and procedures currently available in the session. The session can then be restarted by specifying the saved image when restarting Open GENIE.

Conventions

There are a few conventions which are used in Open GENIE to distinguish different uses for variables and procedures. Some of these conventions are optional, others are not. It is recommended that these conventions are kept, especially when writing procedures.

Naming system procedures and variables

All procedure and variable names beginning with an underscore ("_") are reserved for use by Open GENIE and may change or be removed at any time. The exception to the rule is "_" itself which is the "undefined" value and can be used freely. By using the *Show/Proc/Sys* and *Show/Var/Sys* commands, all system variables or procedures will be shown along with the normal procedures and variables.

It is important to be very careful about using anything beginning with an underscore in a procedure. There is no guarantee that system variables and procedures will remain the same or even continue to exist in later versions of Open GENIE.

Constants

Constants in Open GENIE are variables whose names begin with the "$" symbol. As such they are defined using the same methods as normal variables, the only difference is that a constant variable can only have a value assigned to it once. Usually this would be done when the variable is first created, for example by using an initial value in the declaration. Subsequent attempts to change the value of a constant will fail. System constants begin "$_" and should be avoided. They should not be confused with ordinary constants defined by the system for general use, for example " $sea_green". Constants like this may be used when required and are guaranteed to be supported. All available constants can be viewed using the *Show/Const* command.

Case sensitivity

Open GENIE is largely insensitive to the case of commands. In true FORTRAN fashion it is possible to hit the case lock and type everything in upper case however it does make procedures harder to read if this is done. Keywords however must be in upper case, (e.g. LOOP, PROCEDURE) to ensure that they can be distinguished from user variables.

The recommended approach is use lower case for everything except control constructs (see *Multi-line commands*). For the pedantic, all references to procedures and their qualifiers should be capitalised (i.e. Capital first letter and the rest lowercase). Where a procedure or variable name consists of several words, the first letter in each word should be capitalised or an underscore should be used to separate the words (e.g. MyProcedure or My_procedure). The examples in this manual are kept as close to these conventions as possible.

Variable types

The following section describes the basic variable types available in Open GENIE. They fall into two groups, simple types which for most purposes can be treated as atomic, and compound types which consist of groupings of other variables.

The simple types are Real numbers, Integers and Strings and the compound types are Arrays and Workspaces. In this section, each type is described in some detail with a list of the intrinsic operations applicable to variables of that type.

Real

Real numbers are always floating point double precision values. The normal floating point operations are provided.

- x ^ y
  Raise to a power (x^y).

- + - * /
  Add, subtract, multiply and divide.

- `> = <= >= !=`
  Comparisons; Less than, Greater than, Equal to, Less than or equal to, Greater than or equal to and Not equal to. The equality tests are exact and so should be used with care for real numbers.

- Sin(x), Cos(x), Tan(x), Arcsin(x), Arccos(x), Arctan(x)
  Trig functions sin*x*,cos*x*,tan*x*,arcsin*x*,arcos*x*,arctan*x* (see *Mathematical Functions*).

- Log(x), Ln(x), Exp(x)
  Transcendental functions. $\mathrm{Log}_{10}x$, $\log_e x$, e^x (see *Mathematical Functions*).

Real values can be specified literally as a number with a decimal point or in scientific notation. Some examples of valid real numbers are given below.

```
233.0   +0.5   -.01   33.   1.6e-19
```

In an expression combining integers and real numbers, real numbers are considered to be the more general type and the result is given as a real after *implicitly* converting any integer values to corresponding real values. The conversion is carried out at the point when a real number and an integer are directly combined by an operator. For example,

```
>> Printn 5/2 * 5.0
10.000000
```

is evaluated differently to

```
>> Printn 5/2.0 * 5
12.500000
```

because the conversion to real numbers is delayed in the first expression.

Using this *implicit* conversion, integers may be *explicitly* converted to real numbers by multiplying by 1.0. This is useful when a using a command or procedure which insists on real numbers as parameters.

Integer

Integers are used in GENIE mainly for counting and specifying quantities. Array indices also have to be integer values. The normal integer operations are provided.

- x ^ y
  Raise to a power (x^y). The power need not be an integer but the result will be.

- + - * / |
  Add, subtract, multiply, quotient and remainder (modulo). The quotient and remainder operators use the convention of truncating towards zero where negative numbers are involved.

- < > = <= >= !=
  Comparisons; Less than, Greater than, Equal to, Less than or equal to, Greater than or equal to and Not equal to.

- All trigonometric and transcendental functions as for real numbers. The results of these are given as real numbers.

Integer values can be specified literally in decimal, hexadecimal, octal and binary. Some examples of valid integers are given below.

```
23   -1987   -0xBBC2   0o444   0b1011010100101010   +1
```

To convert a real number to an integer use the *As_integer()* function.

The default is to truncate an integer towards zero.

String

Open GENIE strings are of variable length and may be combined and assigned just like the other types of Open GENIE variables. The normal string operations are provided.

- + or &
  Concatenate two strings

- < > = <= >= !=
  Comparisons; Less than, Greater than, Equal to, Less than or equal to, Greater than or equal to and Not equal to. These have a dual functionality. For strings of the same length, comparisons act as a test based on the collating order of the text in the strings. For strings of different lengths, comparisons are between the string lengths.

- *Substring()*, *Locate()*, *Length()* (see *String Handling Functions*).
  The location of sub-strings within a string and the extraction of parts of the string can be carried out using these functions.

Literal strings are always specified within double quotes """ to avoid any conflict with variable names. The "\" character is a special character when used in a string. It introduces control sequences which could not otherwise be easily typed into a string. The sequences are listed below.

- \\
  Put a single "\" into a string.

- \n \t \f \v \r
  Put a newline, tab, form-feed, vertical tab or carriage-return into a string (respectively)..

- \"
  Put a double quote (") into a string.

- \0nn
  An octal byte value, must start with a zero.

- \Xnn
  An hexadecimal byte value, must start with "x" or "X."

- \nnn
  A decimal byte value, must *not* start with a zero (or it will be taken as an octal value)

Some examples of valid strings are given below.

```
"Hello world\n"           # line with a newline
"column1\tcolumn2\tcolumn3"  # tabs for a table
"Tinker\7 Bell"              # Bell character in decimal
escape = "\x1B"              # <ESC>
reset = escape + "c"         # ANSI device reset string
""                           # Null string
```

Most Open GENIE variable types can be converted to a string of some sort using the *As_String()* function. Used in combination with *Printn()* it allows numbers to be formatted into strings ready for later printing.

Examples

```
# 1. Formatting an array
PROCEDURE PrintArray
PARAMETERS Array_to_print=Realarray
LOCAL buffer

buffer = ""
LOOP i FROM 1 TO Length(Array_to_print)
    buffer = buffer + "\t" + As_string(Array_to_print[i])
    IF Length(buffer) >= 70
        Printn "--" buffer
        buffer = ""
    ENDIF
ENDLOOP
Printn "--" buffer

ENDPROCEDURE
# 2. String concatenation with a number
>> a = as_string(3.0)
>> a = "*twiddly bits -> " + a + " <- twiddly bits*"
>> Printn a
*twiddly bits -> 3.0 <- twiddly bits*
```

Arrays

In Open GENIE there are four array types. They provide for n-dimensional arrays of each of the four the basic variable types. Arrays may be manipulated as a whole or, alternatively, individual elements may be accessed and modified separately. The four array types are:

- RealArray
- IntegerArray
- StringArray
- WorkspaceArray

An array is created in two stages, firstly by creating an empty array of the appropriate size and dimensionality using the *Dimensions()* function and secondly by assigning a value to one of its elements. It is only when an element is assigned a value that the final type of the array is chosen. From this point on the type of the array is fixed.

Examples

```
# Create a RealArray
>> two_d_data = Dimensions(100,200) # created in 2-dimensions
>> two_d_data[1,1] = 5.0            # type is set here to real
>> Printn two_d_data
[5.0 _ _ _ _ ...] Array(100 200 )
```

In the example above the *Printn()* command is used to show the first few elements of the newly created array. Notice that all elements which have not had a value assigned to them are marked as "_". If these are used in a calculation, the undefined value will propagate such that the result of any operation involving an undefined value will also become undefined itself.

The same array operations apply to all Open GENIE array types, any differences in effect are as a result of differences in the types of the array elements. For example, string arrays cannot be multiplied!

- Array[i, j, k, ...] (Indexing)
  Individual array elements are accessed using integer indices. All indices start at one and can be as large as there is memory available for them. Whenever an element is being accessed the appropriate indices must be specified after the array name in a comma separated list between square brackets.

- Array[l1:o1, l2:o2, ...] (Slicing)
  This operation produces a smaller array by "slicing" up a larger array. The smaller array retains the dimensionality of the larger array as long as each dimension is given a slicing interval (i.e. lower:higher). Note that if one dimension is fixed, the dimensionality of the sliced array is effectively reduced by one.

- + - (Unary operations)
  The unary operators are applied individually to each element of the array. The results are placed into a new array of the same size and dimensionality as the old array.

- + - * / | ^ (Binary)
  The Binary operators are applied to corresponding elements of the two arrays, if the arrays differ in dimensionality, the corresponding elements are selected using storage order (all arrays are stored in row-major order as a contiguous block). The result of the operation is to create a new array of identical structure to the array on the left hand side of the operator. There are three cases which this gives rise to:

1. The arrays are of equal length.
   All pairs of elements are processed and a result is stored in the result array.

2. The LHS array is longer.
   The extra elements are set to the undefined value "_" in the result array.

3. The LHS array is shorter.
   The result array is truncated to the length of the LHS array.

   The result of this is that by specifying the order differently (for commutative operators) it is possible for the user to define whether to truncate or fill the result array with undefined values.

- < > <= >= = !=
  Comparisons for arrays, respectively: Shorter than, Longer than, Shorter than or equal to, Longer than or equal to, Elements and length are identical, Elements and/or length differ. All length tests are carried out on the array as stored ignoring dimensionality. Exact equivalence "=" and "!= " also test all corresponding elements for equality.

- &
  Appends the RHS array to the end of the LHS array, for this operation both arrays must be of the same type or an error is generated.

- All trigonometric and transcendental functions are treated in the same way as unary operators

- *Length()* (See *Array Handling Functions*)
  Returns the length of the array as stored (the figure returned will be the product of the dimensions for a multidimensional array).

When arrays are used in algebraic expressions, several implicit type conversions can occur. The aim of these is to allow simple and expressions to express relatively complex concepts. The rules work in a similar way to those already explained when mixing integers and real numbers in an expression and are largely intuitive. For example, given an array of type RealArray called y_values the whole array may be multiplied by a constant.

```
>> y_values = y_values * 2
```

The implicit conversion here is to make the integer constant "2" into a RealArray of elements value 2.0. At this point a standard array multiplication can be carried out to find the result. There are a large number of implicit conversion in Open GENIE (See *Implicit Data Conversions*).

Workspace

A workspace is a compound variable where individual elements of the variable may be of different types (in contrast to arrays which are all of one type). Examples of similar structures in other languages are a record in Pascal or a struct in ‘C.’

Open GENIE workspaces are made up of fields consisting of variables of any of the Open GENIE data types, these fields may be created dynamically while a session is running. This means that during the process of analysing data, new information can be added into the description of a workspace without necessarily needing to know what type the new data will be beforehand. The following example shows the of the construction of a simple workspace interactively.

```
#  create a brand new workspace
>> mywork = fields()    # creates an empty workspace
>> Printn mywork        # look at it now
Workspace []
(
)
>>
# Now put some fields in it
>> mywork.description = "My test workspace"
>> mywork.some_value = 3.14159
>> mywork.anarray = dimensions(2,3)
>> mywork.anarray[1,1] = 25
>> Printn mywork         # now look again
Workspace []
(
  some_value = 3.14159
  anarray = [25 _ _ _ _ ...] Array(2 3 )
  description = "My test workspace"
)
>>
```

Open GENIE workspaces are unlike other types of variable in that the operations on them may be re-defined and/or extended by the user. To make the process easier, a set of template operations are defined which work in a manner similar to the way the original GENIE-V2 program operated on its workspaces (see *Workspace Operations*). The symbolic operators (+ * / etc.) map onto calls to generic Open GENIE procedures which implement the operations. The mapping of symbols to procedures is given in the list of operations shown below.

The basic Open GENIE Workspace operations are listed below.

- workspace.field-name (Field accessing)
  Any field in a workspace may have its value set or read by using the name of the workspace followed by a "." and a field name. A field is automatically created in a workspace when a value is assigned to it (as shown in the previous example). Once created, a field retains its value until the workspace is destroyed or the Open GENIE session ends.

- + - (Unary)
  Calls *Workspace_negated()* (The unary operation "+" is a no-op).

- + - * / | ^ (Binary)
  These respectively call *Workspace_add(), Workspace_subtract(), Workspace_multiply(), Workspace_divide(), Workspace_modulo(), Workspace_raised_to()*

- < > <= >= = !=
  These respectively call *Workspace_less_than(), Workspace_greater_than(), Workspace_less_than_or_equal(), Workspace_greater_than_or_equal(), Workspace_equal(), Workspace_not_equal()*

- &
  Calls *Workspace_append()*

- All the trigonometric and transcendental functions are defined generically and call workspace specific functions *Workspace_sin()*, *Workspace_cos()*, *Workspace_tan()*, *Workspace_arcsin()*, *Workspace_arccos()*, *Workspace_arctan()*, *Workspace_ln()*, *Workspace_exp()*, *Workspace_log()*, *Workspace_sqrt()*, *Workspace_abs()*

- *Length()*
  This generic function returns the number of fields in the workspace.

Automatic conversions (see *Implicit Data Conversions*) apply to operations combining workspaces with other Open GENIE numeric types. It is important to note however, that this will follow whatever is defined in the *Workspace_coerce()* function which can be re-defined by the user.

The default action on a workspace (i.e. defined by the template routines) is to perform the operation on the array of y values contained within it. Consequently, if a workspace is not of the default GENIE-V2 structure, arithmetic routines as defined by the templates will not be applicable.

Chapter 7

Array and Workspace Handling Functions

This section describes the basic utility functions for creating and using Open GENIE workspaces and arrays. For detailed information on the data analysis based use of workspaces see the section on *Workspace Operations*. Most arithmetic functions on arrays are described in the section on *Mathematical Functions* and are simply generalisations of the scalar functions.

The functions documented in this section are listed below.

Array functions	**Description**
Bracket	Finds the index position of a given value in an array.
Centre_bins	Centre the bins of a real array.
Cut	Takes a one-D cut out of a two-D array
Dimensionality	Return an integer giving the dimensionality of the array
Dimensions	Create and size an Open GENIE array
Fill	Fill an array with values
Fix	Fix any undefined values in an array
Max	Find the maximum value in an array
Min	Find the minimum value in an array
Redim	Re-dimensions an array whilst keeping the size and contents identical
Sum	Sums all the elements in an array
Unfix	Replace sentinel values with "undefined"
Workspace	

functions

Fields Create a workspace

See also the generic *Length()* command (documented in String Operations) for getting the number of fields in a workspace or the total length of an array.

Array Function Reference

Bracket()

Finds the lower index position of a given value in an array.

BRACKET	**xarray**=*Realarray* **xval**=*Real*	Allows bracketing of X values.

example:

```
# Rebin data to a smaller range manually
>> lower = bracket(w.x, 1000.0)
>> upper = bracket(w.x, 4000.0)
>> w.x = w.x[(lower):(upper)]
>> w.y = w.y[(lower):(upper-1)]
>> w.e = w.e[(lower):(upper-1)]
```

Note: Normally this would be done with the Rebin() command.

Bracket

This command is effectively used to "find" the location or nearest location to a number *n* in a real array, where the assumption *for all i x[i] <= x[i+1]* holds true for an array *x* (the X values of a histogram). When this is so return *i* such that *x[i] <= n <= x[i+1]*.

This command is used when it is necessary to bracket out a section of a histogram based on the actual X values.

Parameters:

Xarray (Realarray)

An array which fulfils the assumption above.

Xval (Real)

The value being searched for.

RESULT = (Integer)

This is either the index of the first element with the value "xval" or the index of the element such that "xval" would be included within the open interval (x[i], x[i+1]) where i is the returned index.

If the range where the value would be found is not present in the array then a value of -1 is returned.

Centre_bins()

Centre the bins of a real array (i.e. convert from histogram to point mode).

CENTRE_BINS	**array**=*Array*	Centre the bins of a real array (i.e. convert from histogram to point mode).

example:

```
# Do a rough conversion from a histogram workspace to
# to a new point mode workspace
point_mode = my_work
point_mode.x = centre_bins(my_work.x)

# convert an array to point mode
centre_bins my_array
```

Centre_bins

The Centre_bins() command is useful for converting histogram data to point mode data. Used on the X-array of a workspace, it will reduce the total number of X points by one and the X values left will be those of the centre of each bin.

Parameters:

Array (Array)

Any valid Open GENIE array type

RESULT = (Array)

Returns the centered array

Cut()

Takes a one-D cut out of a two-D array

CUT	**array**=*Realarray* **dimension**=*Integer* **index**=*Real*	Takes a slice out of an array.

example:

```
# Take a cuts at 1 and 1.5 along the 2nd dimension
printn y
>> y = Dimensions(2,3)
>> fill y 1.0 1.0
>> printn cut(y, 1, 1.0)
[1.0 2.0 3.0 ] Array(3 )
>> printn cut(y, 1, 1.5)
[2.5 3.5 4.5 ] Array(3 )
```

Note: Returns a one-D array of one element if a one-D array is supplied.

Cut

The Cut() command takes a one dimensional slice out of a two dimensional array (useful in two dimensional graphics applications where a one dimensional plot is required along one axis). Depending on which dimension is chosen from the source array, the length of the resulting array is the length of the orthogonal dimension. The cut is taken at an index value along the dimension selected and the values of the resulting slice are linearly interpolated for this position of the index value.

Parameters:

Array (Realarray)

The array to be cut

Dimension (Integer)

Specifies the dimension along which the index position of the slice is specified.

RESULT = (Realarray)

The values along the orthogonal dimension, interpreted if necessary for a non-integer index value.

Dimensionality()

Return an integer giving the dimensionality of the array

DIMENSIONALITY [**array**=*Array*] Returns the number of dimensions

example:

```
# create an array and print its dimensionality
>> a = dimensions(2,3,4)
>> printn dimensionality(a)
3
```

Dimensionality

Returns the dimensionality of an array.

Parameters:

Array (Array)

The array to test the dimensionality of.

RESULT = (Integer)

Number of dimensions

Dimensions()

Create and size an Open GENIE array

DIMENSIONS()	**dim1**=*Integer* [**dim2**=*Integer*] ... [**dim10**=*Integer*]	Create an empty array

example:

```
# Dimension a 1 x 2 x 3 array
>> a = dimensions(1, 2, 3)
>> printn is_a(a, "Realarray")
$FALSE
>> a[1,2,1] = 4.0
>> printn is_a(a, "Realarray")
$TRUE
>> printn a
[_ _ _ 4.0 _ ...] Array(1 2 3 )
```

Note: Arrays print as if they are one dimensional (in row major order).

Dimensions

The dimensions command is used for creating arrays of a specified length and dimensionality. The number of "dim" parameters specified gives the dimensionality and the values give the length of each dimension. In Open GENIE arrays always start at 1.

Until a value has been assigned to one element of the array, the array has no fixed type and it will take on the type of the first value assigned to one of its elements. After this, trying to assign a different type will create an error as arrays may only contain variables of one type. Elements in an array that are undefined are shown as an "_" character. Undefined values in arrays can be changed to fixed values using the *Fix()* command.

Parameters:

Dim1, ..., Dim10 (Integer)

Sizes of each array dimension.

RESULT = (Array)

An array of the correct size and dimensionality but with an as yet undecided type.

Fill()

Fill an array with values

FILL	**value**=*Any* [**step**=*Number*]	Fill and array with values
[/GEOMETRIC]		Fill array geometrically

example:

```
# Create some geometric data
>> a = dimensions(200)
>> Fill/Geometric a 1.0 0.9
>> printn a
[1.0 0.9 0.81 0.729 0.6561 ...] Array(200 )
```

Note: step values only work with integer and real arrays.

Fill

The Fill() command provides a quick way of filling up an array with data. For non-numeric data, it is just a way of filling the array with identical values. For numeric data, it is also a quick way of putting some useful data in an array, for example, an arithmetic fill is ideal for generating an axis to plot an array of data against. In the same mode, a geometric fill creates automatic log data to plot against.

Parameters:

/Geometric

Increase the values placed into the array geometrically based on the starting value and the step

Value (Any)

The value to fill the array with, or, if "step" is used, this is the starting value of the series and appears in the first element of the array.

Step (Number)

Sets a step size (or rate if /Geometric) is specified.

RESULT = (Array)

The filled array can also be returned as a result. In this case, the original array remains unmodified.

Fix()

Fix any undefined values in an array

FIX	**array**=*Array* **value**=*Any (except Array)*	Replaces any instances of "undefined" in an array.

example:

```
# Fix undefined results to 999
>> fill a -4.0 1.0
>> printn 10/a
[-2.5 -3.33333333333333 -5.0 -10.0  _  ...] Array(6 )
>> fix a 999.0
>> printn a
[-2.5 -3.33333333333333 -5.0 -10.0 999.0 ...] Array(6 )
```

Note: See Unfix() for replacing sentinel values with "_" on data import.

Fix

Open GENIE caters for situations where an element of an array has an undefined value. This value is correctly propagated through all internal operations but can cause problems, especially when exporting data. To get round this problem, the Fix() command is used to replace any instances of an undefined value in an array with a specific sentinel value.

To go the other way, for example with imported data which, say, has undefined values represented by a specific number, the *Unfix()* command can be used.

Parameters:

Array (Array)

The array containing one or more undefined values to be fixed.

Value (Any except an array)

A value of the same type as the array with which to replace any undefined values.

RESULT = (Array)

An array of the same type as the original but with all undefined values replaced with a sentinel value of the correct type.

Max()

Find the maximum value in an array

MAX() **array**=*Array* Find the value of the highest element

example:

```
# Find the index of the highest data value
>> print Bracket(w.y, Max(w.y))
12456
```

Max

The Max() function returns the value of the largest element of the array it is applied to. The size of an element is determined by comparing elements with each other using the "<" and "< "operators. As a result string arrays may also have a maximum value.

In arrays where there are undefined elements, the maximum value returned is the value of the largest defined element. For correctness, the array should have all undefined elements "fixed" explicitly to a defined value first. This can be done using the *Fix()* command.

Parameters:

Array (Array)

Array to be tested.

RESULT = (Array element type)

The maximum value found in the array.

Min()

Find the minimum value in an array

MIN()	**array**=*Array*	Find the value of the lowest element

example:

```
# Find the index of the lowest data value
>> print Bracket(w.y, Min(w.y))
245
```

Min

The Min() function returns the value of the smallest element of the array it is applied to. The size of an element is determined by comparing elements with each other using the "<" and "< "operators. As a result string arrays may also have a minimum value.

In arrays where there are undefined elements, the minimum value returned is the value of the smallest defined element. For correctness, the array should have all undefined elements "fixed" explicitly to a defined value first. This can be done using the *Fix()* command.

Parameters:

Array (Array)

Array to be tested.

RESULT = (Array element type)

The minimum value found in the array.

Redim()

Re-dimensions an array whilst keeping the size and contents identical

REDIM	[**array**=*Array*] **i**=*Integer* [**j**=*Integer*] [**k**=*Integer*][**l**=*Integer*]	Re-dimensions an array whilst keeping the contents identical

example:

```
>> printn a
[1.0 2.0 3.0 4.0 5.0 ...] Array[400 ] Array(1 )
>> printn redim(a,20,10,2)
[1.0 2.0 3.0 4.0 5.0 ...] Array[20 10 2 ] Array(3 )
>> printn a[1,2,1]
3.0
```

Note: Redim will return an error if an attempt is made to change the total size of the array

---

Redim

The Redim() command is used to re-dimension an array whilst keeping the contents and overall size of the array the same. Arrays in GCL are stored in row major order so that the fastest varying index is on the Right hand side.

Parameters:

Array (Array)

Any valid Open GENIE array type

i, j, k, l (Integer)

Up to four integer indices for re-dimensioning the array

RESULT = (Array)

Returns the re-dimensioned array

Sum()

Sums all the elements in an array

SUM()	**array**=*Array*	Sum all the elements

example:

```
# Integrate some data
>> printn sum(w.y)
12456890.789
```

Sum

The Sum() function returns the sum of all the elements in an array.If the array contains an undefined element the result will always be undefined. To get around this when a result is required, all undefined elements should be "fixed" to a chosen value (e.g. 0.0) using the *Fix()* command.

Parameters:

Array (Array)

Array to be summed.

RESULT = (Array element type)

The sum of all the data in an array.

Unfix()

Replace sentinel values with "undefined"

UNFIX	**array**=*Array* **value**=*Any (except Array)*	Replace a specific value with undefined

example:

```
# Fix undefined results to 999
>> scan = get_dat("Old_data_file")
>> Unfix scan 999.0    # 999.0 represents unknown values
```

Note: See Fix() for replacing undefined values with a sentinel value.

Unfix

Open GENIE caters for situations where an element of an array has an undefined value. This value is correctly propagated through all internal operations but when importing data it may be necessary to convert external sentinel values into proper Open GENIE undefined values before operating on the data.

To go the other way, for example when exporting data the *Fix()* command can be used.

Parameters:

Array (Array)

The array containing one or more sentinel values to be fixed.

Value (Any except an array)

The sentinel value to replace with proper undefined values in Open GENIE.

RESULT = (Array)

An array of the same type as the original but with all sentinel values replaced with undefined.

Workspace Function Reference

Fields()

Create a workspace

FIELDS() Create a new empty workspace.

example:

```
# Example in preformatted font
>> w = fields()
>> w.new = 5
>> printn w
  Workspace []
  (
    new = 5
  )
```

Fields

Creates an empty workspace to which fields may be added.

Parameters:

RESULT = (Workspace)

Returns an empty workspace.

Chapter 8

String Handling Functions

Open GENIE supports a "String" type which allows the normal programming operations to be carried out on string variables, see also *String operations* (in the chapter on Storage).

There are also several functions designed for working with Open GENIE strings.

The functions which operate on strings directly are:

Substring()
Locate()
Length()

There are also two powerful functions which handle conversions to and from strings and are very useful for input and output formatting.

As_string()
As_variable()

Many other functions take strings as parameters or return them as results. Arrays of strings are handled similarly to arrays of other Open GENIE types except that no arithmetic operations are permitted (for obvious reasons!).

Examples

```
# 1.
# play around using Locate() and Substring()
>> nth = 2      # find the 2nd occurrence
>> printn Locate("you fish you",  "you", nth)
10              # start position of 2nd "you"
>> pos = 6; length = 4
>> printn Substring("Blackbird", pos, length)
bird
# 2.
# convert some variables
>> printn as_variable(".34e1") * 5.6
19.04
>> t = 19.04
>> temp_string = as_string(t) + "mK"
>> printn temp_string
# 3.
# or even a whole workspace
>> w=s(1)
>> x = as_string(w)
>> printn is_a(w, "string")
```

```
false
>> printn is_a(x, "string")
true
```

String Handling Function Reference

Substring()

Extracts a smaller string out of a larger string using the given indices.

SUBSTRING()	**astring**=*String* [**start**=*Integer*] [**length**=*Integer*]	Select a string of a specified length from a given starting position.

example:

```
# pick the user name from this string
>> printn substring("Run: no, User: A Neutron", 16)
A Neutron

# Select just the initial
>> printn substring("Run: no, User: A Neutron", 16, 1)
A
```

Note: if the length value is null, the rest of the string is returned.

Substring

The Substring() command extracts a smaller string from a larger string. Characters in the string are selected by an index starting at 1 for the first character position in the string.

Parameters:

Astring (String)

The string from which a sub-string is to be selected

Start (Integer)

Starting position of the sub-string to take.

length (Integer)

The length of the sub-string to take.

RESULT = (String)

The sub-string selected.

Locate()

Finds the position (or positions) of a given sub-string within a string.

LOCATE	**astring**=*String* **substring**=*String* [**count**=*Integer*]	Returns the index position of a sub-string within the string

example:

```
# Look for all the positions of the string "quick"
>> position = 0
>> words = "The quick brown quick fox&
  jumped quick over the lazy quick dog"
LOOP i FROM 1 TO 10
    position = locate(words,"quick",i)
    EXITIF position = 0
    print position
ENDLOOP
5 17 33 53>>
```

Note: Locate() is most useful when used in conjunction with the Substring() function.

Locate

Locate allows the position of a substring to be found within a string. If necessary, it is possible to specify an extra count parameter to locate which will allow the position of the nth occurrence of a string to be found. In all cases, if a sub-string is not found, a zero is returned.

Parameters:

Astring (String)

The string which is to be searched for sub-strings.

Substring (String)

The sub-string to search for within the string

count (Integer) [default=1]

Optional parameter to allow the position of the nth substring to be found. This parameter defaults to 1 to look for the first occurrence of the substring.

RESULT = (Integer)

The position of the sub-string within the searched string. The character positions are numbered from 1 for the first character in the string, the same way as for the *Substring()* command.

Length()

Generic command for returning the length of something

LENGTH **item**=*Generic* Returns the length of a variable

example:

```
# a String
>> printn length("abcd")
4
# an array
>> a = dimensions(2,3)
>> printn length(a)
6
```

Note: for workspaces, Length() returns the number of defined elements.

Length

This is a generic command which may be used on nearly all Open GENIE data types but most usefully on Strings, Arrays and Workspaces.

Parameters:

Item (Generic)

This is an Array, String, or Workspace parameter where the type of the item is of variable length. For arrays, the length is the total length, for workspaces it is the number of fields.

RESULT = (Integer)

The length or size of the item.

As_string()

Converts any genie variable into a string.

AS_STRING **var**=*Generic* Print a variable into an internal string

example:

```
# Format a real to two decimal places
>> printn myval
34.234567
>> str = as_string(myval)
>> prec = Substring( str, _, locate(str,".") + 2 )
>> printn prec
34.23
# useful for graphics
>> Draw/text xcoord=0.5 ycoord=0.5 prec
```

Note: Has a similar purpose to the FORTRAN internal write.

As_string

This function is used in much the same way as the FORTRAN internal write statement. It allows a GCL program to get access to the string which would normally be printed out to a terminal with a *Print()* command. A likely use of this command is formatting text which is going to be displayed on the graphics screen (which the *Print()* commands do not write to). Being a generic function, the As_string() command will work with any sort of Open GENIE variable.

Parameters:

Var (Generic)

Any Open GENIE variable or expression can be put here.

RESULT = (String)

A string exactly as if the variable had been printed to the console window with the *Print()* command.

As_variable()

Converts a string into an internal Open GENIE variable type.

AS_VARIABLE **astring**=*String* Read a string into a variable

example:

```
# Obtain a number from an awkward string
>> printn w.title
BaCuO2 Sprogget & Sylvester t=.589E+1K
>> tstr = Substring(w.title, locate(w.title,"=")+1)
>> printn as_variable(tstr)
5.89
```

Note: Only reads valid numbers otherwise returns the string.

As_variable

This function is used in much the same way as the FORTRAN internal read statement. It allows a GCL program to get a real or integer value from a string. Normally for user input this is done automatically by the *Inquire()* command but there may be situations where, as in the example above a number already comes as a string type.

The As_variable() command will attempt to return either a real or integer depending on whether a decimal point is in evidence. If the string passed to the command starts with something which cannot be a number, the original string is returned. It is assumed that, as in the example above, the numeric part of the string is being passed. There is no need to trim the end of the string (unless it makes the number ambiguous) as any extra text will be discarded.

Parameters:

Astring (String)

A string which starts with the number to be read. Any white space in front of the number will be ignored.

RESULT = (Real, Integer or String)

Either a String, Integer or Real depending on whether the function was able to decode a number.

Chapter 9

Mathematical functions

As well as providing intrinsic arithmetic operations for different data types (See *Storage - Variable Types*) Open GENIE provides several basic functions for performing mathematical operations. These are coded generically so that the same function can be applied to any data type which is capable of undergoing the operation. For example, the *Sin()* function may be called for a single number, an array or a data workspace. Normally, the result of the operation is of the same data type as the value operated upon. Where the operand contains several values (eg an array), each value is operated on individually in isolation to the others and an individual result is calculated for that value.

For example we can create an array of real numbers, and square root them all in one go.

```
>> my_numbers = Dimensions(10,20) # First create a 10 x 20 array
>> fill my_numbers 1.0 1.0   # Fill the array with some data
>> printn my_numbers
[1.0 2.0 3.0 4.0 5.0 ...] Array(10 20 )
>> printn sqrt(my_numbers)   # print the square roots.
[1.0 1.414213 1.732050 2.0 2.236067 ...] Array(10 20 )
```

When one of these generic operations is applied to a data workspace, a separate routine is called which allows the user to define the operations that are carried out (see *Workspace Operations*).

Trigonometric Functions

Arccos()
Arcsin()
Arctan()
Cos()
Sin()
Tan()

Transcendental Functions

Exp()
Ln()
Log()

Miscellaneous Functions

Abs()
Sqrt()

Trigonometric Function Reference

Arccos()

Generic trig functions.

ARCCOS(x) **x**=*Generic* Arccosine in radians.

example:

```
# print result in degrees
y = Arccos(0.5) * 180 / $pi
#take the arccos of all elements in an array
a = Dimensions(10)  # create a 10 element array
fill a 0.5          # Set all elements to 0.5
y = Arccos(a)
```

Note: These functions can be applied to all numeric types (including arrays & workspaces).

Arccos()

These function returns an Arccosine value in radians.

Parameters:

X (Generic)

A single number, array or workspace containing the values to which the appropriate trig function is to be applied. Angles must be specified in radians. Undefined values passed to this function will be returned as an undefined result.

RESULT = (Generic)

Angle(s) in radians.

Arcsin()

Generic trig functions.

ARCSIN(x)	**x**=*Generic*	Arcsine in radians.

example:

```
# print result in degrees
y = Arcsin(0.5) * 180 / $pi
#take the arcsine of all elements in an array
a = Dimensions(10)  # create a 10 element array
fill a 0.5          # Set all elements to 0.5
y = Arcsin(a)
```

Note: These functions can be applied to all numeric types (including arrays & workspaces).

Arcsin()

These function returns an Arcsine value in radians.

Parameters:

X (Generic)

A single number, array or workspace containing the values to which the appropriate trig function is to be applied. Angles must be specified in radians. Undefined values passed to this function will be returned as an undefined result.

RESULT = (Generic)

Angle(s) in radians.

Arctan()

Generic trig functions.

ARCTAN(x) **x**=*Generic* Arctangent in radians.

example:

```
# print result in degrees
y = Arctan(1000.0) * 180 / $pi
#take the arctan of all elements in an array
a = Dimensions(10)  # create a 10 element array
fill a 0.5          # Set all elements to 0.5
y = Arctan(a)
```

Note: These functions can be applied to all numeric types (including arrays & workspaces).

Arctan()

These function returns an Arctangent value in radians.

Parameters:

X (Generic)

A single number, array or workspace containing the values to which the appropriate trig function is to be applied. Angles must be specified in radians. Undefined values passed to this function will be returned as an undefined result.

RESULT = (Generic)

Angle(s) in radians.

Cos()

Generic trig function.

COS(x)	**x**=*Generic*	Cosine in radians.

example:

```
# take cosine of angle in degrees
y = Cos(90.0 * 180.0/$PI)
#take the cosine of all elements in an array
a = Dimensions(10)  # create a 10 element array
fill a $pi*2.0      # Set all elements to pi*2
y = Cos(a)
```

Note: This function can be applied to all numeric types (including arrays & workspaces).

Cos()

These function calculates the cosine of an angle in radians.

Parameters:

X (Generic)

A single number, array or workspace containing the values to which the trig function is to be applied. Angles must be specified in radians. Undefined values passed to this function will be returned as an undefined result.

RESULT = (Generic)

Cosine of the angle(s) given in radians.

Sin()

Generic trig function.

SIN(x)	**x**=*Generic*	Sin in radians.

example:

```
# take sin of angle in degrees
y = sin(40.0 * 180.0/$PI)
#take the sin of all elements in an array
a = Dimensions(10)  # create a 10 element array
fill a 0.5          # Set all elements to 0.5
y = Sin(a)
```

Note: This function can be applied to all numeric types (including arrays & workspaces).

Sin()

These function calculates the sine of an angle in radians.

Parameters:

X (Generic)

A single number, array or workspace containing the values to which the trig function is to be applied. Angles must be specified in radians. Undefined values passed to this function will be returned as an undefined result.

RESULT = (Generic)

Sine of the angle(s) given in radians.

Tan()

Generic trig function.

TAN(x) **x**=*Generic* Tangent in radians.

example:

```
# take tangent of angle in degrees
y = Tan(90.0 * 180.0/$PI)
#take the tangent of all elements in an array
a = Dimensions(10)  # create a 10 element array
fill a 0.5          # Set all elements to 0.5
y = Tan(a)
```

Note: This function can be applied to all numeric types (including arrays & workspaces).

Tan()

These function calculates the tangent of an angle given in radians.

Parameters:

X (Generic)

A single number, array or workspace containing the values to which the trig function is to be applied. Angles must be specified in radians. Undefined values passed to this function will be returned as an undefined result.

RESULT = (Generic)

Tangent of the angle(s) given in radians (or undefined where the function result is beyond the precision of the machine).

Transcendental Function Reference

Exp()

Generic transcendental functions.

EXP(x)	**x**=*Generic*	Exponentiate

example:

```
# antilog a data array
# which is in logs to base 10
data = Exp( w.y * ln(10.0) )
```

Note: These functions can be applied to all numeric types (including arrays & workspaces).

Exp()

Exponentiates, i.e. takes e^x

Parameters:

X (Generic)

A single number, array or workspace containing the values to be exponentiated. Undefined values passed to this function will be returned undefined.

RESULT = (Generic)

Exponentiated value(s).

Ln()

Generic transcendental functions.

LN(x)	**x**=*Generic*	Log to the base e

example:

```
# Take logs to base e of a data array
lndata = Ln(w.y)
```

Note: These functions can be applied to all numeric types (including arrays & workspaces).

Ln()

Take natural logarithms.

Parameters:

X (Generic)

A single number, array or workspace containing the values to take logs of. Undefined values passed to this function will be returned undefined. Illegal values (i.e. negative numbers are permitted) but will return an undefined result.

RESULT = (Generic)

Either the log of the value or an undefined result.

Log()

Generic transcendental functions.

LOG(x)	**x**=*Generic*	Log to the base 10

example:

```
# Take logs of a data array
log10data = Log(w.y)
```

Note: These functions can be applied to all numeric types (including arrays & workspaces).

Log()

Take logs to base10.

Parameters:

X (Generic)

A single number, array or workspace containing the values to take logs of. Undefined values passed to this function will be returned undefined. Illegal values (i.e. negative numbers are permitted) but will return an undefined result.

RESULT = (Generic)

Either the log of the value or an undefined result.

Miscellaneous Function Reference

Abs()

Calculate the absolute values.

ABS(x)	**x**=*Generic*	Calculate |x|.

example:

```
# Print the absolute value
printn Abs(-3.3)
3.3
```

Note: This function can be applied to all numeric types (including arrays & workspaces).

Abs()

Calculates the absolute value of a number or of the numbers in an array. All negative numbers will be returned as positive numbers of the same magnitude.

Parameters:

X (Generic)

A single number, array or workspace containing the values for which to find |x|. Undefined values passed to this function will be returned undefined.

RESULT = (Generic)

Either the absolute value or an undefined result.

Sqrt()

Calculate the square root of a value.

SQRT(x)	**x**=*Generic*	Calculate the square root.

example:

```
# Calculate the square root of Pi
printn Sqrt($PI)
1.77245310234149
```

Note: This function can be applied to all numeric types (including arrays & workspaces).

Sqrt()

Calculates the square root of a number.

Parameters:

X (Generic)

A single number, array or workspace containing the values to square root. Undefined values passed to this function will be returned undefined. Illegal values (i.e. negative numbers are permitted) but will return an undefined result.

RESULT = (Generic)

Either the square root of the value or an undefined result.

Chapter 10

General Programming Functions

This chapter lists functions built into the Open GENIE command language which don't fall directly under the other section headings but are likely to be fairly essential in writing any Open GENIE programs.

Alias()	Allows the definition of aliases for command names and qualifiers
As_integer	Truncates an Open GENIE real number to an integer.
Called_as_function	Test if a procedure was called as a function, i.e. x = command(y)
Defined	Tests whether a variable has a defined value (not undefined).
Free	Deletes a variable or workspace field and frees the storage.
Inquire	Prompts the user for terminal input and reads back an input variable.
Interactive	Tests whether code is running as part of a procedure or not.
Is_a	Tests the type of an Open GENIE variable.
Now	Returns the current date and time as a string.
Read_terminal	Read a string back from the input terminal into a variable

General Programming Function Reference

Alias()

Allows the definition of aliases for command names and qualifiers

ALIAS	**alias_name**=*String* **full_name**=*String*	Define one or more aliases for commands and qualifiers.

example:

```
# Some aliases defined for the display command
>> alias "d/e/m/l" "display/errors/markers/line"
>> d/e/l

# can have more than one alias defined for a command
>> alias "showit/Ebars/stars/wigglything" "display/errors/markers/line"

>> showit/e_bars/wigglything
```

Note: Open GENIE already defines some common aliases for commands (e.g. d/e for display/errors).

Alias

The Alias() command is provided as a convenience for the command line interface to Open GENIE. It allows any Open GENIE procedure to effectively take on another, usually shorter name. The same can be done for any qualifiers taken by the procedure. The command relies on matching up the names of the real command and qualifiers with the names you wish to alias them to. The procedure and number of qualifiers must match in the two strings and those in the second string must be a valid command and its qualifiers (or they will not work).

This command is so powerful that some care should be used when exercising it! Note that it is entirely possible to define aliases which swap or duplicate the meanings of procedures and qualifiers.

A more serious use for this command is when defining a custom procedure in place of a standard Open GENIE procedure. For example, say we wish to redefine the *Spectrum()* command to load up some custom two theta values. By aliasing the original Spectrum command as old_spectrum. We can write a new Spectrum() procedure which first calls the old old_spectrum() procedure to get most of the data and then add our own code to fix the two theta values. To the user, there is no change to how the command works and they can still call Spectrum() but will get the modified version (Note that without aliasing the name, Open GENIE would think that we were writing Spectrum() as a recursive procedure).

Parameters:

Alias_name (String)

These are the new (fictitious) names for the command and its qualifiers. It is not necessary to specify all the qualifiers a command has, just the ones that you wish to abbreviate or alias. Any unspecified qualifiers from the original command will still work perfectly with the new command name.

full_name (String)

The exact command name and qualifiers which we wish to alias, keeping the order so that this list matches the aliasing parameters in number and position.

As_integer()

Truncates an Open GENIE real number to an integer.

AS_INTEGER()	**var**=*Real*	Convert to integer

example:

```
# Convert 2.5 to an integer
>> printn As_integer(2.5)
2
```

Note: Add 0.5 before truncating to get the *nearest* integer instead

As_integer

A simple function to convert Open GENIE real numbers into integers. Note that Open GENIE loops work fine with Real numbers as loop counters so the most likely reason for converting to an integer is to index into an array or if exact precision arithmetic is required (a loop with real numbers may be subject to systematic floating point errors).

Parameters:

Var (Real)

An Open GENIE real value.

RESULT = (Integer)

An Open GENIE integer or undefined value if the number cannot be converted successfully. For example if the real number is larger than the representation available for the integer.

Called_as_function()

Test if a procedure was called as a function, i.e. x = command(y)

CALLED_AS_FUNCTION() Test whether a procedure should alter its parameters

example:

```
# Procedure which checks how it is called
PROCEDURE double
PARAMETERS w=workspace
RESULT wres
          IF Called_as_function()       # leave the parameter untouched
                wres = w
                wres.y = w.y * 2.0
          ELSE                # change the parameter
                w.y = w.y * 2.0
          ENDIF
ENDPROCEDURE
```

Called_as_function

This procedure is very important for a language which allows procedures to be called either as functions returning results or as keyword commands where the parameters are affected. For example, with the *Rebin()* command, it may be called either as

```
w = rebin( s(1), ... )
```

where the last thing we would want to do is replace the value "s(1)" with the result of the rebinning. Alternatively we could type

```
Rebin w ...
```

where of course, we would expect to change the contents of workspace "w".

If this check is not made a sequence of functions could produce unexpected results, for example

```
>> x = bad_rebin(w, v.x)
>> y = bad_rebin(w, v.x)
>> printn (x=y)
$FALSE
```

Parameters:

RESULT = (Boolean)

Returns $TRUE if the currently executing procedure was called as a function.

Defined()

Tests whether a variable has a defined value (not undefined).

DEFINED() **var**=*Any* Returns $TRUE if var was defined

example:

```
# Check for arithmetic undefined values
>> x = 5.5 / 0.0
>> printn Defined(x)
$FALSE
```

Note: Useful for testing the presence or absence of procedure parameters.

Defined

Tests an Open GENIE variable to see if its value is undefined. This may happen either as the result of an arithmetic expression which returns an undefined value or because a value has not yet been given to a variable. The results of procedures that terminate with an error are also generally set to undefined.

One of the major uses of this test is to check whether procedure parameters have been supplied by the user. Any parameters which have not been supplied will take an undefined value and can be checked and maybe supplied with a default value if required.

Parameters:

Var (Any)

Variable to test for being defined.

RESULT = (Boolean)

Returns $TRUE if the variable has a defined value.

Free()

Deletes a variable or workspace field and frees the storage.

FREE	**variable**=*String* [**subvariable**=*String*]	Delete a variable
[/PROC]		Delete a procedure definition
[/TYPE]		Delete a user defined workspace type

example:

```
# Delete a workspace field "test" and then an array "arr"
>> printn w
  Workspace []
  (
    test = 45.0
  )
>> free "w" "test"
  Workspace []
  (
  )
>> free "arr"
```

Free

This command is provided to allow the user a little more control over the disposal of memory used by large variables. If for example, a large data array is known to be unused it would normally be automatically removed from memory eventually by Open GENIE. To remove it explicitly may be useful, for example, if the user knows they are about to create another large array.

Parameters:

/Proc

Interpret the given string as procedure name.

/Type

Interpret the given string as user defined type name.

Variable (String)

A string giving the name of the item to be deleted - must be double quoted.

Subvariable (String)

For workspaces only, this parameter contains a string which is the name of the field to delete from the workspace.

Inquire()

Prompts the user for terminal input and reads back an input variable.

INQUIRE() [**aprompt**=*String*] Inquire information from the user interactively

example:

```
# Read in a lower limit from the user
>> llimit = inquire("Enter lower limit")
Enter lower limit: 45.678
>> printn llimit
45.6779999999999
>> printn Is_a(llimit, "Real")
$TRUE
```

Note: To read a string without any conversions or prompting use Read_terminal()

Inquire

This command reads input from the users terminal. It also supplies a prompt string to obtain input from the user. The Inquire() command attempts to read the input variable back into the correct type of Open GENIE internal variable. Currently, this is only the case for single Integer or Real values. Any unrecognized type is returned as a string.

Parameters:

Aprompt (String) [default = " :"]

A string which tells the user what information is required, a ":" is appended to the prompt.

RESULT = (Any)

Either a string or the value of the variable as a Real or Integer. If no input is given "" is returned.

Interactive()

Tests whether code is running as part of a procedure or not.

INTERACTIVE() Returns $TRUE if it is run within a procedure

example:

```
# Only print date if this is code is running
# outside of a procedure (in a .gcl file for example)
IF NOT Interactive()
          printn "Running as batch file at " Now()
ENDIF
```

Note: Useful for code fragments which may be used in different ways

---

Interactive

The Interactive() command checks to see which of two possible modes a piece of Open GENIE code is executing in.

1. As a line inside an Open GENIE procedure

2. As a line running at the terminal or outside a procedure but within a loading GCL file.

Parameters:

RESULT = (Boolean)

Returns $TRUE if the command was run outside a procedure.

Is_a()

Tests the type of an Open GENIE variable.

IS_A()	**var**=*Any* **typestring**=*String*	Checks to see what type an Open GENIE variable is

example:

```
# Convert a real if necessary
IF Is_a(x, "real")
    x = As_integer(x)
ENDIF
```

Note: Important when using user defined workspace types.

Is_a

The format of this command appears a little strange at first however it is designed to be easily used in the sort of programming situations where type checking is common, see the example above. It is possible to test a variable for being any Open GENIE predefined type or any user defined workspace type.

A generic type may also be checked for, for example

```
>> printn Is_a(y, "Array")
$TRUE
>> printn Is_a(y, "Realarray")
$FALSE
```

Parameters:

Var (Any)

The variable to be checked

Typestring (String)

The name of the type being checked for.

RESULT = (Boolean)

Returns $TRUE if the variable was of the type specified.

Now()

Returns the current date and time as a string.

NOW()	Returns date and time

example:

```
# Printn the date and time
>> printn Now()
11-Jul-97 8:47:50
```

Note: Use Substring() and Locate() to select time or date individually

Now

The Now() command returns out a simple time and date string. If time or date are required individually they can be chopped out of the string using the *Substring()* and *Locate()* commands to take the date before the space or the time after the space.

Parameters:

RESULT = (String)

Combined time and date string.

Read_terminal()

Read a string back from the input terminal into a variable

READ_TERMINAL()   Return a string typed in at the terminal

example:

```
# Read a string
>> printn "(" read_terminal() ")"
abcd efg
(abcd efg)
>>
```

Note: To read numbers as well as strings you may wish to use the Inquire() command.

Read_terminal()

This command reads a string (terminated by a carriage return) from the terminal. A numeric string can be converted by using *As_variable()* on the string returned by Read_terminal(). This is the way the *Inquire()* command is written.

Parameters:

RESULT = (String)

The string read from the terminal, not including the return character which terminated the input.

Chapter 11

System Dependent Functions

These functions are listed separately as they all have some dependency on the operating system Open GENIE is running. By careful use of the *Os()* command, procedures can be written to work system independently.

Cd Change the default directory from within Open GENIE

Dir List the files in the current directory or the directory specified

Pwd Print the current working directory

Os Returns a string giving the operating system Open GENIE is running on

System Executes a single command or command session from within Open GENIE

System Dependent Function Reference

Cd()

Change the default directory from within Open GENIE

CD [**path**=*String*] Change the working directory

example:

```
# Change to the examples directory
>> cd "/usr/local/genie/examples"
>> dir
```

Cd

Select the directory which Open GENIE will read and write files to by default. This command avoids the need to exit Open GENIE to change directory.

Parameters:

Path (String)

Directory path as specified on the native operating system, e.g. VMS or Unix.

Dir()

List the files in the current directory or the directory specified

DIR [**path**=*String*] List the directory contents

example:

```
# List the examples directory
>> dir "[.examples]"
```

Dir

Display the contents of a directory on the host operating system.

Parameters:

Path (String)

Directory path as specified on the native operating system, e.g. VMS or Unix.

Pwd()

Print the current working directory

PWD Show the working directory

example:

```
# Change to the examples directory
>> cd "/usr/local/genie/examples"
>> pwd
/usr/local/genie/examples
```

Pwd

Print the current working directory if called as a keyword command or return the directory as a string if called as a function.

Parameters:

RESULT (String)

String giving the current working directory.

Os()

Returns a string giving the operating system Open GENIE is running on

OS()	Returns the name of the operating system

example:

```
# Print the current operating system
>> printn Os()
OSF
```

Os

When writing portable GCL programs is is sometimes necessary to know which operating system is being used. For example if VMS is being used, a disk name must be specified whereas none is needed on Unix. This command is provided to allow a procedure to check for differences which might occur and need to be handled.

Parameters:

RESULT = (String)

Currently returns one of the strings "OSF", "VMS", "IRIX" or "LINUX" depending on the supported operating systems.

System()

Executes a single command or command session from within Open GENIE.

SYSTEM	[**command**=*String*]	Execute system commands from within Open GENIE

example:

```
# find some user details on VMS
>> system "$ SEARCH journal.txt \"flux\" "
IRS14142ZAB/NJR Flux tests  18-FEB-1997 12:00:01 163.2
IRS14143ZAB/NJR Flux tests  18-FEB-1997 13:43:40   6.1
>>
```

Note: The Dir() command in Open GENIE is simply a procedure written using the System() command.

System

The System() command allows a user to execute native operating system commands from within Open GENIE. This can be very useful for accessing facilities not actually built into Open GENIE. To make a procedure Operating system independent, use the *Os()* command to choose which system command to use, see the example below.

```
PROCEDURE TIME
IF Os() = "VMS"
    system "show time"
ELSE   # assume Unix
    system "date"
ENDIF
ENDPROCEDURE
```

If no command is given as a parameter, the System() command starts a sub-shell process which must be terminated with an "exit" on Unix or a "LOGOUT" on VMS. This is a handy form of the command if you just want to type a few commands without closing down your Open GENIE session. Note that on VMS some normal logical name definitions may be missing in the sub process.

Parameters:

Command (String)

The command to be executed by the operating system.

RESULT = (Integer)

The execution status returned by the command.

Chapter 12

Miscellaneous commands

This section lists command which are primarily designed to be used interactively for the control of an Open GENIE terminal session.

Copying	Prints the GNU licensing conditions for Open GENIE
Exit	Exits from Open GENIE
Help	Prints out a general Open GENIE help message
Load	Loads a file containing GCL procedures into GENIE
Save	Saves a complete Open GENIE image file with all variables and procedures.
Warranty	Prints the warranty disclaimer for Open GENIE

Miscellaneous Command Reference

Copying()

Prints the GNU licensing conditions for Open GENIE

COPYING	Print the distribution license

Copying()

Prints a copy of the licensing conditions for distribution/re-distribution of Open GENIE under the *GNU General Public licence*.

Exit()

Exits from Open GENIE

EXIT	Stops Open GENIE

example:

```
# Finish an open genie session
>> Exit
GENIE exiting
$
```

Note: Can also use quit to stop Open GENIE

Exit

Stops an Open GENIE session from the command line.

Help()

Prints out a general Open GENIE help message.

HELP	Prints a help message

Note: Help/Help is not a useful command.

Help

Provides a few tips as to how to find Open GENIE commands and how to get help on them.

Load()

Loads a file containing GCL procedures into GENIE

LOAD **filename**=*String* Load a command file

example:

```
# Load a genie program
>> Load "myprog.gcl"
Loading file myprog.gcl ........
        56 lines scanned/compiled
>>
```

Note: Cannot always load one command file from within another.

Load

The load command allows a file consisting of Open GENIE commands and procedure definitions to be read by Open GENIE. It is normally possible to have one file containing "Load" commands for a few other component files but some care should be taken as the Load() command is primarily an interactive command and some odd effects may occur as a result of multiple nesting or calling of Load() within a procedure (not recommended).

Parameters:

Filename (String)

File consisting of valid GCL statements and PROCEDURE definitions.

RESULT = (Boolean)

Returns $FALSE if the load command failed to work.

Save()

Saves a complete Open GENIE image file with all variables and procedures.

SAVE **filename**=*String* Save a frozen image of the current Open GENIE

example:

```
# Save the current session
>> Save "mygenie.im"
Saving GENIE in file mygenie.im ...
>>
```

Note: Saving an image after loading a large number of procedures can give a quicker startup.

Save

The save command saves a binary image of the complete internal state of Open GENIE. It is most useful when quite a large number of GCL procedures have been loaded as part of a major program. To save time on startup, use the Save() command to create an image immediately after starting Open GENIE and loading the necessary GCL files.

Open GENIE can be re-invoked with the new image containing all the procedures by the command

$ genie "" mygenie.im

or by defining the symbol or logical name GENIE_SMALLTALK_IMAGE to point to the file. Note that if an image is not found, Open GENIE will usually start up with some errors.

Parameters:

Filename (String)

File to save the binary image in, conventionally ending with ".im"

Warranty()

Prints the warranty disclaimer for Open GENIE

WARRANTY	Print the warranty disclaimer

Warranty()

Prints a copy of the warranty disclaimer for Open GENIE.

Chapter 13

Diagnostics and Debugging

This section lists commands specifically to do with diagnostics and debugging

Debug()	Enables simple procedure call debugging
Gripe	Report a problem, solution or idea about Open GENIE
Inspect	Inspect in some detail, the type and contents of Open GENIE variables.
Version	Prints the Open GENIE Version.

Diagnostics and Debugging Command Reference

Debug()

Enables simple procedure call debugging

DEBUG/ON	Enable debugging of procedure calls
DEBUG/OFF	Switch off debugging messages

example:

```
# Switch debugging on for next statement
>> DEBUG/ON
>> printn "HI"
0:PRINTN
hi
```

Debug

This procedure switches on rudimentary procedure debugging. For each procedure that is called, a line giving the name of each procedure is printed on the screen. The level to which this line is indented shows how far down in the program the procedure is called. Although not ideal, this can give you a good idea of which procedure has caused an error message.

Gripe()

Report a problem, solution or idea about Open GENIE

GRIPE	[**subject**=*String*] [**message**=*String*] [**email**=*String*]	Send a message to the Open GENIE development team.

example:

```
# Type in a gripe interactively
>> Gripe
  ... fill in details as requested ...
```

Note: email defaults to genie@isise.rl.ac.uk, the normal support mail address for Open GENIE.

Gripe

This command simplifies the reporting of problems, ideas or solutions to problems you have found when using Open GENIE. Just typing "gripe" should be sufficient to get an error message directly to the development team and management.

Parameters:

Subject (String)

Useful if you can give a succinct subject line so we can categorise any gripes easily

Message (String)

One line gripes are very acceptable, and this makes them possible.

Email (String)

the Gripe() command will attempt to send all gripes to *genie@isise.rl.ac.uk* but if your mailer has to cross a firewall or gateway system to get onto the internet you can put the appropriate address here to make the gripe command work.

Inspect()

Inspect in some detail, the type and contents of Open GENIE variables.

INSPECT **p1**=*Any* Print the internal type or representation of a variable

example:

```
# Example in preformatted font
>> inspect 45
An instance of Integer
```

Inspect

The Inspect() command gives details of the type and structure of the underlying smalltalk variables. Sometimes it can be useful to get information about the type of a complex variable but usually one of the *Print()* commands is a better bet.

Parameters:

P1 (Any)

Variable to be inspected.

Version()

Prints the Open GENIE Version.

VERSION Print out the version string

example:

```
# print out the genie version
>> Version
@(#)Open GENIE V1.1 BUILD-30
[Linked] Thu Jul 10 13:47:02 BST 1997 [library version] 1.1
```

Note: Please quote this when reporting problems with the Gripe() command.

Version

Prints out the Open GENIE version string (the same string printed out on startup of Open GENIE).

Chapter 14

External Programming Interfaces

This section describes ways in which Open GENIE may be interfaced to externally written code in FORTRAN or C++ for a variety of purposes.

External software may be either called from Open GENIE whilst Open GENIE maintains control of the session and is responsible for handling any exception conditions which might arise. Alternatively, external code may call into specific parts of Open GENIE in more of the fashion of a subroutine library. Currently there is one defined interface of each sort in Open GENIE.

Callout Interfaces

Open GENIE supports a simple callout interface which allows user FORTRAN code to be called from within an Open GENIE session and provides a mechanism for providing the code with data from Open GENIE internal variables.

FORTRAN module interface
C module interface

Callin Interfaces

Open GENIE supports a callin interface to allow any user program to use the Open GENIE Data Access Interface (GDAI). This interface provides a multi-language way of getting at the same data as Open GENIE.

GDAI interface

Module Subroutines Callable from FORTRAN

This section describes the user written FORTRAN interface which allows modules in FORTRAN to be compiled and loaded into Open GENIE so that they can be run as if they had been built into the original code. Once loaded, a module can be called as efficiently as any other hard coded function from Open GENIE (see the *Module()* command). The function will run in a "safe" environment so that if it should crash, control will return to Open GENIE without causing a crash of Open GENIE itself.

Here we give reference information for:

1. *FORTRAN template* for a user written module.
2. *Helper functions* to aid communication of the module with Open GENIE.

For more general information and examples of using modules, see the *Open GENIE User Manual*

FORTRAN Template

The essential purpose of this interface is to allow data transfer and communication with the running Open GENIE session from which the external FORTRAN module was called. All data transfer is handled by allowing the *Module()* command in GCL to take a workspace of parameters which are made available to the FORTRAN SUBROUTINEs if they adhere to a certain template. The workspace itself, is an advanced data type and as such is not easily handled itself in FORTRAN, what is provided in the FORTRAN interface is a set of *Helper Functions* which allow data to be read and written to and from this parameter workspace.

The FORTRAN template is actually very simple and is given below. A normal procedure for converting an existing program is to turn the program into one or more subroutines taking appropriate data as parameters and then to call these subroutines from the FORTRAN template subroutine to perform the appropriate functions. It may be necessary to separate the functional part of the program from any subroutines that query a user interactively for data, these subroutines will probably need replacing with helper functions (or they may more easily be done from GCL and/or the TK graphical interface anyway).

```
C An F77 template subroutine for creating an Open GENIE module in FORTRAN
C Several subroutines like the one routine below may be included in one
C module.
C
C Called from GCL with e.g. MODULE:EXECUTE("my_fortran_module", PARS)
C
C Freddie Akeroyd, ISIS, 20/2/97 - modified by CM-S 2/7/97
C
      SUBROUTINE MY_FORTRAN_MODULE(PARS_GET, PARS_PUT)
      IMPLICIT NONE

C Include mandatory definitions

      INCLUDE 'genie_modules.inc'
      EXTERNAL PARS_GET, PARS_PUT

C ... User declarations here ...

C Check we have a recent enough version of the genie library -
C all modules should do this to avoid unexpected crashes

      IF (MODULE_VERSION_OK(GENIE_MAJOR, GENIE_MINOR) .EQ. 0) THEN
          MODULE_ERROR("my_fortran_module",
     +                 "Error - Open GENIE Version mismatch",
     +                 "Please re-compile this module")
      ENDIF

C All user code goes here, generally the format will be to access some
C data using the helper functions, process it via new or existing user
C written subroutines, and finally return any modified parameters using the
C helper functions

      RETURN
      END
```

Helper Functions

There are three sorts of helper functions provided by the FORTRAN module interface. They are all listed below

Commands to obtain data from Open GENIE	**Description**
MODULE_GET_DOUBLE	Reads a double precision real from the PARS workspace
MODULE_GET_INT	Reads an integer from the PARS workspace
MODULE_GET_REAL	Reads a real from the PARS workspace
MODULE_GET_STRING	Reads a string from the PARS workspace
MODULE_GET_DOUBLE_ARRAY	Reads a double precision real array from the PARS workspace
MODULE_GET_INT_ARRAY	Reads an integer array from the PARS workspace
MODULE_GET_REAL_ARRAY	Reads a real array from the PARS workspace

MODULE_GET_STRING_ARRAY	Reads a string array from the PARS workspace

Commands to return data to Open GENIE	**Description**
MODULE_PUT_DOUBLE	Replaces a double precision real in the PARS workspace
MODULE_PUT_INT	Replaces an integer in the PARS workspace
MODULE_PUT_REAL	Replaces a real in the PARS workspace
MODULE_PUT_STRING	Replaces a string in the PARS workspace
MODULE_PUT_DOUBLE_ARRAY	Replaces an double precision real array in the PARS workspace
MODULE_PUT_INT_ARRAY	Replaces an integer array in the PARS workspace
MODULE_PUT_REAL_ARRAY	Replaces an real array in the PARS workspace
MODULE_PUT_STRING_ARRAY	Replaces an string array in the PARS workspace
MODULE_PUT_ND_REAL_ARRAY	Replaces an n-dimensional real array in the PARS workspace

Commands to communicate with the user	**Description**
MODULE_PRINT	Prints to the terminal
MODULE_INFORMATION	Prints to the terminal as an informational message (in blue)
MODULE_ERROR	Prints a formatted error message

FORTRAN HELPER FUNCTION REFERENCE

Commands to Obtain Data from Open Genie

All commands to read data from Genie into a user program start with the "module_get_" prefix, and have their first two parameters in common. The first parameter, called PARS_GET, is an EXTERNAL entity passed by Genie. The second parameter, NAME, is a character variable which is the label assigned to the variable in GCL when the parameters workspace was created. For arrays, a LENGTH variable must also be passed; on input this should be set to the maximum allowed number of points, and on output it will be set to the number actually read. If too many points are passed out from GCL, LENGTH will still indicate how many were passed, but the number actually read will be the length of the array which was passed as input; thus testing LENGTH on return provides a check for array overflow.

There is no need for MODULE_**GET**_ND_REAL_ARRAY to correspond to the MODULE_**PUT**_ND_REAL_ARRAY function because an array can always be converted to 1-D using the *Redim()* command from GCL before passing it out.

The subroutine names and parameter types and names are given below:

```
MODULE_GET_DOUBLE          (EXTERNAL PARS_GET, CHARACTER*80 NAME, REAL*8 VALUE)

MODULE_GET_INT             (EXTERNAL PARS_GET, CHARACTER*80 NAME, INTEGER VALUE)

MODULE_GET_REAL            (EXTERNAL PARS_GET, CHARACTER*80 NAME, REAL VALUE)

MODULE_GET_STRING          (EXTERNAL PARS_GET, CHARACTER*80 NAME, CHARACTER*512 VALUE)

MODULE_GET_DOUBLE_ARRAY    (EXTERNAL PARS_GET, CHARACTER*80 NAME, REAL*8 VALUE(*),      INTEGER
                           LENGTH)

MODULE_GET_INT_ARRAY       (EXTERNAL PARS_GET, CHARACTER*80 NAME, INTEGER VALUE(*), INTEGER
                           LENGTH)

MODULE_GET_REAL_ARRAY      (EXTERNAL PARS_GET, CHARACTER*80 NAME, REAL VALUE(*), INTEGER LENGTH)

MODULE_GET_STRING_ARRAY    (EXTERNAL PARS_GET, CHARACTER*80 NAME, CHARACTER*512(*) VALUE, INTEGER
                           LENGTH)
```

Commands to return Data to Open Genie

These commands return data to Open GENIE by replacing the fields in the PARS workspace, otherwise they are very similar to the corresponding "module_get_" commands. The NAME parameter will be the field name in the result workspace returned to GCL, and the LENGTH parameter for the array routines will indicate how many data points to send.

The subroutine names and parameter types and names are given below:

```
MODULE_PUT_DOUBLE          (EXTERNAL PARS_PUT, CHARACTER*80 NAME, REAL*8 VALUE)

MODULE_PUT_INT             (EXTERNAL PARS_PUT, CHARACTER*80 NAME, INTEGER VALUE)

MODULE_PUT_REAL            (EXTERNAL PARS_PUT, CHARACTER*80 NAME, REAL VALUE)

MODULE_PUT_STRING          (EXTERNAL PARS_PUT, CHARACTER*80 NAME, CHARACTER*512 VALUE)

MODULE_PUT_DOUBLE_ARRAY    (EXTERNAL PARS_PUT, CHARACTER*80 NAME, REAL*8 VALUE(*), INTEGER
                           LENGTH)

MODULE_PUT_INT_ARRAY       (EXTERNAL PARS_PUT, CHARACTER*80 NAME, INTEGER VALUE(*), INTEGER
                           LENGTH)

MODULE_PUT_REAL_ARRAY      (EXTERNAL PARS_PUT, CHARACTER*80 NAME, REAL VALUE(*), INTEGER LENGTH)

MODULE_PUT_STRING_ARRAY    (EXTERNAL PARS_PUT, CHARACTER*80 NAME, CHARACTER*512(*) VALUE, INTEGER
                           LENGTH)

MODULE_PUT_ND_REAL_ARRAY   (EXTERNAL PARS_PUT, CHARACTER*80 NAME, REAL VALUE(*),
                           INTEGER DIMS_ARRAY(NDIMS), INTEGER NDIMS)
```

Commands to Communicate With The User

These provide a means for a FORTRAN program to send messages to the user through the standard Open GENIE messaging system. This means, for example that informational messages can be switched off using the *Toggle/Info* command as with ordinary Open GENIE informational messages. Using these routines ensures that the timing of output messages will be properly synchronized with Open GENIE.

The subroutine names and parameter types and names are given below:

```
MODULE_PRINT               (CHARACTER*(*) MESSAGE)

MODULE_INFORMATION         (CHARACTER*(*) MESSAGE)

MODULE_ERROR               (CHARACTER*(*) FUNCTION_NAME, ERROR_MESSAGE, POSSIBLE_SOLUTION)
```

Module Subroutines Callable from C

This section describes the user written C interface which allows modules in C to be compiled and loaded into Open GENIE so that they can be run as if they had been built into the original code. Once loaded, a module can be called as efficiently as any other hard coded function from Open GENIE (see the *Module()* command). The function will run in a "safe" environment so that if it should crash, control will return to Open GENIE without causing a crash of Open GENIE itself.

Here we give reference information for:

1. *C template* for a user written module.
2. *Helper functions* to aid communication of the module with Open GENIE.

For more general information and examples of using modules, see the *Open GENIE User Manual*

C Template

The essential purpose of this interface is to allow data transfer and communication with the running Open GENIE session from which the external C module was called. All data transfer is handled by allowing the *Module()* command in GCL to take a workspace of parameters which are made available to C functions if they adhere to a certain template. To access parts of the workspace in the C interface a set of *Helper Functions* which allow data to be read and written to and from this parameter workspace are provided.

The C template is actually very simple and is given below. A normal procedure for converting an existing C program is to take the existing analysis functions and to call them from one or more functions following the C template below. User interface functions doing output can be replaced with the print helper functions (or they may more easily be done from GCL and/or the TK graphical interface anyway once the module is in Open GENIE).

```
/* A C template function for creating an Open GENIE module in C
 * Several functions like the one routine below may be included in one
 * module.
 * Called from GCL with e.g. MODULE:EXECUTE:C("my_c_module", PARS)
 *
 * Freddie Akeroyd, ISIS, 19/2/97 - modified by CM-S 2/7/97
 */
/*
 * These two genie includes must be present in any C module
 * They include important definitions and typedefs
 */
#include <genie_cmodule.h>
#include <genie_cmodule_ver.h>

void my_c_module(GenieWorkspace* pars_get, GenieWorkspace* pars_put)
{
/* ... User definitions here ... */
/*
 * All C modules should do the following test - it ensures that a module
 * linked against a recent version of the genie library is
 * not executed by a program linked against an older version
 */
    if (!cmodule_version_ok(CMODULE_MAJOR_VERSION, CMODULE_MINOR_VERSION))
    {
          cmodule_print("Library version mismatch for MY_C_MODULE",
                           CMODULE_PRINT_ERROR);
          return;
    }
/* All user code goes here, generally the format will be to access some
 * data using the helper functions, process it via new or existing user
 * written functions, and finally return any modified parameters using the
 * helper functions
 */
}
```

Helper Functions

There are three sorts of helper functions provided by the C module interface. They are all listed below

Commands to obtain data from Open GENIE	**Description**
cmodule_get_double	Reads a double precision real from the PARS workspace
cmodule_get_int	Reads an integer from the PARS workspace
cmodule_get_real	Reads a real from the PARS workspace
cmodule_get_string	Reads a string from the PARS workspace
cmodule_get_double_array	Reads a double precision real array from the PARS workspace
cmodule_get_int_array	Reads an integer array from the PARS workspace
cmodule_get_real_array	Reads a real array from the PARS workspace

cmodule_get_string_array	Reads a string array from the PARS workspace

Commands to return data to Open GENIE	**Description**
cmodule_put_double	Replaces a double precision real in the PARS workspace
cmodule_put_int	Replaces an integer in the PARS workspace
cmodule_put_real	Replaces a real in the PARS workspace
cmodule_put_string	Replaces a string in the PARS workspace
cmodule_put_double_array	Replaces an double precision real array in the PARS workspace
cmodule_put_int_array	Replaces an integer array in the PARS workspace
cmodule_put_real_array	Replaces an real array in the PARS workspace
cmodule_put_string_array	Replaces an string array in the PARS workspace
cmodule_put_nd_real_array	Replaces an n-dimensional real array in the PARS workspace

Commands to communicate with the user	**Description**
cmodule_print	Prints to the terminal

C HELPER FUNCTION REFERENCE

Commands to Obtain Data from Open Genie

All commands to read data from Genie into a user program start with the "cmodule_get_" prefix, and have their first two parameters in common. The first parameter, called "pars_get", is an pointer to the workspace passed by Genie. The second parameter, "name", is a character variable which is the workspace field name assigned to the variable in GCL when the parameters workspace was created. For arrays, a "len" variable must also be passed; on input this should be set to the maximum allowed number of points, and on output it will be set to the number actually read. If too many points are passed out from GCL, "len" will still indicate how many were passed, but the number actually read will be the length of the array which was passed as input; thus testing "len" on return provides a check for array overflow.

Because C is able to cope with memory allocation, it is possible for the cmodule_get functions to allocate memory if required. You will of course be responsible for calling free() afterwards if this is the case! To get the functions to allocate memory pass a pointer variable containing NULL as the destination for the data and sufficient memory to hold the requested item will be allocated (actual length allocated returned in the "len" parameter where applicable). Note that functions returning strings will *always* allocate memory and hence you are responsible for calling free().

There is no need for cmodule**_get_**nd_real_array to correspond to the cmodule**_put_**nd_real_array function because an array can always be converted to 1-D using the *Redim()* command from GCL before passing it out.

The function prototypes are given below:

```
void cmodule_get_double (GenieWorkspace* pars_get, const char* name, fort_double* val);

void cmodule_get_int    (GenieWorkspace* pars_get, const char* name, fort_int* val);

void cmodule_get_real   (GenieWorkspace* pars_get, const char* name, fort_real* val);

void cmodule_get_string (GenieWorkspace* pars_get, const char* name, char** val);

void cmodule_get_double_array (GenieWorkspace* pars_get, const char* name, fort_double** val,
                           fort_int* len);

void cmodule_get_int_array    (GenieWorkspace* pars_get, const char* name, fort_int** val,
                           fort_int* len);
```

```
void cmodule_get_real_array   (GenieWorkspace* pars_get, const char* name, fort_real** val,
                           fort_int* len);

void cmodule_get_string_array (GenieWorkspace* pars_put, const char* name, const char** val[],
                           fort_int* len);
```

Commands to return Data to Open Genie

These commands return data to Open GENIE by replacing the fields in the "pars_put" workspace, otherwise they are very similar to the corresponding "cmodule_get_" commands. The "name" parameter will be the field name in the result workspace returned to GCL, and the "len" parameter for the array routines will indicate how many data points to send.

The function prototypes are given below:

```
void cmodule_put_double (GenieWorkspace* pars_put, const char* name, fort_double val);

void cmodule_put_int    (GenieWorkspace* pars_put, const char* name, fort_int val);

void cmodule_put_real   (GenieWorkspace* pars_put, const char* name, fort_real val);

void cmodule_put_string (GenieWorkspace* pars_put, const char* name, const char* val);

void cmodule_put_double_array (GenieWorkspace* pars_put, const char* name, const fort_double*
                           val, fort_int len);

void cmodule_put_int_array    (GenieWorkspace* pars_put, const char* name, const fort_int* val,
                           fort_int len);

void cmodule_put_real_array   (GenieWorkspace* pars_put, const char* name, const fort_real*
                           val,fort_int len);

void cmodule_put_string_array (GenieWorkspace* pars_put, const char* name, const char* val[],
                           fort_int len);

fort_int cmodule_put_nd_real_array (GenieWorkspace* pars_put, const char* name, const fort_real*
                           val,
                                  const fort_int* dims_array, fort_int ndims);
```

Commands to Communiate With The User

These provide a means for a C program to send messages to the user through the standard Open GENIE messaging system. This means, for example that informational messages can be switched off using the *Toggle/Info* command as with ordinary Open GENIE informational messages. Using these routines ensures that the timing of output messages will be properly synchronized with Open GENIE.

The C output helper functionality is combined into one function "cmodule_print". The "cmodule_print" function takes one of three constants, CMODULE_PRINT_NORMAL, CMODULE_PRINT_INFORMATION or CMODULE_PRINT_ERROR as the "option" parameter to select the corresponding Open GENIE I/O stream to print to.

The subroutine names and parameter types and names are given below:

```
void cmodule_print (const char* s, fort_int option);
```

Open GENIE Data Access Interface (new get)

This document describes a set of routines designed to create, populate, read and modify ISIS format and NeXus format data files. The aim of these routines is to provide one simple way to read and write all styles of data files. The routines are based on the routines used in Open GENIE and will therefore cope with automatic detection of file input types for all Open GENIE supported file formats.

Basic Principles

Any data interface design is a compromise between flexibility and complexity. The interface below is intended to be simple to use from C, C++ or FORTRAN programs but does make use of some of the most advanced parts of the Open GENIE data access subsystem to do this. As a result, it is worth being aware of how data is actually read or written by the interface.

File conversions

When reading a file on different machine types, there are inherently several different conversion problems which may arise. Some are due to differences in the structure of the data files, others may be due to differences in numerical precision or binary format. As a result, the access routines work in two stages.

For reading:

1. Access and convert the desired data block from the file.

2. Select the data from the block to read into the program.

For writing:

1. Construct a data block to be written to the file.

2. Write and convert the data to the file.

In both cases, this is a very simple operation if the structure of the data file is simple. On the other hand, if the structure of the data file is complex, the data access routines provide a unique and powerful data structure parsing mechanism which allows complex structures to be easily snipped up into manageable (if not byte sized!) pieces.

Handles

A "Handle" is the name given to the data block which has been converted from the file but has not yet been loaded into your program. It is simply a user chosen name to identify the block of data in transition between stages (1) and (2) above, effectively the data enters a holding area where you can then make detailed requests of it. For example, if we read one experimental parameter from an ISIS raw file, "NSP1", the fragment of the program using handles would look something like the one below.

```
        ...
* Access the data item by name and put in the transfer area
    CALL GX_get('MY_HANDLE', 'NSP1', 0)

* Read the value into the INTEGER variable "my_nsp"
    CALL GX_transfer('MY_HANDLE', '-->', 'INTEGER', my_nsp, 0, 0)
        ...
```

The handle name links the request to get the data with the subsequent request to load the data into a program variable.

A handle can be re-used as often as required or several different handles may be used to allow to store data blocks before reading them out into the code. A handle expression can also be used to select a small amount of data to read into the program from a larger data block.

Let's assume now that we want to read the first "MAXLEN" elements of the detector to spectrum mapping table and write it out to a new file. We know the name of the data item in the ISIS raw file and can access it as below.

```
* Somewhere to put the data we are about to read
    INTEGER*4 my_tab(42)
        ...
* Get the whole data block to the handle.
    CALL GX_get('MY_HANDLE', 'SPEC', 0)

* Select part of the data to read out from the handle.
    CALL GX_transfer('MY_HANDLE[1:42]', '-->', 'INTEGER', my_tab, 42, 1)
        ...
```

This way we can slice the first 42 elements out of the array (assuming it has at least 42 elements). This example could have just specified a single array element if only one value was required. In some file formats, the block may contain named fields or attributes. These can be accessed using the *syntax for accessing workspaces*, for example "My_Handle.Two_Theta".

Data access routines descriptions

Before reading or writing data, a session must be initialised using GX_activate_session(), at the end of data access GX_deactivate_session() session should also be called. These routines control the allocation and deallocation of memory as well as setting the default output file format.

The interface makes the general assumption that a user may well want to have one file open for writing at the same time as having a different file open for reading. This is achieved by having both GX_select_source() and GX_select_destination routines(), either or both of these calls may be used in a data access session. The GX_directory() function allows the calling program to find out what data items are available for a program to read in the specified file.

The following is a generic description of the Application Programming Interface (API) and is divided into sections grouping similar functions. Language specific routine calls are given in the reference documentation for each routine.

Session Control

Function	***Return value***	***Description***
GX_activate_session(**in** default_format)	status code	Initialises the interface selecting a default data format for output.
GX_deactivate_session()	status code	Deactivates the interface, frees storage

File operations

Function	***Return value***	***Description***
GX_select_source(**in** filename)	status code	Selects a data source for subsequent operations
GX_select_destination(**in** filename, **in** file_format)	status code	Selects a data destination for subsequent operations
GX_directory(**in** filename, [**out** dir_listing])	status code	Returns a directory of available fields in any data file.

Data reading/writing

Function	***Return value***	***Description***
GX_get(**in** handle, **in** tag, **in** object-id)	status code	Associates a handle with the referenced data file object.
GX_put(**in** handle, **in** tag, **in** object-id, **in** flag)	status code	Writes data associated with a handle into a data file.

Handle manipulation

Function	***Return value***	***Description***
GX_assign_handle(**in** handle_lval, **in** handle_rval)	status code	Assign one handle to the whole or part of another handle.
GX_release_handle(**in** handle)	status code	Deactivate the handle and release and storage.
GX_transfer(**in** handle, **in** direction, **in** type, **inout** data, **in** length, **in** dims)	status code	Transfer data to or from variables in the program.

NOTES:

1. References to an object in the data file can be made either by a string tag or an object-id (number) for the data file. For a GX_get, if both are specified, the tag must correspond to the tag on the object selected by the object-id. Objects may also be accessed by block number if the object-id is negative (e.g. -1 for block 1).

2. For a GX_put, the object at the object-id specified may be overwritten with the new tag and associated handle value. If the "flag" parameter to GX_put is set to "OVERWRITE", when the object-id corresponds with that already in an output file, the object and tag will be overwritten.

Handles are always strings and are marked as "in" parameters. It is worth pointing out that although the handle is effectively a constant reference, what it points to acts like a variable.

Chapter 15

Workspace Operations

Workspace operations and transformations form the heart of Open GENIE. They permit analysis which takes into the account the underlying model of the data from neutron scattering experiments. For example, the *Units()* command can be used to convert the units of a Time-of-Flight (TOF) spectrum.

This level of knowledge about the data being operated on is only possible if some assumptions are made about the fields which will be present in any workspace containing experimental data. This in turn requires agreement about the contents and names of fields for each kind of data.

By default, Open GENIE workspaces are treated as an enhanced version of the original GENIE-V2 workspace which was based on a one dimensional Time-of-Flight spectrum. In Open GENIE two dimensional workspaces are also supported (i.e. workspaces with a two-dimensional Y-array).

The arithmetic functions (as defined in the *Template Routines* section) require arrays of X, Y, and error values to be present (a basic histogram). For the *Units()* command to work, a workspace also requires fields giving parameters such as the primary flight path and incident angle (a TOF spectrum).

In the next sections three topics are discussed before giving a complete listing of the modifiable template routines.

- *Taxonomy*- How different types of Open GENIE workspaces are classified
- *Steps for the creation of new workspace types*
- *Required Open GENIE Workspace Fields*
- *Template routine listing for all customizable routines*.

Taxonomy

Classification of the basic data models used in the analysis of neutron scattering data is still at a very early stage. Open GENIE is being designed to cope with several detailed classifications when they are developed. So far it only recognises one class of neutron scattering data - the GENIE-V2 style TOF spectrum - but extended to cope with 2-D data. This consists of n-histograms of data and may also contain general annotation information suitable for a plotting and units conversion. In experimental terms a workspace can correspond to a single detector scan (in TOF) or multiple scans as would be collected from a single timed run on a position sensitive detector array (the contents of an ISIS raw data file). Open GENIE is also intended for workspaces with different types of data (e.g. triple axis Qx, Qy, Qz arrays) and it is only necessary to redefine the template routines to make sense of operations such as w1 + w2 appropriately.

The next section briefly describes how to go about adding a new workspace type. Before doing this, some experience of writing Open GENIE procedures in GCL should be gained..

Steps for the Creation of New Workspace Types

When defining a different class of data there is a special syntax for defining a new type of data workspace with default fields. Lets say for example, that we wish to create a workspace for triple axis data where we can directly plot/compare this data with ISIS TOF data. The steps in this process are given below.

1. Create a new GCL file to hold the definitions and load it before performing any operations. Lets say the file is "tripleaxis.gcl".

2. At the top of the file add the new workspace type definition as shown below.

   ```
   # Defines a new workspace type called Tripleaxis
   [ Workspace.(subclass="Tripleaxis", fields="QX QY QZ", comment="Triple axis workspace")
                    ]
   ```

   The string after "subclass" gives the new workspace type name and the fields are specified in a list after "fields" and separated by a single space. This syntax is case sensitive so be sure to use a capital letters where they are used below and nowhere else (anything is OK in the comment string of course).

3. Now add any data specific procedure definitions. For example, printing. This is also the

place to redefine the *S()* procedure if the input data format is different.

```
# Now define any PROCEDURES which operate on these workspaces, for example,
# to fill with data, display, convert etc.
PROCEDURE example_print
PARAMETERS scan=Tripleaxis    # we can now type check for triple axis data
     printn "Qx = " scan.qx  "Qy = " scan.qy  "Qx = " scan.qz
ENDPROCEDURE
```

4. To create new workspace of class triple axis you will need to use one more piece of special Open GENIE syntax

```
>> my_work = Tripleaxis.new()          # note capital T, rest lowercase.
>> printn my_work                      # print out workspace
  Tripleaxis [Triple axis workspace]
  (
    qx = _                   # elements undefined and needing values
    qz = _                   # eg  my_work.qx = ...
    qy = _
  )
```

5. If you wish to re-define or augment the workspace operations +, -, *, / etc. Copy the file "workspace_user.gcl" from the genie area and edit the template routines. You can do this to make them take only "Tripleaxis" data but by being more cunning, you can keep the original functionality as well. eg

```
PROCEDURE Workspace_add; PARAMETERS w1=Workspace w2=Workspace; RESULT=wres
IF is_a(w1, "Tripleaxis")
    wres = my_TA_func(w1, w2)
ELSE
    ... what was here before

ENDPROCEDURE
```

6. From now on, your "Tripleaxis" workspace type is as much a part of Open GENIE as any other workspace. Please note that type names cannot contain "_" and "$" characters, must always be lowercase and need to start with a Capital letter.

If you intend to embark on a project of this sort it is probably worth mailing *genie@isise.rl.ac.uk* to check that this work has not already been done for your type of data.

Required Open GENIE workspace fields

This section lists the required fields and types needed in a workspace to "Qualify" as a Histogram, TOFSpectrum or AnnotatedSpectrum.

The field names and requirements are listed here but the distinctions are not made rigidly with separate workspace types at the moment. For example if you try to add two workspaces without Y arrays you will certainly get an error.

Histogram

Workspace Field	Description	Variable type
X(1-D)	array of X values	RealArray
Y(1-D)	array of Y values	RealArray
E(1-D)	array of errors for Y field	RealArray

2-D Histogram

Workspace Field	Description	Variable type
X(1-D)	array of X values	RealArray
Y(2-D)	2-D array of Y values	RealArray
E(2-D)	2-D array of errors for Y field	RealArray

TOF Spectrum

Workspace Field	Description	Variable type
Histogram +	(all fields in Histogram)	
L1	primary flight path (m)	Real
L2	secondary flight path (m)	Real
Twotheta	scattering angle	Real
Delta	hold off in microseconds	Real
Emode	energy mode	Integer
Efixed	fixed energy (if applicable)	Real
Xlabel	Units for X values	String
Ylabel	Units for Y values	String
Ut(1-D)	User parameters (this array may be of any length and caters for information not already named in a field	RealArray

2-D TOF Spectrum

Workspace Field	Description	Variable type
2DHistogram +	(all fields in 2DHistogram)	
L1	primary flight path (m)	Real
L2(1-D)	secondary flight path (m)	Real
Twotheta(1-D)	scattering angle	Real
Delta	hold off in microseconds	Real
Emode	energy mode	Integer 0=inelastic, 1=incident,

		2=transmitted
Efixed	fixed energy (if applicable)	Real
Xlabel	Units for X values	String
Ylabel	Units for Y values	String
Ut(1-D)	User parameters (this array may be of any length and caters for information not already named in a field)	RealArray

Annotated Spectrum

Workspace Field	**Description**	**Variable type**
TOFSpectrum +	(all fields in TOFspectrum)	
File	File from which data came	String
Title	run title	String
User_name	User name	String
Time	run start date and time	String
Run_duration	run duration in seconds	Real
Spec_no	spectrum number	Integer
Run_no	run number	String
Inst_name	instrument name	String
History	workspace history	String

Annotated 2-D Spectrum

Workspace Field	**Description**	**Variable type**
2DTOFSpectrum +	(all fields in TOFspectrum)	
AnnotatedSpectrum +	(all fields in AnnotatedSpectrum but...)	
Spec_no(1-D)	spectrum number (this is now an array)	Integer

Template Routine Reference

The following functions are called whenever an intrinsic operation involving workspaces is specified (for example w1 + w2 will invoke the template routine Workspace_add () with the workspaces as parameters). The purpose of this is to allow customisation of the routines. These functions can be redefined by a (careful) user to take account of any experiment specific data which may be passed in the workspace.

The example with each function description gives the template procedure currently installed in Open GENIE, the aim of these is to broadly approximate the operations implicit in GENIE-V2 and to give an example of how to write the routine and handle the errors as Open GENIE does.

Customising the templates

All the template routines are kept together in one file "workspace_user.gcl." By modifying all the template routines in the file, the behaviour of the Open GENIE workspace operations may be completely modified. It is important to maintain the default functionality for each procedure, or at least to be aware of routines which may fail to operate when it is changed!

To activate the changes, the modified copy of "workspace_user.gcl" can be loaded into Open GENIE using the Load command. e.g.

```
>> Load "SRC/workspace_user.gcl"
Loading file SRC/workspace_user.gcl
    275 lines scanned/compiled  buffer space remaining = 10958
```

Workspace operations index

This table lists all the Workspace operations which can be *modified*.

Unary operations	**Description**
Workspace_abs	|w|
Workspace_arccos	acos(w)
Workspace_arcsin	asin(w)
Workspace_arctan	atan(w)

Workspace_coerce	Defines how workspaces combine with other types, e.g. w1 * 4.0
Workspace_cos	cos(w)
Workspace_exp	exp(w)
Workspace_ln	ln(w)
Workspace_log	log(w)
Workspace_negated	-w
Workspace_not	NOT w
Workspace_sin	sin(w)
Workspace_sqrt	sqrt(w)
Workspace_tan	tan(w)

Binary Operations

Workspace_add	w1 + w2
Workspace_append	End on join of one histogram to another (w1 & w2)
Workspace_divide	w1 / w2
Workspace_raised_to	w1 ^ w2
Workspace_subtract	w1 - w2
Workspace_modulo	w1 | w2
Workspace_multiply	w1 * w2

Comparison Operations

Workspace_and	w1 AND w2
Workspace_equal	w1 = w2
Workspace_greater_than	w1 > w2
Workspace_greater_than_or_equal	w1 >= w2
Workspace_less_than	w1 < w2
Workspace_less_than_or_equal	w1 <= w2
Workspace_not_equal	w1 != w2

Workspace_or	w1 OR w2

Unary Operations Reference

Workspace_arccos()

Called by the generic function Acos(x) when x is of type Workspace.

WORKSPACE_ARCCOS()	**w1**=*Workspace*	acos(w)

Workspace_arccos

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_arccos; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = arccos(w1.y)
          wres.e = w1.e / sqrt( 1 - w1.y^2 )
ENDPROCEDURE
```

Workspace_arcsin()

Called by the generic function Asin(x) when x is of type Workspace.

WORKSPACE_ARCSIN()	**w1**=*Workspace*	asin(w)

Workspace_arcsin

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_arcsin; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = arcsin(w1.y)
          wres.e = w1.e / sqrt( 1 - w1.y^2 )
ENDPROCEDURE
```

Workspace_arctan()

Called by the generic function Atan(x) when x is of type Workspace.

WORKSPACE_ARCTAN()	**w1**=*Workspace*	atan(w)

Workspace_arctan

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_arctan; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = arctan(w1.y)
          wres.e = w1.e / ( 1 + w1.y^2 ) * w1.e/(1_+ w1.y^2)
ENDPROCEDURE
```

Workspace_coerce()

Called to convert numbers and arrays automatically in to workspaces.

WORKSPACE_COERCE()	**something**=*Any*	convert "something" into a workspace (if possible)

Workspace_coerce

The default procedure definition and the calculation of errors for this procedure is shown below. This routine is called whenever a mixed type expression involving workspaces requires to convert the NON workspace parameter into a workspace. The result will always be a workspace

This is procedure is best not changed without a good understanding of how it works. It is not likely to be necessary to alter this when defining a new type of workspace.

```
PROCEDURE workspace_coerce; PARAMETERS something; RESULT wres

          LOCAL self

          # assign self to be the current workspace object to allow sizing & copying to the new
                           one
          %( gXself _ line at: 1 )%

          wres=fields()                           # new a workspace

          # these cases shouldn’t be able to happen
          IF is_a(something, "workspace") OR is_a(something, "workspacearray")
                    printen "Coercion failure - workspace(array) -> workspace"
          ELSE
                    wres = self                   # copy all workspace fields
                    wres.ylabel = "(dimensionless)"
                    fill wres.e 0.0

                    # use array coercion to fill the workspace
                    wres.y = fill(wres.y, 0.0) + something  ENDIF

ENDPROCEDURE
```

Workspace_cos()

Called by the generic function Cos(x) when x is of type Workspace.

WORKSPACE_COS()	**w1**=*Workspace*	cos(w)

Workspace_cos

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_cos; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = cos(w1.y)
          wres.e = abs( sin(w1.y) * w1.e )
ENDPROCEDURE
```

Workspace_exp()

Called by the generic function Exp(x) when x is of type Workspace.

WORKSPACE_EXP()	**w1**=*Workspace*	exp(w)

Workspace_exp

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_exp; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = exp(w1.y)
          wres.e = w1.e * wres.y
ENDPROCEDURE
```

Workspace_ln()

Called by the generic function ln(x) when x is of type Workspace.

WORKSPACE_LN()	**w1**=*Workspace*	ln(w)

Workspace_ln

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_ln; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = ln(w1.y)
          wres.e = w1.e/w1.y
ENDPROCEDURE
```

Workspace_log()

Called by the generic function log(x) when x is of type Workspace.

WORKSPACE_LOG()	**w1**=*Workspace*	log(w)

Workspace_log

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_log; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = log(w1.y)
          wres.e = w1.e/(w1.y * ln(10.0))
ENDPROCEDURE
```

Workspace_negated()

Called by the generic function -x when x is of type Workspace.

WORKSPACE_NEGATED()	**w1**=*Workspace*	-w

Workspace_negated

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_negated; PARAMETERS w1=workspace; RESULT wres
          wres=w1
          wres.y = -w1.y
ENDPROCEDURE
```

Workspace_not()

Called by the generic function NOT x when x is of type Workspace.

WORKSPACE_NOT()	**w1**=*Workspace*	NOT w

Workspace_not

The default procedure definition and the calculation of errors for this procedure is shown below. This function does not have a useful implementation currently. This is really only included for the sake of completeness as the syntax does allow it to be called.

```
PROCEDURE workspace_not; PARAMETERS w1=workspace; RESULT wres
          printen   "Operation NOT is not defined for workspaces"
ENDPROCEDURE
```

Workspace_sin()

Called by the generic function Sin(x) when x is of type Workspace.

WORKSPACE_SIN()	**w1**=*Workspace*	sin(w)

Workspace_sin

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_sin; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = sin(w1.y)
          wres.e = abs( cos(w1.y) * w1.e )
ENDPROCEDURE
```

Workspace_sqrt()

Called by the generic function Sqrt(x) when x is of type Workspace.

WORKSPACE_SQRT()	**w1**=*Workspace*	sqrt(w)

Workspace_sqrt

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_sqrt; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = sqrt(w1.y)
          wres.e = w1.e / (2.0 * wres.y)
ENDPROCEDURE
```

Workspace_tan()

Called by the generic function Tan(x) when x is of type Workspace.

WORKSPACE_TAN()	**w1**=*Workspace*	tan(w)

Workspace_tan

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_tan; PARAMETERS w1=workspace; RESULT wres
          wres = w1
          wres.y = tan(w1.y)
          wres.e = w1.e * ( 1 + wres.y^2)
ENDPROCEDURE
```

Binary Operations Reference

Workspace_add()

Called by the generic function x + y when x and y are of type Workspace.

WORKSPACE_ADD() **w1**=*Workspace* **w2**=*Workspace* w1 + w2

Workspace_add

The default procedure definition and the calculation of errors for this procedure is shown below. Note that the characteristics of the Left hand workspace are those that define the resultant workspace (i.e. final length, dimensionality of arrays).

```
PROCEDURE workspace_add; PARAMETERS w1=workspace w2=workspace; RESULT wres

          IF (w1.x = w2.x) AND (w1.xlabel = w2.xlabel)
                    wres = w1
                    wres.y = w1.y + w2.y
                    wres.e = sqrt( w1.e*w1.e + w2.e*w2.e )
          ELSE
                    printen "Workspaces are not compatible - please re-bin"
          ENDIF

ENDPROCEDURE
```

Workspace_append()

Called by the generic function x & y (append) when x and y are of type Workspace.

WORKSPACE_APPEND() **w1**=*Workspace* **w2**=*Workspace* w1 & w2

Workspace_append

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_append; PARAMETERS w1=workspace w2=workspace; RESULT wres
LOCAL lenx1 lenx2 leny1 leny2
          lenx1 = length(w1.x)
          lenx2 = length(w2.x)
          leny1 = length(w1.y)
          leny2 = length(w2.y)
          IF (Max(w1.x) = Min(w2.x))
                    wres = w1
                    wres.x = w1.x & w2.x[2:lenx2]
                    wres.y = w1.y & w2.y
                    wres.e = w1.e & w2.e
          ELSEIF (leny1 = lenx1) OR (leny2 = lenx2)
                    printen "Error - both workspaces must be histograms"
          ELSE
                    printen "Error - Start and end X values must match"
          ENDIF

ENDPROCEDURE
```

Workspace_divide()

Called by the generic function x / y when x and y are of type Workspace.

WORKSPACE_DIVIDE() **w1**=*Workspace* **w2**=*Workspace* w1 / w2

Workspace_divide

The default procedure definition and the calculation of errors for this procedure is shown below. Note that the characteristics of the Left hand workspace are those that define the resultant workspace (i.e. final length, dimensionality of arrays).

```
PROCEDURE workspace_divide; PARAMETERS w1=workspace w2=workspace; RESULT wres

    IF (w1.x = w2.x) AND (w1.xlabel = w2.xlabel)
          wres = w1
          wres.y = w1.y / w2.y
        wres.e = wres.y * sqrt( (w1.e/w1.y)^2 + (w2.e/w2.y)^2 )
        IF w2.ylabel != "(dimensionless)"
            wres.ylabel = "(dimensionless)"
        ENDIF
    ELSE
          printin "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

Workspace_raised_to()

Called by the generic function x ^ y when x and y are of type Workspace.

WORKSPACE_RAISED_TO() **w1**=*Workspace* **w2**=*Workspace* w1 ^ w2

Workspace_raised_to

The default procedure definition and the calculation of errors for this procedure is shown below. Note that the characteristics of the Left hand workspace are those that define the resultant workspace (i.e. final length, dimensionality of arrays).

```
PROCEDURE workspace_raised_to; PARAMETERS w1=workspace w2=workspace; RESULT wres

          IF (w1.x = w2.x) AND (w1.xlabel = w2.xlabel)
                    wres = w1
                    wres.y = w1.y ^ w2.y
                    wres.e = wres.y * sqrt((ln(w1.y)*(w2.e))^2 + ((w2.y/w1.y)*w1.e)^2)
          ELSE
                    printin "Workspaces are not compatible - please re-bin"
          ENDIF

ENDPROCEDURE
```

Workspace_subtract()

Called by the generic function x - y when x and y are of type Workspace.

WORKSPACE_SUBTRACT() **w1**=*Workspace* **w2**=*Workspace* w1 - w2

Workspace_subtract

The default procedure definition and the calculation of errors for this procedure is shown below. Note that the characteristics of the Left hand workspace are those that define the resultant workspace (i.e. final length, dimensionality of arrays).

```
PROCEDURE workspace_subtract; PARAMETERS w1=workspace w2=workspace; RESULT wres

          IF (w1.x = w2.x) AND (w1.xlabel = w2.xlabel)
                    wres = w1
                    wres.y = w1.y - w2.y
                    wres.e = sqrt( w1.e*w1.e + w2.e*w2.e )
          ELSE
                    printin "Workspaces are not compatible - please re-bin"
          ENDIF

ENDPROCEDURE
```

Workspace_modulo()

Called by the generic function x | y when x and y are of type Workspace.

WORKSPACE_MODULO() **w1**=*Workspace* **w2**=*Workspace* w1 | w2

Workspace_modulo

The default procedure definition and the calculation of errors for this procedure is shown below. Note that the characteristics of the Left hand workspace are those that define the resultant workspace (i.e. final length, dimensionality of arrays).

```
PROCEDURE workspace_modulo; PARAMETERS w1=workspace w2=workspace; RESULT wres

          IF (w1.x = w2.x) AND (w1.xlabel = w2.xlabel)
                    wres = w1
                    wres.y = w1.y | w2.y
                    wres.e = wres.y / 0.0  # Set errors to undefined
                    wres.ylabel = "(dimensionless)"
          ELSE
                    printin "Workspaces are not compatible - please re-bin"
          ENDIF

ENDPROCEDURE
```

Workspace_multiply()

Called by the generic function x * y when x and y are of type Workspace.

WORKSPACE_MULTIPLY() **w1**=*Workspace* **w2**=*Workspace* w1 * w2

Workspace_multiply

The default procedure definition and the calculation of errors for this procedure is shown below. Note that the characteristics of the Left hand workspace are those that define the resultant workspace (i.e. final length, dimensionality of arrays).

```
PROCEDURE workspace_multiply; PARAMETERS w1=workspace w2=workspace; RESULT wres

    IF (w1.x = w2.x) AND (w1.xlabel = w2.xlabel)
          wres = w1
          wres.y = w1.y * w2.y
        wres.e = sqrt((w1.y*w2.e)^2 + (w2.y*w1.e)^2)
    ELSE
          printin "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

Workspace_and()

Called by the generic function x AND y when x and y are of type Workspace.

WORKSPACE_AND() **w1**=*Workspace* **w2**=*Workspace* w1 AND w2

Workspace_and

The default procedure definition and the calculation of errors for this procedure is shown below. This function does not have a useful implementation currently. This is really only included for the sake of completeness as the syntax does allow it to be called.

```
PROCEDURE workspace_and; PARAMETERS w1=workspace w2=workspace; RESULT wres
          printin   "Operation AND is not defined for workspaces"
ENDPROCEDURE
```

Workspace_equal()

Called by the generic function x = y when x and y are of type Workspace.

WORKSPACE_EQUAL()	**w1**=*Workspace* **w2**=*Workspace*	w1 = w2

Workspace_equal

The default procedure definition and the calculation of errors for this procedure is shown below. Note that this does not test exact equality of all the workspace, just the main operational fields and units.

```
PROCEDURE workspace_equal; PARAMETERS w1=workspace w2=workspace; RESULT wres
          wres = (w1.x = w2.x) AND (w1.xlabel = w2.xlabel) AND ( w1.y = w2.y )
ENDPROCEDURE
```

Workspace_greater_than()

Called by the generic function x > y when x and y are of type Workspace.

WORKSPACE_GREATER_THAN() **w1**=*Workspace* **w2**=*Workspace* w1 > w2

Workspace_greater_than

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_greater_than; PARAMETERS w1=workspace w2=workspace; RESULT wres
          printin   "w1 > w2 is not currently defined for workspaces"
ENDPROCEDURE
```

Workspace_greater_than_or_equal()

Called by the generic function x >= y when x and y are of type Workspace.

WORKSPACE_GREATER_THAN_OR_EQUAL()	**w1**=*Workspace* **w2**=*Workspace*	w1 >= w2

Workspace_greater_than_or_equal

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_greater_than_or_equal; PARAMETERS w1=workspace w2=workspace; RESULT wres
          printin   "w1 >= w2 is not currently defined for workspaces"
ENDPROCEDURE
```

Workspace_less_than()

Called by the generic function x < y when x and y are of type Workspace.

WORKSPACE_LESS_THAN() **w1**=*Workspace* **w2**=*Workspace* w1 < w2

Workspace_less_than

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_less_than_or_equal; PARAMETERS w1=workspace w2=workspace
RESULT wres
          printin   "w1 <= w2 is not currently defined for workspaces"
ENDPROCEDURE
```

Workspace_less_than_or_equal()

Called by the generic function x <= y when x and y are of type Workspace.

WORKSPACE_LESS_THAN_OR_EQUAL()	**w1**=*Workspace* **w2**=*Workspace*	w1 <= w2

Workspace_less_than_or_equal

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_less_than_or_equal; PARAMETERS w1=workspace w2=workspace
RESULT wres
          printin   "w1 <= w2 is not currently defined for workspaces"
ENDPROCEDURE
```

Workspace_not_equal()

Called by the generic function x != y when x and y are of type Workspace.

WORKSPACE_NOT_EQUAL() **w1**=*Workspace* **w2**=*Workspace* w1 != w2

Workspace_not_equal

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_not_equal; PARAMETERS w1=workspace w2=workspace; RESULT wres
          wres = ( w1.x != w2.x ) OR ( w1.y != w2.y )
ENDPROCEDURE
```

Workspace_or()

Called by the generic function x OR y when x and y are of type Workspace.

WORKSPACE_OR()	**w1**=*Workspace* **w2**=*Workspace*	w1 OR w2

Workspace_or

The default procedure definition and the calculation of errors for this procedure is shown below.

```
PROCEDURE workspace_or; PARAMETERS w1=workspace w2=workspace; RESULT wres
          printin   "Operation OR is not defined for workspaces"
ENDPROCEDURE
```

Chapter 1

New Data Formats

There is quite a lot of support already built into Open GENIE pending ratification and a formal definition of the Neutron and X-ray scattering data format NeXus. There are some routines already in Open GENIE to support this style of data representation however until the format is ratified, this section only serves to mark some likely placeholder procedures from prototype versions.

As_new	Converts a GENIE-V2 compatible workspace/data file to a new format workspace
As_old	Converts a new format workspace/data file to a GENIE-V2 compatible workspace
Data	Constructor for Data information
Experiment	Constructor for Experiment information
History	Constructor for History information
Instrument	Constructor for Instrument information
Sample	Constructor for Sample information
User	Constructor for User information
Unit	Constructor for Unit information

Chapter 2

Appendices

Supported Data File Formats

Open GENIE supports access to the file formats listed below:

- ISIS Raw File (Read only)
- GENIE-II Intermediate file (Read only)
- Open GENIE Intermediate File (Read and Write, Machine independent)
- Open GENIE ASCII File (Read and Write)
- HDF Files (Write only, see following note)

These files can all be read directly with the *Get()* and *Put()* commands by specifying numbered or named data elements. Open GENIE supports access to any format of ASCII file via the *Asciifile()* command.

Supported Graphics Devices

This is a list of Graphics device currently supported by Open GENIE. Currently Open GENIE relies on the PGPLOT package and it may well be possible to use other pgplot drivers with Open GENIE. In the distributed version of Open GENIE we have only checked out the functioning of the devices below on all platforms. To see other devices which may be available on a particular version of Open GENIE type

```
>> Device/Open "Help"
```

Display devices

Device Name	Description String	Not supported on
Tcl/Tk driver	"TK" or "tk"	
X-Windows devices	"XWINDOW" or "xwindow"	

Hardcopy devices

Device Name	Description String	Not supported on
Postscript (portrait)	"PS" or "ps"	
Colour Postscript (portrait)	"CPS" or "cps"	
Postscript (landscape)	"PS" or "ps"	
Colour Postscript (landscape)	"VCPS" or "vcps"	

Supported Graphics Attributes

Fonts

Open GENIE uses the PGPLOT fonts when drawing fonts onto the open graphics device, currently, the supported fonts available are:

$NORMAL
$ROMAN
$ITALIC
$SCRIPT

Note: For accessing special scientific/greek characters within these fonts, the documented PGPLOT escape sequences may be used in the Open GENIE text string being plotted. For example "Time-of-Flight (\gms)" where "\gm" produces a greek Mu character. For further details see the PGPLOT documentation, available at *ftp://astro.caltech.edu/pub/pgplot*.

Markers

Markers types available for plotting.

$POINT
$PLUS
$STAR
$CIRCLE
$CROSS
$BOX

Linestyles

Linestyles available for plotting.

$FULL
$DASH
$DOT_DASH
$DOT

Colours

Pre-defined colours available for the graphics by default. For using further colours in the graphics, see the *Colourtable()* and *Colour()* comamnds.

$BLACK
$WHITE
$RED
$GREEN
$BLUE
$CYAN
$MAGENTA
$YELLOW
$ORANGE
$LIGHT_GREEN
$SEA_GREEN
$LIGHT_BLUE
$PURPLE
$CRIMSON
$DARK_GRAY
$LIGHT_GRAY

Implicit Data conversions table

This table shows how implicit data conversions are handled in Open GENIE. These occur when any two items of *different* data types are used together in one expression, for example mutliplying an array of Real values by a single integer. The table is symmetrical so only the possibilities on and below the diagonal are shown.

For the example above, we look in the row labeled RA and column labeled I, from this we can see that the result will be an array of Real values.

+ - * /	**I**	**R**	**IA**	**RA**	**W**	**WA**	**U**
I	I						
R	R	R					
IA	IA	RA	IA				
RA	RA	RA	RA	RA			
W	W	W	W	W	W		
WA	WA	WA	WA	WA	WA	WA	
U	U	U	U	U	U	U	U

Key

I = Integer
R = Real
W = Workspace
IA = Integer Array
RA = Real Array
WA = Workspace Array
U = Undefined value (nil)

Regular Expressions

This description was copied from the TCL documentation which in turn was copied from a manual page written by Henry Spencer. The definition below is very succinct but accurate and worth a study, alternatively you may find a more readable description on UNIX systems by doing "man ed" and looking up regular expressions!

A regular expression is zero or more **branches**, separated by "|". It matches anything that matches one of the branches.

A branch is zero or more **pieces**, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an **atom** possibly followed by "*", "+", or "?". An atom followed by "*" matches a sequence of 0 or more matches of the atom. An atom followed by "+" matches a sequence of 1 or more matches of the atom. An atom followed by "?" matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a **range** (see below), "." (matching any single character), "^" (matching the null string at the beginning of the input string), "$" (matching the null string at the end of the input string), a "\" followed by a single character (matching that character), or a single character with no other significance (matching that character).

A **range** is a sequence of characters enclosed in "[]". It normally matches any single character from the sequence. If the sequence begins with "^", it matches any single character **not** from the rest of the sequence. If two characters in the sequence are separated by "-", this is shorthand for the full list of ASCII characters between them (e.g. "[0-9]" matches any decimal digit). To include a literal "]" in the sequence, make it the first character (following a possible "^"). To include a literal "-", make it the first or last character.

Here are some examples to get you going.

(abc|def)	Matches the string "abc" or the string "def"
^U +	Matches the beginning of the line followed directly by a U then one or more spaces
F.REWO	Matches the strings FOREWORKS FIREWORKS F@REWORKS etc.

RKS

A[^BC]D Matches anything A.D matches but excluding ABC and ACD.

^[\t]*$ Matches any lines with only spaces or tabs on them.

List of Functions

This index lists all the supported functions in Open GENIE. In some cases commands fall naturally under more than one category, where this occurs the command will be found documented under the most appropriate section. References in parentheses show where a command could also be listed appropriately.

- ***Data Analysis Functions***

  focus peakgen rebin sumspec units integrate peakfit

- ***Graphics Commands***

  alias alter axes cell cell_array cell_function cell_wedge centre_bins colour colourtable contour contour_array contour_function contour_label cursor debug delete dev device display draw errors get getcursor graph graticule hardcopy histogram labels limits line markers multi_plot multiplot new_zoom obj peak pic pic_add picture plot polygon redim redraw select text title toggle undraw win win_autoscaled win_multiplot win_scaled win_twod win_unscaled window zoom select

- ***GENIE-V2 Emulation***

  assign cfn groupbins jump keep s setpar show set scatmode

- ***I/O Commands***

  asciifile filetype Get() list module put (*set*) (*show*) *nblocks* (*inquire*) *print printd printdn printe printen printi printin printn* (*read_terminal*)

- ***Array and Workspace Handling Functions***

  bracket Centre_bins() cut dimensionality dimensions fields fill fix (*length*) *max min Redim() sum unfix*

- ***String Handling functions***

  as_string as_variable length Locate() Substring()

- ***Mathematical functions***

  abs arccos arcsin arctan cos exp ln log sin sqrt tan

- ***General Programming functions***

  Alias() as_integer (*as_string*) (*as_variable*) *called_as_function defined free inquire interactive is_a now read_terminal*

- ***System Dependent Functions***

  cd dir pwd os system

- ***Interactive Control of Open GENIE sessions***

  exit copying help load save warranty

- ***Diagnostics and Debugging***

  Debug() gripe inspect version

- ***Workspace Operations***

  workspace_add, *workspace_and*, *workspace_append*, *workspace_arccos*, *workspace_arcsin*, *workspace_arctan*, *workspace_coerce*, *workspace_cos*, *workspace_divide*, *workspace_equal*, *workspace_exp*, *workspace_greater_than*, *workspace_greater_than_or_equal*, *workspace_less_than*, *workspace_less_than_or_equal*, *workspace_ln*, *workspace_log*, *workspace_modulo*, *workspace_multiply*, *workspace_negated*, *workspace_not*, *workspace_not_equal*, *workspace_or*, *workspace_raised_to*, *workspace_sin*, *workspace_sqrt*, *workspace_subtract*, *workspace_tan*

- ***New data formats***

  as_new as_old data experiment history instrument sample user unit

List of Keywords

This section lists all the predefined keywords in Open GENIE (always in capitals). Some of these work on their own, others are part of larger control structures.

Boolean Expressions

OR
AND
NOT

Procedures

FORWARD
PROCEDURE
ENDPROCEDURE
PARAMETERS
QUALIFIERS
RESULT
GLOBAL
LOCAL
RETURN

Loops

LOOP
FROM
TO
STEP
ENDLOOP
EXITIF

Selection

CASE
IS

ENDCASE
OTHERWISE

IF
ELSEIF
ELSE
ENDIF