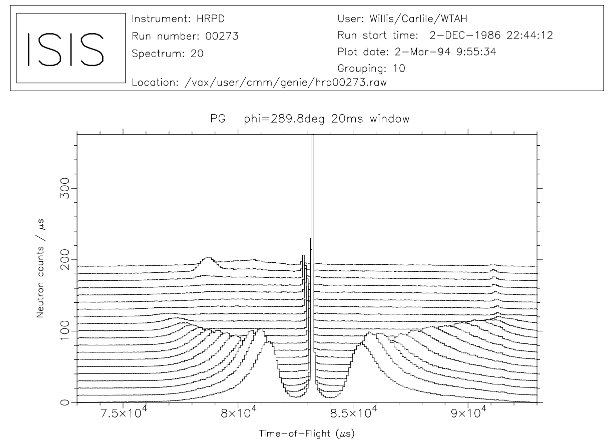
Open GENIE V1.0

USER GUIDE and REFERENCE



Chris Moreton-Smith, Steve Johnston, Freddie Akeroyd
ISIS Facility
DRAL
Chilton, Didcot, OXON
OX11 OQX

January 30, 1995

Chapter 1

Introduction

1.1 Open GENIE

GENIE is a program used for the analysis and display of data from the ISIS neutron scattering instruments. For several years, a stable VAX/VMS only version of the software (GENIE–V2)[1] has been in use at the ISIS facility and at several scientific and commercial establishments worldwide.

Constant improvements in the capabilities of the neutron scattering instruments used at the ISIS and also in the hardware and software available for data analysis have necessitated a re-write of GENIE. The new GENIE is known as Open GENIE to reflect the intention that the software be used on a wide variety of different computers and operating systems.

1.2 This manual

This manual describes Open GENIE. It is intended to allow a new user to become familiar with genie quickly as well as providing detailed technical backup for the experienced user.

[1]See Rutherford Appleton Laboratory report RAL-86-102

There are two parts to this manual, one to cater for those who are learning to use GENIE for the first time and one for those who already know how to use GENIE but need to look somthing up.

- Part I provides an easy to read introduction to the philosphy and use of Open GENIE. A quick reading of Part I whilst trying a few examples should be sufficient to "Get you going."
- Part II serves as a reference manual for all aspects of the GENIE. It is designed as a detailed reference for individual procedures and control constructs.

Part I

Open GENIE User Guide

Chapter 2

The Three Wishes

Most people using GENIE wish to do three things.

1. *Access* their data, either from a file or directly from the electronics.
2. *Manipulate* their data, for example, to focus or normalize it.
3. *Display* their data on the screen and make a hardcopy on a printer

Here is an example of a simple GENIE session to access some time of flight data collected on the IRIS instrument. A constant background is subtracted and the result displayed.

```
$ opengenie
Loading file /library/genie_test/stinit.gcl ..
12 lines scanned/compiled buffer space remaining = 13127
Loading file /library/genie_test/genieinit.gcl
***************************************************************
*                  WELCOME TO Open GENIE 1.0                  *
*                                                             *
***************************************************************
3 lines scanned/compiled buffer space remaining = 13073
>>
```

The command "opengenie" (to start GENIE) is the same on both VMS and Unix based systems though on Unix systems, it must be in lowercase. The prompt "`>>`" means that GENIE is ready to accept commands.

Now that GENIE is running, most people will set up some defaults to allow easy access to files in the data area of the instrument they are using. If you always use one instrument, these defaults can be set up in a command file to be run automatically when GENIE starts (see section 2 Loading Command Files).

`>> set/disk "iris$disk0:"`
Default disk: iris$disk0:
`>> set/instrument "irs"`
Default instrument: irs
`>> set/directory "[irsmgr.data]"`
Default directory: [irsmgr.data]
`>> set/extension "raw"`
Default filename extension: .raw

From this point it is possible to select any raw dataset using the Assign command followed by the run number.

`>> assign 8202`
Default input file: iris$disk0:[irsmgr.data]irs08202.raw

On Unix, the commands for setting defaults are similar and use the appropriate syntax for Unix file names.

`>> set/instrument "irs"`
Default instrument: irs
`>> set/directory "/vax/iris0/irsmgr/data/"`
Default directory: [irsmgr.data]
`>> set/extension "raw"`
Default filename extension: .raw

(no Set/Disk command is required). Note that on both operating systems it is *essential* to put string arguments like filenames in double quotes.

Now that the source for the data has been selected it is possible to access individual spectra from the dataset for manipulation and display. This is done using the Spectrum function. This is normally abbreviated to "s."

```
>> w = s(1)
```
Reading spectrum 1 from iris$disk0:[irsmgr.data]irs08202.RAW
```
>> corrected = w - 40.0
```
¿¿

Here the spectrum is read into a workspace called `w`. A constant background of 40.0 is subtracted from the spectrum in `w` and the result is put into another workspace called `corrected`. Workspaces are created whenever a spectrum is assigned to them and their names are arbitrary.

The corrected spectrum can now be displayed using the Display command. The resulting plot is shown in Figure 2.1.

```
>> display corrected
```
Opening graphics device XW Displayed using bin-grouping of 10

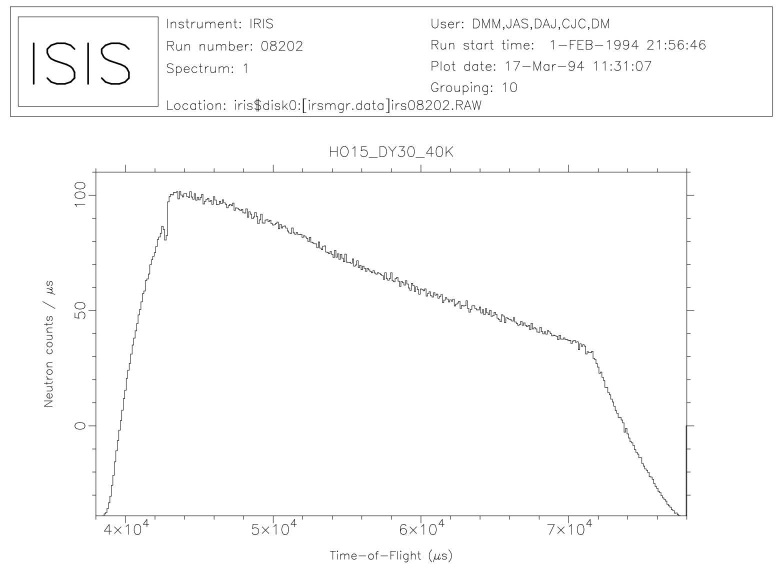


Figure 2.1: Plot of a time of flight spectrum from the IRIS instrument

This example should have given you a good idea what an Open GENIE session looks like and a mental framework into which the more detailed information in the next three chapters will fit.

Chapter 3

The First wish - Access to data

The basic unit of neutron data at the ISIS The basic unit of neutron data at the ISIS is the spectrum.

These may be big or small and there may be just a few, or many thousands of them. As a result, the basic element of data which GENIE deals with is a spectrum. Spectra can be accessed directly from the data acquisition electronics during a data collection run, or, after a run has been finished, from a raw data file.

Chapter 4

The second wish - To manipulate data

Chapter 5

The third wish - To display data

Part II

Open GENIE Reference Manual

Chapter 6

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.

6.1 Single-line commands

The term "single-line" refers to a single logical line, normally terminated by a newline character and containing one keyword command or assignment. It is possible to spread one logical line over several physical lines by appending an "`&`"character and breaking the line immediately after it. 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.

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.

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.

6.1.1 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 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-bgnd 0 xmax/100.0 ymin=MinY(w1) ymax=MaxY(w1)
>>
```

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/Line/Error 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!

6.1.2 Assignments

Apart from keyword commands there is one other sort of single-line command, an 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. eg `5.6` or `"Hello"`.
- A variable, or part of a variable. eg `TwoTheta` or `w1.npts`.
- A bracketed expression eg `( x - 7 )`.
- A function invocation eg `Sin(Pi)`. See Function commands (section 6.1.3).
- A unary expression eg `-xvalue` or `NOT logAxes`. The unary operators are "`- + NOT`" and are evaluated before any binary operators.
- A binary expression eg `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 Variable types (section 7.3).

6.1.3 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

```
>> integ = 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 "`integ`". 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, Xhigh=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 eg "`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 = Read:G2( "My_raw_file.raw" )
>>
```

Here the equivalent keyword command would be Read/G2 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.

6.2 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. All Open GENIE procedures should avoid names which clash with these keywords.

6.2.1 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 <Condition2> 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

```
IF NOT Defined(A4)
    A4 = Dimensions(1000)   # Create a 1000 element long array
    Zero(A4)                # initializes each element to 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
```

6.2.2 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 Mode()                # Check for batch mode operation
    IS "BATCH"
        Device/Open "PS"   # Save postscript plots if in batch
    IS "INTERACTIVE"
        Device/Open "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
```

6.2.3 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

```
LOOP i FROM 1 TO 10 STEP 1
```

```
    Print i^2 " "                # print squares
    IF i = 10; Printn; ENDIF     # print a newline
ENDLOOP


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


position = 0
sr = Dimensions(10)      # Dimension an array of strings
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
```

6.2.4 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 customize Open GENIE for a specific task or for a specific group of data manipulation and display operations.

```
    PROCEDURE <name>
  [ QUALIFIERS /<Qname1> /<Qname2> ... /<QnameN> ]
  [ PARAMETERS <parname1>=<partype1> ... ]
  [ RESULT <resname>=<default-value> ]
```

```
[ GLOBAL <gloabalvar1> [ = <initial-value> ] ... ]
[ LOCAL <localvar1> [ = <initial-value> ] ... ]

  <Statements>

  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, section 6.1).

`<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. 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 ie true if the qualifier was specified on the command line, false otherwise. Qualifier variables can be tested with IF statements.

  Examples

```
QUALIFIERS /Line /Markers
...
IF NOT Markers
    printn "Without markers"
ENDIF
```

- `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

```
PROCEDURE Fix_results
PARAMETERS bad_point=Integer yarray=RealArray
#   zero the bad point
    yarray[bad_point] = 0.0
ENDPROCEDURE

>> # The call to this procedure could be
>> Fix_results  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

```
PROCEDURE SinD         # sind(x) for angle 0 <= x <= 360
PARAMETERS x=real
RESULT res = 999.0     # Sentinel silly value
GLOBAL $PI             # Global (defined elsewhere)

IF ( 0.0 <= x ) AND ( x <= 360.0 )
    res = Sin( x * (2.0*$PI/360.0) )
ENDIF

ENDPROCEDURE
```

- `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

```
PROCEDURE TestG
GLOBAL something = 45.0  anything = 50.0
LOCAL anything = 90.0

Printn something " " anything

ENDPROCEDURE

>> # Session 1
>> Testg
45.000000 90.000000
>>

>> # New session 2
>> something = 88.0    # Override the GLOBAL
>> anything = 22.0     # a new global variable
>> Testg
88.000000 90.000000
>>
```

Chapter 7

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.

7.1 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 startup. It is possible to see what variables are defined and the values they hold by using the Show/Variables command. The Show/Procedures command will list currently defined procedures.

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 on the opengenie command line.

7.2 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.

7.2.1 Naming system procedures and variables

All procedure and variable names beginning with an underscore are reserved for use by Open GENIE. The exception to the rule is "_" itself which is defined to be undefined and can be used freely. By adding the /System qualifier to the Show/Procedures and Show/Variables 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.

7.2.2 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/Constants command.

7.2.3 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.

The recommended approach is use lower case for everything except control constructs (see section 6.2 on Multi-line commands). For the pedantic, all references to procedures and their qualifiers should be capitalized (ie 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 capitalized (eg MyProcedure). The examples in this manual are kept as close to these conventions as possible.

7.3 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.

7.3.1 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$x$,$\arctan x$
- Log(x), Ln(x), Exp(x), Alog(x)
  Transcendental functions. $\mathrm{Log}_{10} x$, $\log_e x$, e^x, Anti-$\log_{10} x$.

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. Alternatively, the more general conversion function As_Real can be used. This will attempt to obtain a real number from any variable type supplied to it.

7.3.2 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.

7.3.3 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.

- `+`
  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(), Index(), Length()
  The location of sub-strings within a string and the extraction of parts of the string can be carried out using these functions.

Examples

```
>> nth = 2      # find the 2nd occurrence
>> printn Index("you fish you",  "you", nth)
10              # start position of 2nd "you"
>> pos = 6; length = 4
>> printn Substring("Blackbird", pos, length)
bird
```

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.
- `\"`
  Put a “"” 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

```
# 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


# String concatenation to number
>> a = as_string(3.0)
>> a = "*twiddly bits -> " + a + " <- twiddly bits*"
>> printn a
*twiddly bits -> 3.0 <- twiddly bits*
```

7.3.4 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

Real array
===========
dims[0..n]= [ 100  200 ]
dope[0..n]= [ 200  1 ]
  5.000000  (undefined)  (undefined)  ...(20000 elements)
```

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 undefined. 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[l,m,n...]` (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)
  Not yet available but watch this space !!.

- `+ -` (Unary)
  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 equal length – All pairs of elements are processed and have a result stored in the result array.
  2. The LHS array is longer – The extra elements are set to undefined in the result array.
  3. The LHS array is shorter – The result array is truncated to the length of the LHS array.

- `<  > <=  >=  = !=`
  Comparisons for arrays: 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()
  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. The same result could have been achieved explicitly by calling the function As_Realarray() eg

```
>> y_values = y_values * As_Realarray(2, Length(y_values))
>>
```

This function along with the corresponding As_Integerarray(), As_Stringarray() and As_Workspacearray() will try to convert any variable to an array of the type requested. The implicit conversions allowed in Open GENIE are shown in Table 7.1.

7.3.5 Workspaces

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 shows the of the construction of a simple workspace interactively.

Examples

```
#  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)    = (3.141590)
        (anarray)       = (
            Integer array
            =============
            dims[0..n]= [ 2  3 ]
            dope[0..n]= [ 3  1 ]
                25 (undefined) (undefined) ...(6 elements) )
        (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 operate in a manner similar to the way the original GENIE-V2 program operated on its workspaces. The details of these template operations are given in Chapter 8, 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)
  Workspace_negated() (Note that unary "`+`" is a no-op).

- `+ - * / | ^` (Binary)
  Workspace_add()
  Workspace_subtract()
  Workspace_multiply()
  Workspace_divide()
  Workspace_modulo()
  Workspace_raised_to()

- `< > <= >= = !=`
  Workspace_less_than()
  Workspace_greater_than()
  Workspace_less_than_or_equal()
  Workspace_greater_than_or_equal()
  Workspace_equal()
  Workspace_not_equal()

- `&`
  Workspace_append()

- All the trigonometric and transcendental functions are defined generically and call workspace specific functions.

$sin \rightarrow$ Workspace_sin()
$cos \rightarrow$ Workspace_cos()
$tan \rightarrow$ Workspace_tan()
$arcsin \rightarrow$ Workspace_arcsin()
$arccos \rightarrow$ Workspace_arccos()
$arctan \rightarrow$ Workspace_arctan()
$ln \rightarrow$ Workspace_ln()
$exp \rightarrow$ Workspace_exp()
$log \rightarrow$ Workspace_log()
$sqrt \rightarrow$ Workspace_sqrt()

- Length()
  Returns the number of fields in the workspace.

Automatic conversions (see table 7.1, Implicit conversions) apply to operations combining workspaces with other Open GENIE numeric types. The As_workspace() function will make a workspace from any type where possible, however, it is important to note that this will follow the rules specified in the template routine Workspace_coerce() which is user re-definable.

The default action on a workspace (ie. defined by the template routines) is to perform the action 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. The next chapter should make the operations on workspaces clearer.

I	=	Integer	R	=	Real
W	=	Workspace	IA	=	IntegerArray
RA	=	RealArray	WA	=	WorkspaceArray
U	=	Undefined			

`+,-,*,/, ...`	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

Table 7.1: Implicit conversions for numerically based types

Chapter 8

Workspace Operations

Workspace operations and transformations form the heart of Open GENIE. They permit analysis which takes into the account the underlying model of 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 orignal GENIE-V2 workspace. The arithmetic functions (as defined in the Template routines, section8.2) require arrays of x, y, and error values to be present (a basic histogram). For the Units command to work the workspace also requires fields giving parameters such as the primary flight path and incident angle (a TOF spectrum).

Soon it will also be possible to load and display data in the form of two-dimensionsal workspaces. Where relevant, the internal functions will be generalised to cope with this extension. If a function cannot operate on the data supplied, either because it is not in a suitable format, or because required fields are missing, an error message will be produced.

8.1 Classification

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, this consists of one histogram of data and may also contain annotation information suitable for a plot.

In documenting the operations on workspaces, the type of the parameter is given as "TOFSpec" where a time of flight spectrum is required or "Histogram" where a histogram without any other information is sufficient. The one thing to remember is that both these are workspace variables but with certain required fields present.

8.1.1 Required fields

This section lists the required fields and types needed in a workspace to "Qualify" as a Histogram, TOFSpectrum or AnnotatedSpectrum.

When two-dimensional workspaces are added it is likely that a two-dimensional histogram will be defined with a different form of TOF spectrum to make use of it. Functions like Rebin will be extended to operate on it and new functions will be defined relevant to the underlying data being analysed.

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

2DHistogram

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

TOFSpectrum

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	Real
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

2DTOFSpectrum

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	Real
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

AnnotatedSpectrum

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

Annotated2DSpectrum

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

8.2 Template routiones

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 customization 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.

8.2.1 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. eg

```
>> Load "SRC/workspace_user.gcl"
Loading file SRC/workspace_user.gcl
    275 lines scanned/compiled  buffer space remaining = 10958
>>
```

8.3 The Workspace_add() Command

The Workspace_add() command is used to sum the Y-values in the two workspaces taking errors into account. It will be called automatically whenever the "+" operator is used in to operate between two workspaces.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
P2)	W1	(Histogram)	Second operand
RES)		(Histogram)	Sum of w1 and w2

Examples

```
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
        printn "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

8.4 The Workspace_subtract() Command

The Workspace_subtract() command is used to subtract the Y-values in the two workspaces taking errors into account. It will be called automatically whenever the "-" operator is used in to operate between two workspaces.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
P2)	W1	(Histogram)	Second operand
RES)		(Histogram)	Difference of w1 and w2

Examples

```
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
        printn "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

8.5 The Workspace_multiply() Command

The Workspace_multiply() command is used to multiply the Y-values in the two workspaces taking errors into account. It will be called automatically whenever the "*" operator is used in to operate between two workspaces.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
P2)	W1	(Histogram)	Second operand
RES)		(Histogram)	Product of w1 and w2

Examples

```
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 = w1.y*w2.e + w2.y*w1.e
    ELSE
        printn "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

8.6 The Workspace_divide() Command

The Workspace_divide() command is used to divide the Y-values in the first workspace by the Y-values in the second workspace taking errors into account. It will be called automatically whenever the "/" operator is used in to operate between two workspaces.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
P2)	W1	(Histogram)	Second operand
RES)		(Histogram)	division of w1 by w2

Examples

```
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 = (w1.e*w2.y + w1.y*w2.e) / (w2.y*w2.y)
        wres.ylabel = "(dimensionless)"
    ELSE
        printn "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

8.7 The Workspace_modulo() Command

The Workspace_modulo() command is used to give the remainders of the Y-values in the first workspace after division by the Y-values in the second workspace (errors are set as for division). It will be called automatically whenever the "|" operator is used in to operate between two workspaces.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
P2)	W1	(Histogram)	Second operand
RES)		(Histogram)	Remainder of w1/w2

Examples

```
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 = (w1.e*w2.y + w1.y*w2.e) / (w2.y*w2.y)
        wres.ylabel = "(dimensionless)"
    ELSE
        printn "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

8.8 The Workspace_raised_to() Command

The Workspace_raised_to() command is used to raise the Y-values in the first workspace to the powers given in Y-values in the second workspace taking account of errors. It will be called automatically whenever the "^" operator is used to operate between two workspaces.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
P2)	W1	(Histogram)	Second operand
RES)		(Histogram)	w1 to the power of w2

Examples

```
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 = n * wres.y * w1.e / w1.y
    ELSE
        printn "Workspaces are not compatible - please re-bin"
    ENDIF

ENDPROCEDURE
```

8.9 The Workspace_negated() Command

The Workspace_negated() command is used to negate the Y-values, the errors are not affected. It will be called automatically whenever the unary "-" operator is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	-w1

Examples

```
PROCEDURE Workspace_negated
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = -w1.y

ENDPROCEDURE
```

8.10 The Workspace_sin() Command

The Workspace_sin() command is used to take the sines of the Y-values in radians, the errors are set to undefined. This function will be called automatically whenever the generic Sin() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Sines of Y-values

Examples

```
PROCEDURE Workspace_sin
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = sin(w1.y)
    wres.e = wres.e/0.0  # makes undefined

ENDPROCEDURE
```

8.11 The Workspace_cos() Command

The Workspace_cos() command is used to take the cosines of the Y-values in radians, the errors are set to undefined. This function will be called automatically whenever the generic Cos() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Cosines of Y-values

Examples

```
PROCEDURE Workspace_cos
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = cos(w1.y)
    wres.e = wres.e/0.0  # makes undefined

ENDPROCEDURE
```

8.12 The Workspace_tan() Command

The Workspace_tan() command is used to take the tangents of the Y-values in radians, the errors are set to undefined. This function will be called automatically whenever the generic Tan() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Tangents of the Y-values

Examples

```
PROCEDURE Workspace_tan
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = tan(w1.y)
    wres.e = wres.e/0.0  # makes undefined

ENDPROCEDURE
```

8.13 The Workspace_arcsin() Command

The Workspace_arcsin() command is used to return the arcsines of the Y-values in radians, the errors are set to undefined. This function will be called automatically whenever the generic Arcsin() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Arcsines of Y-values

Examples

```
PROCEDURE Workspace_arcsin
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = arcsin(w1.y)
    wres.e = wres.e/0.0  # makes undefined

ENDPROCEDURE
```

8.14 The Workspace_arccos() Command

The Workspace_arccos() command is used to return the arccosines of the Y-values in radians, the errors are set to undefined. This function will be called automatically whenever the generic Arccos() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Arccosines of Y-values

Examples

```
PROCEDURE Workspace_arccos
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = arccos(w1.y)
    wres.e = wres.e/0.0  # makes undefined

ENDPROCEDURE
```

8.15 The Workspace_arctan() Command

The Workspace_arctan() command is used to return the arctangents of the Y-values in radians, the errors are set to undefined. This function will be called automatically whenever the generic Arctan() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Arctangents of the Y-values

Examples

```
PROCEDURE Workspace_arctan
    PARAMETERS w1=workspace
    RESULT wres

    wres=w1
    wres.y = arctan(w1.y)
    wres.e = wres.e/0.0  # makes undefined

ENDPROCEDURE
```

8.16 The Workspace_ln() Command

The Workspace_ln() command is used to take the natural logarithms of the Y-values. This function will be called automatically whenever the generic Ln() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Natural logarithms of the Y-values

Examples

```
PROCEDURE Workspace_ln
    PARAMETERS w1=workspace
    RESULT wres

    wres = w1
    wres.y = ln(w1.y)
    wres.e = w1.e/w1.y

ENDPROCEDURE
```

8.17 The Workspace_exp() Command

The Workspace_exp() command is used to exponentiate the Y-values. This function will be called automatically whenever the generic Exp() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Exponentiated Y-values

Examples

```
PROCEDURE Workspace_exp
    PARAMETERS w1=workspace
    RESULT wres

    wres = w1
    wres.y = exp(w1.y)
    wres.e = w1.e * w1.y

ENDPROCEDURE
```

8.18 The Workspace_log() Command

The Workspace_log() command is used to take $\log_{10}$ of the Y-values. This function will be called automatically whenever the generic Log() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Log_{10} of the Y-values

Examples

```
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
```

8.19 The Workspace_sqrt() Command

The Workspace_sqrt() command is used to take the square root of the Y-values. This function will be called automatically whenever the generic Sqrt() function is used on a workspace.

Parameters

P1)	W1	(Histogram)	First operand (dominant)
RES)		(Histogram)	Sqare root of the Y-values

Examples

```
PROCEDURE Workspace_sqrt
    PARAMETERS w1=workspace
    RESULT wres

    wres = w1
    wres.y = sqrt(w1.y)
    wres.e = w1.e / 2.0

ENDPROCEDURE
```

The following functions are not defined at the moment in Open GENIE but templates do exist to allow a definition to be inserted if desired.

Function name	Argument1	Argument2	Result
Workspace_less_than()	Workspace	Workspace	TruthValue
Workspace_greater_than()	Workspace	Workspace	TruthValue
Workspace_less_than_or_equal()	Workspace	Workspace	TruthValue
Workspace_greater_than_or_equal()	Workspace	Workspace	TruthValue
Workspace_equal()	Workspace	Workspace	TruthValue
Workspace_not_equal()	Workspace	Workspace	TruthValue
Workspace_append()	Workspace	Workspace	Workspace

Table 8.1: Unimplemented workspace functions

8.20 Operations on One-Dimensional Time Of Flight Spectra

The following functions expect the arguments passed to them to be valid neutron scattering TOF Spectra.

The operations provided in this section are listed below.

- Integrate()
  performs an integration of a TOF spectrum.
- Rebin()
  performs an area constant interpolation to chosen X-values.
- Tshift()
  Shifts the X values by a constant amount.
- Units()
  Transforms the units of the workspace.

8.21 The Integrate() Command

The Integrate() command is used to perform an integration on a Time of flight spectrum. As the command only requires basic histogram information, the integration can be carried out on spectra with changed units, it is up to the user to make sense of the final units of an integration and its error value.

The Integrate() command is somewhat unusual amongst Open GENIE functions in that it takes a workspace and returns only a single value and its error. To allow the the function to return both, a special workspace is returned containing just these two elements. If if is not required, the error value can be ingnored.

Parameters

P1)	Spectrum	(TOFSpectrum)	Spectrum to be integrated
RES)		(Workspace)	A workspace containing the Result/Error pair

Integrate/Quiet

Performs the integration as normal but does not print the results to the screen.

Examples

```
# find the integral for spectrum 2
>> integrate s(2, "USER$DISK:[CMM.GENIE]HRP00273.RAW")
INTEGRATE: Integrating between 73000.0 and 93000.0
Sum of bins = 423313.000000 counts
Error = 650.633249 counts
>>
```

/continued

```
# Save the result of an integration in a workspace
>> int_res = Integrate:quiet(w1)
>> printn int_res

    Workspace
    =========
        (integral)      = (407544.000000)
        (error)         = (638.401059)
>>


# Just print the error from an integration
>> printn Integrate:quiet(w1).error
638.401059
>>


# Save the result and discard the error
>> myresult =  Integrate:quiet(w1).integral
>>
```

8.22 The Rebin() Command

The Rebin() command allows the X boundaries of a Histogram to be altered, usually this is done to make the histograms in two different spectra congruent so they may be combined using arithmetic operations on the Y-values.

In the first form of the rebin command, the workspace `W1` defines the positions of the X bin boundaries. The second workspace is re-binned to use these boundaries. This form of the command is used to make the workspaces compatible and ready for arithmetic operations between the two.

The second form of the rebin command allows bin widths to be specified explicitly over two different ranges.

Parameters

P1)	Spectrum	(TOFSpectrum)	Spectrum to be re-binned
P2)	Boundaries	(Realarray)	Array of defining X-values
RES)		(Workspace)	Re-binned spectrum
or			
P1)	Spectrum	(TOFSpectrum)	Spectrum to be re-binned
P2)	X1	(Real)	First binning boundary
P3)	D1	(Real)	Bin width for first region
P4)	X2	(Real)	Second binning boundary
P5)	D2	(Real)	Bin width for second region
...			
Pn)	Xn	(Real)	nth binning boundary
Pn+1)	Dn	(Real)	Bin width for nth region
RES)		(Workspace)	Re-binned spectrum

Rebin/Log

Gives logarithmic bin widths. In this case the bin boundaries are decided by the following algorithm. Step from X_n to X_{n+1} in steps of $\delta X_n/X_n$ where δX_n is chosen such that $D_n = \delta X_n/X_n$

Rebin/Linear

Gives linear bin widths (the default).

Examples

```
# load and rebin spectrum 2
# into two distinct ranges
>> w = s(2)
>> j = rebin(w,600.0,100.0,5000.0,200.0,10000.0)
REBIN: X range now 600.000000 to 10000.000000
REBIN: Maximum Y value now 0.075872
>>
```

```
# Rebin w2 with the x-boundaries in w1
>> rebin w2 w1.x
REBIN: X range now 33.000000 to 1000.000000
REBIN: Maximum Y value now 0.75872
>>
```

```
# Rebin the spectrum in workspace `Nio2´ with
# the bin boundaries in the vanadium file
>> rebin Nio2 s(1,"vanadium.sum").x
REBIN: X range now 200.000000 to 8800.000000
REBIN: Maximum Y value now 35.6
>>
```

```
# load and rebin spectrum 2
# logarithmically (clipping the top)
>> w = s(2)
>> j = rebin:Log(w,600.0,0.001,10000.0)
REBIN: X range now 600.000000 to 8000.000000
REBIN: Maximum Y value now 0.484848
>>
```

8.23 The Tshift() Command

The Tshift() command shifts the X-values in the workspace by a constant amount (positive or negative). Care should be taken when shifting workspaces when non-linear binning has been used. There is no guarantee that any relationship between bin widths and absolute X-values will be preserved. For example, if logarithmic binning started at x=2000.0 a positive shift of 1000.0 would shift the start of logarithmic binning to x=3000.0.

NOTE: As is stands, the Tshift command is obsolete and is only provided for compatibility with GENIE-V2. The command can be simulated easily by adding a constant to the X-values in a workspace. For example `w.x = w.x + 1000.0` will have the same effect as a `tshift w 1000.0 command.`

Parameters

```
P1)   Spectrum  (TOFSpectrum)  Spectrum to be shifted
P2)   Offset    (Real)         Size of X shift
RES)            (Workspace)    Shifted spectrum
```

Examples

```
>> tshift w2 1000.0
```
X range now 73004.000000 to 93004.000000

8.24 The Units() Command

The Units() command is used to perform a units conversion on a spectrum workspace. Units are converted in a rigorous manner making use of conversion parameters already present in the spectrum workspace. If any of these parameters are absent or incorrect they must be put right (usually with the Setpar command) before the Units() command can be used. The spectrum workspace returned by a Units() command will have the data values, axis labels and possibly other parameters altered to be consistent with the new units.

Parameters

P1)	Spectrum	(TOFSpectrum)	Spectrum to be converted
RES)		(Spectrum)	The converted spectrum

Units/Channel

Converts the units of the spectrum to a basic histogram of the raw data showing raw count values against histogram channel number. This is a destructive conversion in that information is lost and a reversing of this command is impossible.

Units/Dspacing

Converts the units of the spectrum to d-spacing in Angstroms.

Units/Energy

Converts the units of the spectrum to energy in meV.

Units/Lambda

Converts the units of the spectrum to wavelength in Angstroms. The whole of the flight path (L1+L2) is used in the conversion.

Units/Q

Converts the units of the spectrum to momentum transfer (Q) in Angstroms^{-1}.

Units/SQ

Converts the units of the spectrum to momentum transfer (Q) in Angstroms^{-1}.

Units/Time

Converts the units of the spectrum to time-of-flight in neutrons/microsecond.

Units/La1

Converts the units of the spectrum to wavelength in Angstroms based on the primary flight path (L1).

Units/Tau

Converts the units of the spectrum to reciprocal time-of-flight in neutron counts/metre.

Units/W

Converts the units of the spectrum to energy transfer (E1-E2) in meV.

Units/Wn

Converts the units of a spectrum to Wavenumber in cm^{-1}.

Examples

```
# convert the units for spectrum 2 to d-spacing
>> dspaced = units:d(s(2))
```
GET: Reading spectrum 2 from user$disk:[cmm.genie]hrp00273.raw
UNITS: Changing units to d-spacing (A) ...
UNITS: X range now 1.472168 to 1.875502
UNITS: Maximum Y value now 34390066.679320
```
>>
```

```
# convert the spectrum in "monitor" to
# Energy transfer and reset parameters which
# were not available in the workspace
>> setpar vanadium ef=5.0 em=2 >> transfer = units:w(vanadium)
```
UNITS: Changing units to energy transfer E1-E2 (meV) ...
UNITS: X range now 1.351029 to 5.127391
UNITS: Maximum Y value now 3807842.837987
```
>>
```

8.25 The Setpar() Command

The Setpar command is provided to allow modification of specific spectrum/instrument parameters within a workspace. The parameters could be altered directly in the workspace but there is some risk of errors caused by typing mistakes which are best avoided by using setpar().

Parameters

P1)	Spectrum	(TOFSpectrum)	Spectrum for which the parameters are to be changed
RES)		(Spectrum)	The spectrum with updated parameters

Examples

```
# change a parameter in workspace "work"

>> setpar work ef=4.8
```
Value of fixed energy EFIXED set to 4.800000 meV

```
# read a spectrum in and set the two-theta angle
fixed = setpar( s(23), tt=19.5)
```
GET: Reading spectrum 23 from user$disk:[cmm.genie]hrp00273.raw
Scattering angle (2*THETA) set to 19.500000 degrees (0.340339 radians)

Chapter 9

Workspace Operations

Chapter 10

I/O Commands

This section details all the commands used in Open GENIE for input and output of data. The commands can be grouped in three different areas, each area based around a few major commands and their associated ancillary functions.

1. Input and output of whole spectra from and to different data file formats. A spectrum consists of The major commands in this section for spectrum input/output are the Spectrum(), Mspectrum(), Write/Spectrum and Write/MSpectrum commands
2. Binary storage and retrieval of processed Open GENIE data. These commands work at a more basic level than those dealing with whole spectra. The major commands in this section are Get() and Put.
3. ASCII formatted input and output. Mainly used for text input and output in GCL programs. This covers various Print command derivatives and the Get/Ascii() function

10.1 Whole spectra input/output

Open GENIE allows complete spectra to be input from a variety of raw data file formats. Currently these are ISIS .RAW and .SAV files, direct

DAE access, and Front End Machine memory (CRPT). Formats for input of data from muon experiments and other neutron establishments (eg ILL and Argonne) are or will shortly be provided. See Supported data file formats (Appendix A) for information on the formats currently supported. Where a data file type is supported Open GENIE automatically determines the file type. If a user wishes to check the type of a data file the Filetype() function may be used.

In all cases, the Open GENIE input functions will return either a single spectrum or a group of spectra sharing common X-values (a 2-D histogram). The two main spectrum input functions are Spectrum() usually returning a single TOF spectrum and MSpectrum() returning a 2-D TOF histogram. As well as these there are routines such as Focus() which are used to perform preliminary analysis on large groups of spectra directly from raw data files.

The spectrum output functions allow processed spectra to be output to an Open GENIE intermediate file or if required (for ISIS data only) into a GENIE-V2 intermediate file. The Write/Spectrum and Write/MSpectrum commands handle single spectra and 2-D spectra respectively.

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 to explicity to set the files to be used. For input, 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 shown with the Show/Defaults command.

10.2 The Spectrum() Command

The Spectrum() function reads a single spectrum from the currently set default data file or from a file specified explicitly. The function returns a workspace containing an annotated spectrum. For ISIS data, the time of flight parameters are also drawn from the raw file to allow transformation commands like the Units command to be used. See Classification (Section 8.1) for details of the contents of the returned workspace.

Parameters

P1)	Number	(Integer)	The spectrum no to read
P2)	FileId	(String or Integer)	Full filename or run number
RES)		(AnnotatedSpectrum)	A workspace containing one spectrum

Examples

```
# Read one spectrum from a specified file
>> w = Spectrum(34, "MARI$DISK0:[MARMGR...")
```
GET: Reading spectrum 34 from MARI$DISK0:[MARMGR ...

```
# Set up some defaults
>> set/disk "MARI$DISK0:"
```
Default disk: MARI$DISK0:
```
>> set/directory "[MARMGR.DATA]"
```
Default directory: [MARMGR.DATA]
```
>> set/instrument "MARI"
```
Default instrument: MARI
```
>> set/extension "RAW"
```
Default filename extension: .RAW

```
# now read two spectra from two different files
# (`Spectrum´ is often abbreviated to `S´) # Full and partial defaults
are shown
```

```
>> assign 2244
```
Default input file: MARI$DISK0:[MARMGR...
```
>> background = s(1)
```
GET: Reading spectrum 1 from MARI$DISK0:[MARMGR ...
```
>> fiddled = s(5,2233) - background
```
GET: Reading spectrum 5 from MARI$DISK0:[MARMGR ...

10.3 The MSpectrum() Command

The MSpectrum() function reads a 2-D spectrum from the currently set default data file or from a file specified explicitly. The function returns a workspace containing an annotated 2-D spectrum. For ISIS data, the time of flight parameters are also drawn from the raw file to allow transformation commands like Units to be used. See Classification (Section 8.1) for details of the contents of the returned workspace.

Parameters

P1)	First	(Integer)	The first spectrum number
P2)	Last	(Integer)	The last spectrum number
P3)	FileId	(String or Integer)	Full filename or run number of the file containing the spectrum, defaults to the file set by Assign.
RES)		(Annotated2DSpectrum)	A workspace containing a 2-D spectrum

MSpectrum/Specify

This is very similar to the basic MSpectrum() function but gives more flexibility in choosing which spectra to read. The integer array `SpectrumNumbers` is scanned from beginning to end for spectrum numbers. These spectra are read in order. If an element of the array is undefined then no spectrum will be read. No check is made for duplicates and non-existent spectrum numbers will generate an error.

Parameters

P1)	SpectrumNumbers	(IntegerArray)	Array containing spectrum numbers to read
P2)	FileId	(String or Integer)	Full filename or run number
RES)		(2DAnnotatedSpectrum)	A 2-D workspace

Examples

10.4 The Set() Command

The Set command is used in conjunction with the Assign command. It is used to set up default values for the disk, directory path, instrument name and file extension.

Set/Disk

This sets the default disk, on UNIX systems this is redundant and should be left null. On VMS this is the disk name.

Parameters

P1)	Disk	(String)	The disk to set
RES)		(None)	

Set/Dir

This sets the default directory path for the data files.

Parameters

P1)	Directory	(String)	The directory path
RES)		(None)	

Set/Instrument

This sets the current instrument name. This is an intelligent command and will set parts of the filename which are dependent on the instrument name. It is also able to alter the format of a file name for non-ISIS file names as long as it is given enough information to go on and the instrument is known to Open GENIE.

Parameters

P1)	Instrument	(String)	The name of the Instrument (or possibly the establishment eg “LANCE”)
RES)		(None)	

Set/Extension

This changes the filename extension. On ISIS files this is usually either “.SAV” or “.RAW”. For instruments where the extension is not significant this should be left blank.

Parameters

P1)	FileExtension	(String)	The expected file extension
RES)		(None)	

Set/File

This overrides all the previous qualifiers and allows a complete filename and pathname to be specified. This is most useful for un-numbered data files when the Assign command cannot be used.

Parameters

P1)	File	(String)	The complete file path specification
RES)		(None)	

Set/File/Input

Sets the default input file explicitly. Set/File/Input is the default.

Set/File/Output

Sets the default output file explicitly.

Examples

(see examples for the Spectrum() command)

10.5 The Show() Command

The Show command is used to look at the current default settings. It responds to the same qualifiers as the Set command to show the current setting. In addition, there is an extra command.

Show/Default

This displays all the default settings for file input.

Parameters

RES) (None)

Examples

```
>> Show/directory

Current default directory = [hrpmgr.samples]
```

10.6 The Assign() Command

The Assign command is used in conjunction with the Set commands. It takes a run number as a parameter and combines this with the default values set for disk, directory, extension and instrument to select the file from which data is retrieved. This command is about as intelligent as the Set/Instrument command.

Parameters

P1)	RunNumber	(Integer)	The run number to select the input data
RES)		(None)	

Examples

(see examples for the Spectrum() command)

10.7 The Write() Command

Write/Spectrum

The Write/Spectrum command is used to output complete spectra to an intermediate file. The spectra are appended to the end of an existing file or a new file is created. The spectra can be read out of the file using the Spectrum() command, the spectrum number is given by the position of the spectrum in the file.

Parameters

P1)	Spectrum	(Workspace)	The spectrum to be output
P2)	Filename	(String)	The name of the intermediate file
RES)		(None)	

Write/G2

Writes data out in GENIE-V2 intermediate file format (only available for ISIS data).

Write/MSpectrum

Works similarly to the standard command but writes out a 2-D workspace instead of a single spectrum. Single and 2-D spectra can be stored in the same file but care must be taken to read them with the correct commands.

Parameters

P1)	2DTOFSpectrum	(2-D workspace to be saved)

10.8 The Focus() Command

The Focus command is ...

Parameters

P1)	FileId	(String or Integer)	Identifier of file to inspect
RES)		(String)	String giving the type of the file

Examples

```
# Focus the contents of a data file
>> first_quadrant = focus("user$disk:[cmm.genie]hrp00273.raw", 1, 30)
```

10.9 Basic binary input/output

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 can be viewed as being 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. A spectrum is just a specialised workspace containing fields with standard names, see Classification (Section8.1).

10.10 The Get() Command

The Get() command operates on a lower level than the spectrum based commands. Its allows direct access to any item of data within any format of supported data file. Access to an item in the file is either by a keyword, or a numerical position, or both. See Supported data file formats (Appendix A) for information on keywords for accessing raw data files.

The Get() command can be used in conjunction with the Set and Assign commands, or with an explicit file name specified. Due to the nature of the parameters, the filename parameter must be specified with a keyword if it is used.

Parameters

P1-P_n)	Specifier	(String or Integer)	Specifier to select the required item
P_{n+1})	File	(String)	File to get data from (Optional)
RES)	Returned Item	(Anyvariable)	The item obtained

Examples

```
# Get the fifth stored item from file GEO.DAT
# (could have been stored with the ``Put´´ command) >> varray =
Get(5, File="GEO.DAT")

# Use Get to find the number of spectra
# in the default file
>> printn "Number of spectra = " Get("NSP1")
```

GET: Reading item NSP1 from /vax/mari0/marmgr/data/mar02244.raw

```
548

# Get the raw counts for the 5th spectrum
>> rawspec = Get("DAT1", 5)
```

10.11 The Put() Command

The Put command operates on a lower level than the spectrum based Write command. It 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 it is created.

Variables being put into a file may be optionally tagged with a character string. This makes direct access using the Get command easier.

Parameters

P1)	Variable	(Anyvariable)	Variable to be written
P2)	File	(String)	File to write to
P3)	Tag	(String)	Identifying tag
P4)	Comment	(String)	Comment field
RES)		(None)	

Examples

```
# Store array varray in file GEO.DAT
# (could have been stored with the "Put" command)
>> Put varray File="GEO.DAT")

# Store a group of correction spectra in one file
Put wbk   "suite1.cor" "S_BACK" "Standard background"
Put wvan  "suite1.cor" "S_VAN"  "Standard Vanadium"
Put wpeak "suite1.cor" "S_PEAK" "Sample peak shape"

# Now use one of the spectra just stored
set/file "/vax/mari0/marmgr/data/mar02244.raw"
bg = get("Standard background", File="suite1.cor")
corrected = s(23) -  bg
```

10.12 The Dir() Command

The Dir command is provided to help find out which fields are present in a data file. The output of the command gives the block numbers, sizes, tags and comments (if any) for each field in the file. These block numbers or tag fields can be used to retrieve the data using the Get() command.

Parameters

P1)	File	(String)	File to write to
RES)		(None)	

Examples

```
# Look at the contents of file "suite1.cor"
>> Dir "suite1.cor"
Block    Size (bytes)  Label   Comment
  1     64558         S_BACK Standard background
  2     64558         S_VAN Standard vanadium
  3     64558         S_PEAK Sample peak shape
```

10.13 The Filetype() Command

The Filetype command is provided to return the type of a file which may be read by Open GENIE.

Parameters

P1)	File	(String)	File to inspect
RES)		(String)	String giving the type of the file

Examples

```
# Look at the contents of file "suite1.cor"
>> printn Filetype("user$disk:[cmm.genie]hrp00273.raw")
raw
```

10.14 Character based input/output

Get/ascii Show/data printxx Show/procs ? Show/version ?

Chapter 11

Graphics Commands

11.1 The Device() Command

Device/Open

The supported interactive devices include "XW" Xwindows "TEK" Tek 4010 compatible (with switching) "VT340" Regis graphics

Parameters

P1)	Device	(String)	Device type
P2)	Width	(Number)	Device width (in inches)
P3)	Height	(Number)	Device height (in inches)
RES)		(None)	

Opens an interactive device.

Device/Hardcopy

The supported non-interactive devices include "PS" Postscript "CP" Colour Postscript "HPGL" HPGL driver for 7475A plotter "LA50" Sixel printer

Parameters

P1)	Device	(String)	Device type
P2)	Filename	(String)	Filename for hardcopy device (sets default)
P3)	Width	(Number)	Device width (in inches)
P4)	Height	(Number)	Device height (in inches)
RES)		(None)	

Opens a hardcopy device

Device/Close

Closes the interactive device or file non-interative device. This command flushes output to the file when using a hardcopy device and must be called before the file can be used.

Device/Clear

Clears the screeen or advances to a new sheet of paper.

Examples

```
example here
```

11.2 The Hardcopy() Command

Parameters

P1)	Filename	(String)	Filename to save hardcopy (optional or sets default)
P2)	Type	(String)	Device type see Device/Hardcopy (optional or sets default)
P3)	Picture	(Integer)	Picture number relative to current picture
RES)		(None)	

Copies the contents of the display to the hardcopy device.

Examples

```
example here
```

11.3 The Undraw() Command

Parameters

RES) (None)

Removes the last graphics primitive to be drawn on the display. Hardcopy after an undraw will NOT include primitives that have been undrawn.

If the last item to be undrawn is a window, the world coordinates revert to those of the previous window.

Undraw/All

Clears the screeen or advances to a new sheet of paper.

Undraw/Undo

Undoes the last undraw command (ie redoes the most recently deleted primitive).

Examples

```
example here
```

11.4 The Redraw() Command

Parameters

P1)	Picture	(Integer)	Picture number relative to current picture
RES)		(None)	

Redraw/Info

Prints information about known pictures including the title of the plot. The picture number given may be used with Redraw to redraw that picture.

Redraw/Reset

Redraws the picture as for Redraw with no qualifiers but resets that picture to be the current picture. At this point drawing can continue on the selected picture (in the world coordinate system of the original picture).

The Redraw/info command will now show that the current picture is not the lastest picture. Care should be taken not to overwrite subsequent pictures by doing a Device/Clear or an Undraw/All.

Examples

```
example here
```

11.5 The Window() Command

The window command permits creation of a separate mapping of world coordinates to device coordinates and groups the primitives drawn within it. All primitives except for text are clipped at the edge of the window they are drawn in.

Window/Unscaled

Parameters

P1)	Xmin	(Number)	Lower X boundary (Device coordinates)
P2)	Xmax	(Number)	Upper X boundary (Device coordinates)
P3)	Ymin	(Number)	Lower Y boundary (Device coordinates)
P4)	Ymax	(Number)	Upper Y boundary (Device coordinates)
P5)	Colour	(Integer)	Colour for background
RES)		(None)	

Creates an unscaled window. In an unscaled window, world coordinates range from 0.0 to 1.0 whatever the size or shape of the window.

Window/Scaled

Parameters

P1)	Xmin	(Number)	Lower X boundary (Device coordinates)
P2)	Xmax	(Number)	Upper X boundary (Device coordinates)
P3)	Ymin	(Number)	Lower Y boundary (Device coordinates)
P4)	Ymax	(Number)	Upper Y boundary (Device coordinates)
P5)	WXmin	(Number)	Minimum X world coordinate
P6)	WXmax	(Number)	Maximum X world coordinate
P7)	WYmin	(Number)	Minimum Y world coordinate
P8)	WYmax	(Number)	Maximum Y world coordinate
P9)	Colour	(Integer)	Colour for background
RES)		(None)	

Creates an Scaled window with the given world coordinates.

Window/Autoscale

Parameters

P1)	Xmin	(Number)	Lower X boundary (Device coordinates)
P2)	Xmax	(Number)	Upper X boundary (Device coordinates)
P3)	Ymin	(Number)	Lower Y boundary (Device coordinates)
P4)	Ymax	(Number)	Upper Y boundary (Device coordinates)
P5)	Xarray	(Realarray)	X data
P6)	Yarray	(Realarray)	Y data
P7)	Earray	(Realarray)	Error in Y data
P8)	Colour	(Integer)	Colour for background
P9)	Minmode	(Integer)	Rounding mode for bottom of plot
P10)	Maxmode	(Integer)	Rounding mode for top of plot
RES)		(None)	

Creates an Scaled window with the world coordinates scaled on the data arrays provided. Note, all three arrays must be provided.

Rounding modes control how rounding is done on the Y data. The meanings are as follows. 0) 1) 2) 3)

Window/Multiplot

Parameters

P1)	Xmin	(Number)	Lower X boundary (Device coordinates)
P2)	Xmax	(Number)	Upper X boundary (Device coordinates)
P3)	Ymin	(Number)	Lower Y boundary (Device coordinates)
P4)	Ymax	(Number)	Upper Y boundary (Device coordinates)
P5)	Colour	(Integer)	Colour for background
P6)	Minmode	(Integer)	Rounding mode for bottom of plot
P7)	Maxmode	(Integer)	Rounding mode for top of plot
RES)		(None)	

Creates a window scaled on a multiplot which has already been created by the Multi_plot command.

Rounding modes are as described for Window/Scaled.

Examples

```
example here
```

11.6 The Multi_plot() Command

The Multi_plot command controls the creation, modification and display of multiplots.

Multi_plot/Create

Parameters

P1)	Xarray	(Realarray)	Common X coordinates
P2)	Numhist	(Integer)	Total number of histograms in the multiplot
P3)	Chmin	(Integer)	Index of first X value to be plotted
P3)	Chmax	(Integer)	Index of last X value to be plotted
P4)	Ygap	(Number)	Gap between each successive histogram in world coordinates
P5)	Colour	(Integer)	Colour for histograms
P6)	Mode	(Integer)	Determines whether to work in point or histogram mode
P7)	Linetype	(Integer)	Type of line to use
P7)	Linewidth	(Integer)	Width of line to use
RES)		(None)	

Creates a new multiplot and allocates space for all the spectra subsequently to be added.

Multi_plot/Spectra

Parameters

P1)	Yarray	(Realarray)	Y data values for one histogram
RES)		(None)	

The total number of spectra added should not exceed the number set in the Multiplot/Create command. Arrays longer than the origional X array will be truncated however shorter arrays are allowable.

Multi_plot/Draw

Draws the multiplot in the current window (which ideally should have been created with the Window/Multiplot command to ensure correct scaling).

Examples

```
example here
```

11.7 The Draw() Command

The Draw command is the Open GENIE command for drawing graphics primitives.

Draw/Axes

Parameters

P1)	Colour	(Integer)	Colour for axes
P2)	Linewidth	(Integer)	Width of line to use
P3)	Textheight	(Real)	Text height in multiples of 1/40 display height
P4)	Font	(Integer)	Font to use for text

Draws axes on an already scaled window.

/Axes/[No]Xlog

Forces X-Axes to be logarithmic

/Axes/[No]Ylog

Forces Y-Axes to be logarithmic

Draw/Text

Parameters

P1)	Xpos	(Real)	X position of lower left corner of text string
P2)	Ypos	(Real)	Y position of lower left corner of text string
P3)	Text	(String)	The text string to be drawn
P4)	Colour	(Integer)	Colour for text
P5)	Textheight	(Real)	Text height in multiples of 1/40 display height
P6)	Angle	(Real)	Text angle in degrees anti-clockwise from +ve X directon
P7)	Font	(Integer)	Font to use for text
P8)	Linewidth	(Integer)	Thickness of text lines

Draws text on an window.

Draw/Histogram

Parameters

P1)	Xarray	(Realarray)	X data values
P2)	Yarray	(Realarray)	Y data values
P3)	Colour	(Integer)	Colour for text
P4)	Mode	(Integer)	Determines whether to work in point or histogram mode
P5)	Linetype	(Integer)	Type of line to use
P6)	Linewidth	(Integer)	Thickness of lines

Draws a histogram on a previously scaled window.

Draw/Plot

Parameters

P1)	Xarray	(Realarray)	X data values
P2)	Yarray	(Realarray)	Y data values
P3)	Colour	(Integer)	Colour for text
P4)	Linetype	(Integer)	Type of line to use
P5)	Linewidth	(Integer)	Thickness of lines

Draws a line plot on a previously scaled window.

Draw/Marker

Parameters

P1)	Xarray	(Realarray)	X data values
P2)	Yarray	(Realarray)	Y data values
P3)	Colour	(Integer)	Colour for text
P4)	Symbol	(Integer)	Selects marker type to use
P5)	Textheight	(Real)	Text height in multiples of 1/40 display height
P6)	Linewidth	(Integer)	Thickness of lines

Draws a marker plot on a previously scaled window.

Draw/Errors

Parameters

P1)	Xarray	(Realarray)	X data values
P2)	Yarray	(Realarray)	Y data values
P3)	Earray	(Realarray)	Error in X or Y data values (see qualifiers)
P4)	Colour	(Integer)	Colour for text
P5)	Linewidth	(Integer)	Thickness of lines

Draws error bars on a plot on a previously scaled window.

/Vertical

Draws vertical error bars for errors in the Y data.

/Horizontal

Draws horizontal error bars for errors in the X data.

Draw/Graticule

Parameters

P1)	Colour	(Integer)	Colour for graticule
P2)	Linewidth	(Integer)	Thickness of lines

Draws a graticule on a previously scaled window which will be consistent with the axes unless expliclity overriden by using the Draw/Xlog or Draw/Ylogqualifier.

Draw/Labels

Parameters

P1)	Xlabel	(String)	Label for the X axis
P2)	Ylabel	(String)	Label for the Y axis
P3)	Colour	(Integer)	Colour for graticule
P4)	Textheight	(Real)	Text height in multiples of 1/40 display height
P5)	Font	(Integer)	Font to use for text

Draws X and Y labels around the axes of scaled window.

Draw/Title

Parameters

P1)	Title	(String)	String to use for plot title
P2)	Colour	(Integer)	Colour for graticule
P3)	Textheight	(Real)	Text height in multiples of 1/40 display height
P4)	Font	(Integer)	Font to use for text

Draws a title above the Axes.

Draw/Box

Parameters

P1)	Xmin	(Number)	Lower X boundary (World coordinates)
P2)	Xmax	(Number)	Upper X boundary (World coordinates)
P3)	Ymin	(Number)	Lower Y boundary (World coordinates)
P4)	Ymax	(Number)	Upper Y boundary (World coordinates)
P5)	Colour	(Integer)	Colour for the lines or area if filled
P6)	Linewidth	(Integer)	Thickness of lines

Draws a box within a window.

/Filled

Makes the box filled

Draw/Line

Parameters

P1)	Xmin	(Number)	Lower X boundary (World coordinates)
P2)	Ymin	(Number)	Lower Y boundary (World coordinates)
P3)	Xmax	(Number)	Upper X boundary (World coordinates)
P4)	Ymax	(Number)	Upper Y boundary (World coordinates)
P5)	Colour	(Integer)	Colour for the lines or area if filled
P6)	Linetype	(Integer)	Type of line to use
P7)	Linewidth	(Integer)	Thickness of lines

Draws a line between the two points.

Draw/Quality

Forces text to be of higher quality than the default, eg for papers, presentations etc.

Examples

```
example here
```

11.8 The Get_cursor() Command

Parameters

P1)	Xpos	(Number)	X Start position for cursor
P2)	Ypos	(Number)	Y Start position for cursor
RES)	Cursor	(Workspace)	Workspace containing cursor hit information

The Get_cursor command returns the X and Y coordintes and mouse button pressed for a cursor hit. The values are returned in a workspace with field names as below. The world coordinates returned are those of the CURRENT window which may not be the window in which the cursor hit occured.

w_x,w_y - World coordinates of cursor d_x,d_y - Device coordinates of cursor
char (String) - Button pressed

Get_cursor/Picture

Parameters

P1)	WindowNo	(Integer)	Number of the window specifying world coordinates
P2)	Xpos	(Number)	X Start position for cursor
P3)	Ypos	(Number)	Y Start position for cursor

Examples

```
example here
```

Chapter 12

Session Control

Chapter 13

Operating system dependent functions

Chapter 14

String handling functions

This chapter describes the functions and commands which operate on Open GENIE strings.

Chapter 15

Maths functions

Maths functions... Sin CosD etc..

Chapter 16

Array handling functions

This chapter gives details of all the GENIE functions specifically designed to be used on array data types.

16.1 The Dimensions() Command

The Dimensions() function is used for dimensioning and returning an Open GENIE array. Currently an array may be created with up to ten dimensions. The number of dimensions for a particular array is determined by the number of integer dimension sizes supplied in the parameter list.

All Open GENIE arrays use 1 as the first index, as a result, the size of an array dimension is also the index value for the last element in that dimension. Arrays are also stored in Row-major order, this means that the most efficient way to access elements in an array is by varying the *rightmost* index most quickly.

By default the Dimensions() function returns an array with no element type assigned and all elements undefined. The array gains its type from the type of the first of its elements to be assigned a value. For example, an empty array becomes a "RealArray" when its first element is assigned a "Real" value. If desired, the Fill() command may be used to initialize an to a sentinel value after it has been dimensioned.

Parameters

P1..P_n)	Dim_n	(Integer)	Size of the nth dimension
RES)		(Array)	An array of the correct dimensionality with undefined elements.

Examples

```
# Create a large one dimensional integer array
arr = dimensions(1000)  # dimension the storage
arr[1] = 455            # Make it an IntegerArray

# Create and zero a 3-dimensional array
A = Fill( Dimensions(1,2,3), 0.0)
```

16.2 The Fill() Command

The Fill() command is used for intializing all elements of an Open GENIE array to the same value. It would normally be used on a newly dimensioned array to set the type and to populate it with sentinel values. It can be used on a previously initialized array, for example, to zero all elements.

Parameters

P1)	Array	(Array)	The array to be filled
P2)	Value	(Basic type)	Value with which to initialise the array
RES)		(Array)	An array of the same dimensionality with all elements replaced by "value"

Examples

```
# Initializing an array to -1.0
Fill Array=J Value=-1.0

# Initializing an array with null strings
Names = Fill( Dimensions(45), "" )
```

16.3 The Fix() Command

The Fix() command is used explicitly to "fix" the values for undefined elements in an array to a defined value. This procedure is largely pragmatic in its intention and ensures that assumptions about undefined elements in the data are made explicitly. Normally these assumptions are made before passing data to external applications. For operations *within* Open GENIE there is no problem; undefined values are treated as a first class data type in all numerical operations.

Parameters

P1)	Array	(Array)	The array to be fixed
P2)	Value	(Basic type)	Value with which to replace all undefined elements
RES)		(Array)	An array of the same dimensionality with all undefined elements replaced by "value"

Examples

```
# Replacing undefined elements in array J with zero.
Fix Array=J Value=0.0

# Changing undefined elements to null strings
Fixed_names = Fix( Name_array, "" )
```

16.4 The Max() Command

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

P1)	Array	(Array)	The array to be searched
RES)		(Simple type)	The largest value found

Examples

```
>> arr = fill(dimensions(20), -100.0)
>> arr[18] = 0.0
>> printn max(arr)
0.000000
>>
```

16.5 The Min() Command

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

P1)	Array	(Array)	The array to be searched
RES)		(Simple type)	The smallest value found

Examples

```
>> arr = fill(dimensions(20), 89)
>> arr[18] = 98
>> printn min(arr)
89
>>
```

16.6 The Sum() Command

The Sum() function returns the sum of all the elements in an array. Again as a matter of convenience, undefined elements are ignored. To be correct, all undefined elements should be "fixed" to a chosen value using the Fix command.

The Sum function has no meaning for arrays of strings but can be used quite properly to sum an array of (congruent) workspaces into a single result.

Parameters

P1)	Array	(Array)	The array to be summed
RES)		(Simple type)	The result of the sum

Examples

```
# summing integers
>> arr = dimensions(20)
>> arr[1] = 1; arr[2] = 2; arr[20] = 3
>> printn sum(arr)
6
```

```
# averaging workspaces
>> set/file "/vax/user/cmm/genie/hrp00273.raw"
```
Default input file: /vax/user/cmm/genie/hrp00273.raw
```
>> w=dimensions(4)
>> LOOP i FROM 1 TO 4
w[i]=s(i)
ENDLOOP
```
GET: Reading spectrum 1 from /vax/user/cmm/genie/hrp00273.raw
GET: Reading spectrum 2 from /vax/user/cmm/genie/hrp00273.raw
GET: Reading spectrum 3 from /vax/user/cmm/genie/hrp00273.raw
GET: Reading spectrum 4 from /vax/user/cmm/genie/hrp00273.raw
```
>> average = sum(w) / 4
```

Chapter 17

General programming functions

This chapter lists functions built into the Open GENIE command language (GCL) used in writing GCL programs but which don't fall directly under the categories in previous chapters.

17.1 The Called_from_function() Command

The Called_from_function() command allows a procedure to determine whether it has been called as a result-returning function or as a keyword command. The most likely use of this function is to give a warning when a function expects to return a result but is called using the keyword syntax which ensures there is nowhere for the result to go!

Parameters

RES)	(Boolean)	Set true if the current procedure was called using the function syntax, false otherwise.

Examples

```
# Simple example of using called_as_function
PROCEDURE test
    IF called_as_function()
        printn "called as a function"
    ELSE
        printn "called with a keyword"
    ENDIF
ENDPROCEDURE

# call in both ways
>> test
called with a keyword
>> a=test()
called as a function
>>
```

17.2 The Now() Command

The Now() command returns a string containing the current date and time in the normal format for the system Open GENIE is being used on.

Parameters

RES)	(String)	Current date and time

Examples

```
>> printn now()
9-Jun-94 12:11:54
>>
```

Appendix A

Supported data file formats

Some good supported formats are...