Prospero Command Procedure Scripts

A toolkit for automating astronomical observing with the *Prospero* package.

Richard Pogge & Paul Martini
The Ohio State University
Department of Astronomy

2001 November 3

Published by The Ohio State University, Department of Astronomy, 140 W. 18th Avenue, Columbus, Ohio 43210-1173

This manual has been composed in Times New Roman font using Microsoft Word. PDF versions were created using Adobe Acrobat v5

Prospero Command Procedure Scripts

Richard Pogge & Paul Martini

The Ohio State University

Department of Astronomy

2001 November 3

User Support

Limited user support for *Prospero* is available via email and the World Wide Web. This support only applies to problems with the *Prospero* package software proper. Problems with the instruments themselves must be referred to the appropriate instrument support personnel. Please report any software problems or bugs to the Prospero email support line:

prospero@astronomy.ohio-state.edu

Replies will be sent at the earliest possible convenience (this is not a support "hot seat").

World Wide Web

Information, additional documentation, a searchable online manual, sample scripts from this manual, and other resources for *Prospero* users are available at the Prospero Homepage on the World Wide Web:

URL: http://www.astronomy.ohio-state.edu/~prospero

Acknowledgements

A number of people have made Prospero possible, including many users who have pointed out errors and omissions, or who have suggested new commands and procedures. All are thanked for their help. In particular, Robert Blum at CTIO has be instrumental in a number of changes which helped support the deployment of OSIRIS at CTIO, and has continued to provide valuable inputs. Andrew Stephens (formerly of OSU, now the Princeton/Catolica fellow) wrote many Prospero scripts and helped with the essential early debugging efforts. Mark Wagner and Ray Bertram did a lot of debugging early in the CCDS deployment at Lowell and later MDM Observatory. Other observers and mountain support personnel who have pointed out errors and omissions include (in no particular order) M. Tavariz, P. Romano, J. Thorstensen, D. DePoy, R. Probst, M. Merrill, J. Frogel, B. Peterson, R. Barr, and H. Roussel,. Our apologies if we have forgotten anyone.

The Prospero command language is based on the XVista command language, much of the syntax of which is due to the efforts of Richard Stover, De Clarke, Don Terndrup, Tod Lauer, and Jon Holtzman. Without their efforts on Vista and later XVista, Prospero would not be what it is today.

Table of Contents

Chapter 1: Introduction

1.1 Overview

Prospero normally works by executing individual commands typed at the keyboard. It is also possible to execute a list of commands stored in an external text file. The ability to execute commands from files, called "Command Procedure Scripts," is one of the most powerful features of *Prospero*. It allows observers to customize *Prospero* to make repetitive or complex observations by writing mini-programs in the *Prospero* interactive command language, rather than having to write (and debug and install) a new Fortran subroutine module for each new task.

In addition to simple scripts, *Prospero* provides a set of specialized functions for doing more advanced procedure control: DO-loops, IF/ELSE conditional branching, GOTO branching, input prompting, message printing, string manipulation, external file I/O, and various types of PAUSE and SLEEP directives. These greatly expand the possibilities of procedure scripts to allowing you to write "programs" in the *Prospero* command language. This makes *Prospero* user-extensible and greatly enhances its utility.

Since *Prospero* is descended from the *XVista* image-processing package, it shares the basic scripting syntax of *XVista*. Even if you don't know *XVista* (and have no intention of learning), you should find the *Prospero* script utility to be very intuitive and easy to learn At the very least, a procedure script can be nothing more than a list of the *Prospero* commands that you would have typed by hand anyway. The difference is that they are stored in a file and executed as a procedure instead of being typed in line-by-line (i.e., literally a "script" of commands to be followed by the program). Much of this manual is devoted to what you can do beyond the simple "one command after the other" type of scripts.

To give credit where credit is due, the true authors of the procedure scripting utility are primarily Richard Stover, Tod Lauer, and Don Terndrup, all of whom were responsible for much of the development work on Versions 1 through 3 of *VISTA* at Lick Observatory during the 1980s. Jon Holtzman (NMSU) has carried on support of *XVista*, with support from many others, and was responsible for version 4.0 upon which *Prospero* is based. Since this guide reproduces much of the contents of the online help pages for the individual scripting commands, this makes them authors, if perhaps only in absentia, and the exact linkages as to who wrote what when have been lost. This work is thus highly derivative, and we (RP and PM) consider ourselves as compilers and commentators as well as authors proper. We also take full responsibility for any and all errors or inconsistencies, and the script examples are entirely our own fault.

1.2 How to use this manual

This manual is divided into two parts. The first, encompassing chapters 2 through 6, defines the basic scripting commands common to both simple ("command-list") and complex ("command program") scripts. Observers interested only in writing simple command-list scripts need only read these chapters. Observers who wish to create more complex command-program scripts, including the use of variables, command-line arguments, and flow control like DO-loops and conditional tests (IF/THEN logical tests), will need also need to read chapters 7 through 10. We give worked out examples of scripts in Chapter 11, but caution the reader that these are meant to be illustrative of different ways to perform observing tasks with scripts of varying complexity, not the only way to do things. *XVista* users who are experienced with scripts in that package should read Chapter 12 describing some important differences in the *Prospero* implementation of scripting.

In the main body of the text, *Prospero* commands are printed in ALL CAPS to distinguish them, although in general *Prospero* commands are case insensitive. Specific examples of commands (or script fragments where those commands only make sense in a particular context, e.g., DO-loops) are typeset in a `typewriter font`. These are usually set apart from the text with indentation to distinguish them as examples of commands you would actually type into a script file or on the command line. When a command is first introduced, its command-line arguments are printed in *Italics* to distinguish them from the command verb proper. If a command-line argument is optional, it is enclosed in of []'s. Numerical arguments may be integers, floating-point numbers, or arithmetic expressions to be evaluated. Character-string arguments longer than one word must be enclosed within single quotes (' s), as spaces are used to separate arguments on the *Prospero* command line.

Since *Prospero* shares *Vista*'s heritage, it also shares some of its peculiarities. If there are especially important considerations, or particular oddities that might trip up the unsuspecting user, we have put these in little boxes labeled "**IMPORTANT!!!**" throughout the text. We have chosen these based on common problems that first-time users often encounter, or known "features" (things you think *should* work but don't). Future editions of this manual will no doubt expand upon these.

Chapter 2: Procedure Script Basics

The basic commands for creating, storing and modifying procedure scripts are:

PEDIT	Edit the contents of the procedure buffer.
WP	Write the procedure buffer to a disk file.
RP	Read a script file on disk into the procedure buffer
SHOW	Show the currently loaded procedure script
RUN	Run the script in the procedure buffer

There are several "control commands" that effect the operation of a procedure.

VERIFY	Executes a procedure line-by-line for debugging.
PAUSE	Pauses during execution of a procedure.
SLEEP	Put a procedure to sleep for a specified interval
CALL	Calls a procedure disk file as a subroutine.
RETURN	Returns from a procedure used as a subroutine.
DO, END_DO	A DO-loop to repeat execution a number of times.
GOTO	Jump to another place in the procedure.
: (colon)	Defines a jump-point in the procedure.
IF, END_IF	Define a block of commands that are executed only under certain conditions.
ELSE, ELSE_IF	Control branching that has many options.
#	Comment character.

A well-written procedure script can eliminate the boredom and errors that come from typing commands repeatedly. However, it does much more than that: it greatly expands the functions of *Prospero* so that new observing or data acquisition applications do not require new internal subroutines. A familiarity with procedure scripts will make observing with *Prospero* much more efficient and free from fatigue- or boredom-induced errors.

Chapter 3: The Startup Script File

When the *Prospero* program is started up, it attempts to execute the startup script stored in the file defined by the UNIX environment variable V_STARTUP (defined in your .cshrc file). For example, if you had defined V_STARTUP to be

```
setenv PR_STARTUP ./myprocs/myproc.pro
```

before running *Prospero*, then ./myprocs/myproc.pro will be executed at *Prospero* start up. Typically, the startup script will contain definitions of personal aliases, settings for particular symbol values, or load repeatedly used images into working image buffers.

Sample Startup Script:

```
# Sample Startup Script
# Defines some useful Unix Aliases
alias ls '$ ls -F'
alias rm '$ rm -i'
alias cp '$ cp -i'
alias mv '$ mv -i'
alias pwd '$ pwd'
alias lpr '$ lpr'
alias lpq '$ lpq'
alias cat '$ cat'
alias more '$ less -m'
alias df '$ df'
alias du '$ du'
end
```

This example uses the ALIAS command to define a set of commonly used Unix operating system commands so that you can execute them from within *Prospero* without having to remember to use the $ shell-escape character. Note that END must be the last command in any procedure script.

Chapter 4: Procedure Files and Directories

Prospero reads procedure scripts into an internal "procedure buffer" before execution (with the exception of the startup procedure). External procedure scripts may be loaded from files into the buffer, written from this buffer to external files, or the contents of the buffer may be edited directly. The contents of this buffer are erased when a *Prospero* session ends.

Procedure files are simply ASCII text files containing *Prospero* commands laid out as if they were the source code for a program written in the *Prospero* command language. They must contain no special characters (control or ESC characters), and must use normal carriage-control and line-feed delimiters to mark the ends of lines. You must edit them outside of *Prospero*, or use the $ shell escape character to invoke your favorite editor.

IMPORTANT!!!

At present, *Prospero* procedure scripts are limited to **1024** lines, with each line no more than **80** characters long.

Procedures may contain blank lines to make things readable, and many observers indent DO-Loops or IF-Blocks to "structure" the program and make it more readable. Rules for flow control, I/O etc. are similar to Fortran-77 conventions, but logical operators use more intuitive symbols (e.g., >, <, etc.) rather than Fortran ".le." constructs.

Comments may be placed in a command procedure to auto-document them using the "#" character in the first column. The "#" character may also be used on a command line to comment that line, for example:

```
# this is a procedure to do stuff
#
rd 1 myimage.fits  # read in myimage.fits
...
end
```

Note that any comments you put in the procedure file are counted towards the maximum. Blank lines also count against you, so don't go too crazy with them, either.

By default, *Prospero* will search for procedure files in the directory defined prior to startup by the "PR_PRODIR" environment variable. This variable may also be redefined in mid-session by executing the SETDIR command.

Procedure files are assigned the ".pro" filename extension by default. This, too, may be redefined using the SETDIR command. If you intend to use a custom procedure file extension, we recommend you put an appropriate SETDIR command into your startup procedure file.

Chapter 5: Procedure Commands

5.1 PEDIT: edit the procedure buffer

Usage: pedit

The command PEDIT loads the procedure buffer into a temporary file in your current directory, then runs a process that allows you to edit it with your designated screen editor. If you leave the editor with EXIT (i.e., save all changes and exit), the modified procedure is loaded back into the procedure buffer, but not executed. If you leave the editor with QUIT (i.e., ignore changes and abort), the contents of the procedure buffer are left unchanged.

By default, PEDIT uses the editor defined at startup by the PR_EDITOR environment variable. If either PR_EDITOR or VISUAL (an environment variable commonly defined by the .cshrc shell script) is undefined, *Prospero* defaults to the generic vi editor expected to be found on any rational Unix system in a canonical place. You can use another editor either by defining the PR_EDITOR environment variable **before** executing *Prospero*, or by using the SETEDITOR command from within *Prospero*, which lets you select between at least vi and Emacs.

For example, to use the Emacs editor, you might define

```
setenv PR_EDITOR /usr/local/bin/emacs
```

before running *Prospero*, or once within *Prospero*, executing the command:

```
SETEDITOR emacs
```

Note: most *Prospero* installations are such that *Prospero* is executed by way of a shell script that defines all local environment variables for the observer.

Procedures may contain no more than 1024 lines of not more than 80 characters each. If you leave the editor and your procedure contains more than 1024 lines, it is truncated and the last line is set to 'END'. Also, each line is allowed a maximum length of 80 characters.

You may also define procedures (also with a maximum length of 1024 lines) using the editor **before** running *Prospero*, and read them into the procedure buffer with RP or execute them with CALL. Many observers build up a collection of custom procedures in this way.

5.2 SHOW: list the contents of the procedure buffer

Usage: SHOW [output redirection]

SHOW prints the current contents of the procedure buffer. The output may be redirected, as in

```
SHOW >special.pro
```

(Note that there is no space between the > and the filename, cf. §10.8) To store a procedure in the buffer to a disk file, however, it is better to use the WP command.

5.3 WP: write the contents of the procedure buffer to disk

Usage: WP *filename*

where:

filename is the name of the procedure file to create

Unless otherwise specified in "*filename*", WP will write to the default directory for procedures (defined by the PR_PRODIR environment variable), and will be given the ".pro" filename extension if none is given. The current definitions of the default procedure directory and default file extension may be viewed using the PRINT DIR command, or changed using the SETDIR PR DIR= EXT= command.

Examples:

In the following, the `default` procedure directory path is `~/Prospero/proc/` and the default file extension is "`.pro`":

```
WP MYSCRIPT
```

writes the procedure file `~/Prospero/proc/MYSCRIPT.pro` (note the case of "`MYSCRIPT`": all Unix filenames are case sensitive!).

```
WP ./myscript
```

writes the procedure into the file "`./myscript.pro`" in the current working directory.

```
WP myscript.XYZ
```

writes "`myscript.XYZ`" into the `~/Prospero/proc/` directory.

5.4 RP: read a procedure file from disk

Usage: RP *filename*

where:

filename is the name of the procedure file to read in

RP will read a maximum of 1024 lines, with a maximum of 80 characters per line, from the designated filename into the procedure buffer.

Unless otherwise specified in "filename', RP will read from the default directory for procedures (defined by the PR_PRODIR environment variable), and will be given the ".pro" filename extension if none is given. The current definitions of the default procedure directory

and default file extension may be viewed using the PRINT DIR command, or changed using the SETDIR PR DIR= EXT= command.

Examples:

In the following, the default `procedure` directory path is `~/Prospero/proc/` and the default file extension is "`.pro`":

```
RP MYSCRIPT
```

reads the file `~/Prospero/proc/MYSCRIPT.pro` into the procedure buffer, wiping out the previous contents.

```
RP ./myscript
```

reads the file "`./myscript.pro`" in the current working directory into the procedure buffer (erasing the contents of the `buffer`).

```
RP myscript.XYZ
```

reads the file "`~/Prospero/proc/myscript.XYZ`" into the procedure buffer.

IMPORTANT!!!

All files used by *Prospero* must have file extensions, even if Unix allows for extension-less filenames.

5.5 RUN: Execute the procedure in the procedure buffer

Usage: RUN [*parameter1*] [*parameter2*] ...

where:

parameter1,2,...are parameters passed to the procedure.

RUN tells *Prospero* to start executing the procedure held in its procedure buffer. You may also supply parameters to the procedure on the command line. The parameters must be evaluated using the PARAMETER command (§6.5) in the procedure. See also the CALL command.

Examples:

```
RUN 10
```

executes the procedure in the buffer, passing the numeric parameter 10 to the procedure.

```
RUN /usr1/data1/image
```

executes the procedure, passing the string parameter to the procedure.

5.6 VERIFY: verify execution of a procedure (trace-mode)

Usage: VERIFY Y or VERIFY N

VERIFY echoes each line of an executing procedure to the terminal screen as it is executed, allowing you to trace the operation of the procedure line-by-line. The keyword "Y" turns the display on and the keyword "N" turns the display off. This is useful for debugging a procedure, and is set to "No" by default.

Chapter 6: Procedure Execution Control

6.1 CALL: call and execute a procedure file as a subroutine

Usage: CALL *procfile* [*parameter1*] [*parameter2*] ...

where:

procfile is the name of a procedure file
parameter1,*2*,... are optional parameters to be passed

The CALL command tells *Prospero* to save the contents of its current procedure buffer, read in the desired procedure file, and begin execution at its first line. The CALL command can be executed directly in the immediate input mode, or be used inside procedures to call other procedures. In both cases, at the completion of the called procedure, *Prospero* will return properly to either the input mode or calling procedure. *Prospero* will support up to 10 levels or subroutine calls. If an error occurs while a called procedure is executing, *Prospero* will unwind and display the complete subroutine stack.

A procedure is allowed a maximum length of 1024 lines, and each line is allowed a maximum length of 80 characters. If your procedures or lines exceed this maximum, they will be truncated.

Optional parameters may be passed to the procedure. Parameters may be numeric or string arguments. Command-line parameters are evaluated by the procedure using the PARAMETER command. The sole limitation is that the order of command-line arguments for the procedure is fixed by the order used in the procedure's PARAMETER statement. See §6.5 for details.

Suppose the default procedure is ~/*Prospero*/proc/ and the default extension is .pro, then:

Examples:

```
CALL myscript
```

executes ~/Prospero/proc/myscript.pro

```
CALL ./myscript
```

executes `./myscript.pro`

```
CALL ./myscript.txt
```

executes `./myscript.txt`

```
CALL myscript kill
```

executes `~/Prospero/proc/myscript.pro`, passing it the parameter "`kill`".

If the script being called as a subroutine requires command-line arguments and you wish to use variables or strings to pass those arguments, the variables and strings are passed to the called script "unqualified". Thus

```
CALL mysubscript n t '{title}'
```

is the correct calling syntax for using "mysubscript" as a subroutine within another script, passing the numerical variables "n" and "t", and the string variable "{title}" as shown. Note, however, that the syntax

```
CALL mysubscript $n %t '{title}'
```

is invalid (here variable "n" has been qualified as integer, and "t" as a float). Variable qualification is discussed in §8.6 below.

6.2 END: end a procedure

Usage: END

All procedures must end with END or RETURN.

This command signals the end of the current procedure when it is executed. If the procedure is executed as a subroutine of another procedure (using the CALL command), the END command, like RETURN (§6.4), tells *Prospero* to return to the calling procedure.

A procedure may be prematurely aborted by hitting the Ctrl-C key.

6.3 STOP: stop procedure execution

Usage: STOP [*'A message'*]

The STOP command causes a procedure to stop executing. If a message is supplied, it will be typed on the terminal, otherwise a default message is typed. A stack unwind is also produced if the stop occurs in a CALLed procedure.

6.4 RETURN: return from a procedure subroutine

Usage: RETURN

RETURN tells *Prospero* to halt execution of the current sub-procedure and return to any calling procedure or back to the normal command-mode as is appropriate. This command is intended to allow a return from the procedure as a result of an IF test. In cases where no condition testing is required, the final END command in the procedure buffer will tell *Prospero* that the procedure has completed and act like an implicit RETURN statement.

Examples:

Here is an example of a conditional exit from a procedure. This procedure may be run with CALL or RUN.

```
LOOP:
ASK 'Enter 0 (zero) to quit >> ' TEST
IF TEST=0
      RETURN
END_IF
...
GOTO LOOP:
```

Note: END may be used as RETURN *only* if it is the *last* command in the file. Thus a procedure like:

```
(commands)
END
```

may be executed as a subroutine, but one that looks like:

```
IF TEST=0
      END
END_IF
```

will fail. In this instance you must use RETURN instead of END to correctly exit the loop.

6.5 PARAMETER: evaluate command-line parameters

Usage: PARAMETER [*varname*] [*varname*] [*STRING=string_var*] ...

PARAMETER evaluates parameters to be passed to a procedure. For example, suppose that the following command was typed:

```
CALL test 2 X IMAGEFILE
```

The parameter list portion of the command line, `2 X IMAGEFILE`, is saved, then the procedure file named "`test.pro`" is read in and executed ("called"). Note that on a Unix system, procedure names are filenames, and therefore are *case sensitive*.

In order to be able to process the command line arguments given above, the test.pro procedure has in it the command:

```
PARAMETER BUFNUM FACTOR STRING=FILENAME
```

`BUFNUM` and `FACTOR` are taken to be variable names and are associated with the first two parameters from the parameter list. In effect it does the same as the command "`SET BUFNUM=2 FACTOR=X`." Note that "`2`" and "`X`" could have been any arithmetic expression.

The `STRING=` keyword means that the third parameter is used as a literal string and the string variable name `FILENAME` is given the string value `IMAGENAME`, in effect executing the command:

```
STRING FILENAME IMAGENAME
```

If there are fewer parameters in the CALL than required in the PARAMETER command, the missing parameters are given default values of 0 (if numbers) or blank (if strings), as appropriate. If more arguments are given than expected, an error results.

Note that there is only one common area in *Prospero* for saving lists of passed parameters, so the PARAMETER command must be executed *before* calling another command procedure as a subroutine. In general, the PARAMETER line must be the *first non-comment line* in any given script file.

A further note on script files used as subroutines. All *Prospero* variables are *global*, in the sense that if you define a variable `X` to have a particular value in one procedure, and then call a second that also uses variable `X`, they will be assumed to be the *same variable.* This means you should be very careful about what variables you use in scripts designed to be called as subroutines of other scripts.

6.6 "#" Insert a comment line into a script

Usage: # *text with comment*

The # character is used to denote comments, and all text following the # (including the #) will be ignored. This is useful for temporarily removing commands from a procedure, for example:

```
DO I=1,20
      #  PRINTF 'Displaying Image %I' I
      TV $I Z=0. L=100.
      PAUSE 'Hit C to continue'
END_DO
```

in which the procedure will no longer print "Displaying Image ..." on each pass through the loop. It also provides a way to internally document the operation of a procedure script, for example:

```
#
# Display images 1-20 in Box 1
#
BOX 1 SR=1 NR=512 SC=15 NC=25
DO I=1,20
      MEAN $I BOX=1 # compute the mean in box 1
      TV $I Z=0. L=2*MEAN # display w/span=2*MEAN
      PAUSE 'Hit C to continue'
END_DO
```

in which the # is used to insert a header explaining what the procedure does, and in-line comments on particular commands.

Note that all comments count against the 1024-line maximum for *Prospero* scripts.

Chapter 7: Flow Control in Procedures (Loops, Conditional Tests, & Branching)

7.1 PAUSE: pausing during procedure execution

Usage: PAUSE 'prompt message'

When the PAUSE command is encountered in a script, *Prospero* prints the prompt "PAUSE" followed by the rest of the (optional) prompt that appears on the command line. The execution of the procedure is then paused, any regular (i.e., non-script) Prospero commands at the command prompt. To resume the script from the point where it was paused, type the CONTINUE command. If, however, you give the command RUN or CALL, any previous pause state will be canceled, and *Prospero* will start the new script from the beginning.

There are two important provisos when using PAUSE:

1. The PAUSE command must always be placed on its own line in a script. For example, a procedure that looks like:

   ```
   Command_1
   Command_2; PAUSE; command_3
   Command_4
   ```

   will pause properly, but CONTINUE will resume the script at `command_4` not `command_3`!

2. While a script is paused you can issue any regular Prospero commands, *but you cannot execute another script*. If you call (or run) another script while a PAUSE is active, Prospero will reset the script command stack, essentially forgetting the previous script responsible for the PAUSE! A subsequent CONTINUE statement will not restart the old script. Be especially careful if you make overly liberal use of command aliases, especially if an alias executes a script "call" statement.

If you do not need to execute interactive commands, but simply want to halt script execution to wait for some action to proceed, see the WAIT or SLEEP commands below. They reduce the temptation to do something bad that "breaks" a pause so that you cannot resume execution of the script.

7.2 CONTINUE: resume a PAUSEd procedure

Usage: CONTINUE

Re-starts a procedure stopped by PAUSE. See PAUSE for more information.

7.3 SLEEP: Put a procedure to sleep for a given time interval

Usage: SLEEP *seconds* [*'sleep message'*] [SILENT]

where:

seconds is the time interval to sleep for in seconds
'sleep message' is an optional message to print while sleeping
SILENT tells SLEEP to work silently (no messages)

SLEEP will put *Prospero* to sleep for the specified time interval in seconds. This is useful for inserting delays in time-critical tasks. Once the specified interval has elapsed, execution will resume with the next command. SLEEP accepts time intervals from 1 to 300 seconds in duration.

Prospero will either print the optional sleep message provided in quotes on the command line, or the generic message:

```
Sleeping for xxxx seconds...
```

Messages are limited to 64 characters in length, and multi-word messages must be contained within single quotes ('s).

If the SILENT keyword is used, no sleep message will be printed to the terminal screen, and any optional message included on the command line will be ignored.

SLEEP may be interrupted by a Ctrl-C to shorten the nap. This Ctrl-C will **not** terminate the procedure, but will return a message giving the time remaining when it was prematurely awakened.

Examples:

```
SLEEP 10
```

Prints "`Sleeping for  10 seconds...`", and then resumes after a 10-second pause.

```
SLEEP 5 'Waiting 5 sec for scope to settle...'
```

Prints "`Waiting 5 sec for scope to settle...`", then resumes execution after a 5-second pause.

Note: SLEEP "keeps one eye open" to watch for any messages from the data-taking system sent during its nap. If the instrument control system (the data-taking PCs) generates any message traffic during SLEEP, you may see occasional output on the terminal or (if enabled) the status display may update.

7.4 WAIT: Suspend a procedure and wait for the RETURN key

Usage: WAIT [*'wait message'*] [BELL]

where:

'wait message' is an optional message to print while it waits
BELL rings the bell to get the observer's attention

WAIT will suspend operation of a command procedure script, and then print the wait message (or some default message) and wait until either

1. The observer hits RETURN, at which point execution of the procedure continues to the next command.

2. The observer hits Ctrl-C and the procedure terminates prematurely.

WAIT is used in procedures to tell the observer to go do something like setup the autoguider, close the door, or any other manual operation that (a) *Prospero* cannot do itself and (b) must be done before the observing procedure may continue.

Unlike PAUSE (which puts you temporarily back into the command prompt) or SLEEP (which just takes a nap for a time), WAIT only requires a RETURN key to continue the procedure.

The BELL keyword will add an aural cue to the WAIT command.

Examples:

```
WAIT
```

Prints out "`Procedure Waiting: Hit RETURN to Continue`", and then waits for the observer to hit the RETURN key.

```
WAIT 'Re-Acquire the Guide Star and hit RETURN' BELL
```

Rings the bell and prints the message, and then waits for the observer to hit the RETURN key.

Note: Like SLEEP, WAIT will watch for any data-taking system messages while waiting for a response from the observer. If the instrument control system is generating any message traffic during the WAIT, you may see occasional output on the terminal or (if enabled) the status display may update.

7.5 ALERT: Print an alert message to the screen then continue

Usage: alert *'alert message'* [*ntimes*] [*tdelay*]

where:

'alert message' an alert message to print on each alert
ntimes is the number of times to repeat the alert
tdelay is a time delay, in seconds, to insert between alert repeats

ALERT will ring the console bell and print an alert message to get the observer's attention and then continue execution. It can be used to insert "wake-up calls" in long procedures, to add an aural cue to call attention to input or error messages that need servicing, etc. By default, the alert will print once, wait 3 seconds, and then continue execution.

The number of repeats (ntimes) and the inter-repeat delay time (tdelay) may be set on the command line, thus:

```
alert 'WAKE UP!' 5 1
```

rings the bell and prints "WAKE UP!" 5 times, pausing 1 second between alerts. Rude but effective.

Like SLEEP, an ALERT may be canceled with a Ctrl-C without terminating the procedure script that executed the ALERT.

Examples:

```
ALERT 'Exposure Done!' 3 2
```

Prints out "`Exposure Done!`", rings the bell once every 2 seconds three times, then continues the procedure

```
ALERT 'Object Sequence Completed...'
```

Prints out "`Object Sequence Completed...`", rings the bell, and then waits 3 seconds before executing the next command in the script.

7.6 GOTO: jump to a labeled place in a procedure

Usage: GOTO *LABEL_NAME*

where:

LABEL_NAME is a label defined somewhere else in the procedure.

GOTO tells *Prospero* to jump to the line in the procedure buffer that begins with the string "LABEL_NAME:", where "LABEL_NAME" can be any alphanumeric string terminated with a colon (:) to mark it as a label. You can jump out of, but not into, a procedure DO-Loop or IF-block. If you attempt to break into a DO-Loop or IF-block, *Prospero* will signal an error

condition and stop the procedure execution. Jumps to labels may be forwards or backwards within a procedure.

A simple example of using a GOTO to jump over some lines of a procedure:

```
GOTO WHEREVER
A number of procedure lines to jump...
WHEREVER:
The next commands to be executed...
```

This is an example of a sloppy infinite loop using a GOTO:

```
NOWHERE:
A number of procedure lines...
GOTO NOWHERE
```

7.7 ":" label a line as a GOTO jump-point

Usage: LABEL_NAME:

To label a procedure line as jump-point for the GOTO command, the line must start with the LABEL_NAME string terminated with a colon (":") immediately following the label (no spaces). No other commands can appear on the same line as a label.

7.8 DO Loops

Usage:

DO *var=N1,N2,N3*
any *Prospero* commands
END_DO

where:
var is a *Prospero* variable name,
N1 is the initial value of the variable,
N2 is the final value of the variable,
N3 is the amount to increment the variable by on each pass

The DO commands enable you set setup repeatable groups of commands within the procedure buffer. The *Prospero* DO-Loop is very similar to the Fortran-77 DO-Loop.

The variable "*var*" is initially set to the starting value *N1*. When the END_DO statement is encountered the value is changed by an amount equal to *N3*. The value of *N3* can be either positive or negative. If *N3* is positive then the looping terminates when *N1* becomes greater than *N2*. If *N3* is negative then looping terminates when *N1* becomes less than *N2*. If *N3* is not specified then it defaults to +1 if *N2* is greater than *N1* or to –1 if *N2* is less than *N1*.

N1, *N2*, and *N3* can also be arithmetic expressions. The value of "var" can be changed within the loop without affecting the DO-Loop operation. However, *Prospero* will reset it to its appropriate loop value at the beginning of each loop.

The underscore character is required in END_DO because *Prospero* uses spaces to delimit commands from keywords. DO-Loops may be nested up to 20 deep. DO-Loops are recognized only within procedures, and cannot be executed "by-hand". The GOTO command can be used to jump out of a DO-Loop, but it cannot be used to jump INTO one. Further, DO-Loops must contain, or be contained completely within, any IF/THEN blocks. These rules should be familiar from standard Fortran or C programming style.

Example 1:

```
DO I=1,3
      Any number of procedure lines.  These lines
      are executed 3 times.
END_DO
```

Example 2:

```
DO Q=1,N
      A number of procedure lines.  These lines
      are executed N times.  N is a variable
      previously defined using the SET command.
END_DO
```

Example 3:

```
DO B=D+I,N-J,-1
      Any number of procedure lines.  The
      Counter decrements from D+I to N-J.
END_DO
```

IMPORTANT!!!

Flow-control statements (DO, IF, END_DO, etc.) ***cannot*** appear on a line with other commands (using the ; construct). They must appear on *separate* lines. For example:

```
DO I=1,10
     DO J=1,10
     (some stuff)
END_DO; END_DO
```

or similar constructions are wrong! Each END_DO must appear on a line by itself. See Chapter 0 for details.

7.9 IF/THEN Logical Flow Control

Prospero procedures allow testing of variables and branching based on the results of those tests. This capability greatly expands the usefulness of procedures.

The simplest use of IF is to mark a section of a procedure that is executed only if some logical condition is true. It has the form:

```
IF condition
      Procedure lines (any number) executed if condition is true.
END_IF
```

You can also have two-level branching using ELSE:

```
IF condition
      Procedure lines to be executed if condition is true.
ELSE
      Procedure lines to be executed if condition is false.
END_IF
```

Finally, the ELSE_IF command lets you create a multi-level IF/ELSE block as follows:

```
IF condition1
      Procedure lines executed when only condition1 is true.
ELSE_IF condition2
      Procedure lines executed when only condition2 is true.
ELSE_IF conditionN
      Lines executed when only conditionN is true.
ELSE
      Lines to be executed if all other conditions are false.
END_IF
```

The conditions tested by the IF and ELSE_IF statements are any valid *Prospero* logical expressions. An expression is considered to be "TRUE" if it evaluates to be non-zero, and "FALSE" otherwise. *Prospero* arithmetic supports various logical operators whose value is either 1 or 0 depending on whether the logical test is TRUE or FALSE.

The logical operators are as follows, where A and B can represent either *Prospero* variables or arithmetic expressions.

`IF A>B`	Test A greater than B
`IF A>=B`	Test A greater than or equal to B
`IF A==B`	Test A equal to B
`IF A~=B`	Test A not equal to B
`IF A<=B`	Test A less than or equal to B
`IF A<B`	Test A less than B

There are two logical (Boolean) conjunctions:

`&`	Boolean AND
`|`	Boolean OR

that can be used to join several of the above tests. Examples of these conjunctions are below:

`IF (A>B)&(A==C)`	Tests A>B AND A=C
`IF ((A==B)|(C<D))&(C==B)`	Tests (A=B OR C<D) AND C= B

Parentheses are used to set the order of the test, as is common to most programming languages.

The syntax of the IF statements is designed to look similar to the Fortran-77 IF-block structures (actually, it is closer to C in logical syntax). Each IF-block must begin with an IF command and end with the END_IF command. An algebraic statement to be tested must follow the IF on the same line. If the relation is true, then the procedure commands following the IF command are executed. If the relation is false, *Prospero* looks for any ELSE_IF tests, any final ELSE statement, or jumps to the procedure lines following the END_IF statement.

The ELSE_IF command also must have a condition to be tested on the same line. ELSE_IF's are optional, but permit you to test other conditions and execute other blocks of the procedure buffer in the event that the initial IF or any preceding ELSE_IF's are false. In this way you can allow *Prospero* to "trickle down' through several tests looking for one that is true.

The ELSE statement is also optional and marks a set of procedure lines for *Prospero* to execute if and only if the initial IF and any following ELSE_IF's all test out false. Basically, the IF, any ELSE_IF's, or any ELSE statements all mark out various blocks of the procedure to be executed under different conditions. After the execution of any block, *Prospero* transfers control to the procedure lines following the END_IF statement.

IF-blocks can be nested within other IF-blocks up to 15 levels deep. The GOTO command may jump into, but not out of, IF-blocks. IF-blocks must contain or be contained within DO loops completely. Some examples of IF blocks are given below:

Example 1:

```
IF X>Y
      Do these procedure lines if X > Y
END_IF
```

Example 2:

```
IF (X>Y)&(X<Z)
      Do these procedure lines if X > Y
      but less than Z.
ELSE
      Otherwise jump to these procedure lines.
END_IF
```

Example 3:

```
IF SKY-LIMIT>BACKGRND
      Do these procedure lines if test true.
ELSE_IF BACKGRND==0
      Do these procedure lines if test false,
      but ELSE_IF condition true.
END_IF
```

Example 4:

```
IF IMAGE-1
      Do these procedure lines if IMAGE is not
      equal to 1 (which would make the
```

```
      expression evaluate to 0).
END_IF
```

7.10 ERROR: execute on error

Usage: ERROR *command*

where:

command is any valid *Prospero* command.

ERROR tells *Prospero* to execute the given *Prospero* command whenever an execution error occurs. The command can be any *Prospero* command but is quite often a GOTO command. The command to execute *cannot* contain multiple commands separated by semicolons (;).

For example:

```
ERROR GOTO WHEREVER
Any number of procedure lines...
WHEREVER:
A group of commands to only be executed after an error has
occurred...
```

In this example, ERROR is told to set a variable ERRFLAG to 1 if an error occurs:

```
ERROR ERRFLAG=1
```

In this instance, the rest of the procedure might periodically check ERRFLAG before taking additional steps.

7.11 EOF: execute on End-of-File (EOF)

Usage: EOF *command*

where:

command is any valid *Prospero* command.

EOF tells *Prospero* to execute the given *Prospero* command whenever an end of file is encountered in an opened ASCII data file. See the OPEN and READ commands (Chapter 10) for information on how to use ASCII data files. The command can be any *Prospero* command but is quite often a GOTO command (§7.6). The command *cannot* contain multiple commands separated by semi- colons.

For example, here we tell the script to jump to "WHEREVER" immediately upon detecting an EOF during an ASCII file read:

```
EOF GOTO WHEREVER
Any number of procedure lines...
WHEREVER:
The commands to be executed only after an end-of-file is detected
```

In this example, detecting an EOF will set the *Prospero* variable EOF_FLAG to 1 on detecting an EOF:

```
EOF EOF_FLAG=1
```

This would go at the top of a procedure, and the rest of the procedure would then periodically check the value of EOF_FLAG before proceeding to trap EOFs.

Chapter 8: *Prospero* Variables

Prospero Variables are character names to which are associated values. You get access to the value by using the name. The following commands are used in *Prospero* to manipulate variables:

SET	defines the value of a variable, either directly or by evaluating arithmetic operations on other variables.
TYPE	print the value of a variable or arithmetic expression.
ASK	asks (prompts) for data to be entered at the keyboard.
PRINTF	formatted printing of variables and strings.
STRING	define a string variable (with or without prompting)

8.1 Arithmetic Expressions

Here we review the syntax of mathematical expressions. Any observer who wishes to use *Prospero* effectively should review this section carefully.

Arithmetic expressions come in four types:

EXPLICIT NUMERIC VALUES, such as:

```
3.14159
59.39
2.32E-05
```

VARIABLE NAMES with values assigned with the SET command. Variable names must be composed of alphanumeric characters. There can be up to seven characters in the name. The first character of the variable name should be an alphabetic character.

DATA FILE REFERENCES that allow you to use numeric values from an ASCII text file. Data file references begin with the "@" symbol followed by the file name, a period (“.”), and then the column number. The column number can itself be either a numeric integer constant or a variable name. An example is given in the description of the READ command (§10.3).

CONSTANTS or **VARIABLES** used in arithmetic expressions or on the command line.

The following table lists arithmetic functions supported in *Prospero*, with examples of their use. ('Binary' means relating two objects—“unary' means applying to one object.)

binary addition	`B+C`	`2+5`
binary subtraction	`B-C`	`2-5`
binary multiplication	`B*C`	`2*5`
binary division	`B/C`	`2-5`
unary negation	`-B`	`-2`
exponentiation	`A^0.5`	`10^3`
equate	`A=B`	`A=B=3`

The following logical operators are also supported.

logical greater than	A>B	A>2.5
logical less than	A<B	A<100
logical equal to	A==B	A==10
logical not equal to	A~=B	A~=10
logical less than or equal to	A<=B	A<=5
logical greater than or equal to	A>=B	A>=3

Logical operators return a value of either 1 (TRUE) or 0 (FALSE). Expressions are evaluated in the same order as they are in FORTRAN. You may change the order of evaluation using parentheses "()". An example is (B+0.53)*10^(45.6/(A+5))

IMPORTANT!!!

There must be **no spaces** between operators and arguments!

Prospero supports a limited set of mathematical functions. The arguments to the function are contained in square brackets. The following functions are currently supported:

Arithmetic Functions	
INT[E]	nearest integer to the expression E
ABS[E]	absolute value of E
MOD[E,I]	E modulo I

IFIX[E]	integer part of E (truncation)
MAX[E,F]	the larger of E or F
MIN[E,F]	the smaller of E or F
LOG10[E]	log to the base 10 of E
LOGE[E]	log to the base e of E
EXP[E]	e raised to the power E (Use ^ for all other exponentiations)
SQRT[E]	square root of absolute value of E
Trigonometric Functions	
SIN[E]	sine of E (E in radians)
SIND[E]	sine of E (E in degrees)
COS[E]	cosine of E (E in radians)
COSD[E]	cosine of E (E in degrees)
ARCTAN[E]	arctan of E, producing radians
ARCTAND[E]	arctan of E, producing degrees
ARCCOS[E]	arccos of E, producing radians
ARCCOSD[E]	arccos of E, producing degrees
Image Parameters	
NR[B]	number of rows of the object in buffer B.
NC[B]	number of columns of ...
SR[B]	start row ...
SC[B]	start column ...
EXPOS[B]	exposure time ...
RA[B]	right ascension in sec of time of ...
DEC[B]	declination in sec of arc ...

ZENITH[B]	zenith distance in radians ...
UT[B]	universal time of mid-exposure in hours ...
GETVAL[I,R,C]	returns the value of the pixel at row R and column C in image I
SETVAL[I,R,C,V]	returns the value of the pixel at row R and column C in image I, then sets the value of that pixel to V.
WL[I,P]	returns the wavelength of pixel P in image I.
PIX[I,W]	returns the pixel number corresponding to the wavelength W in image I.

Functions can appear inside other functions, and may contain expressions of arbitrary complexity (so long as the parentheses are balanced!).

```
PLOT 4 R=200 XS=X XE=X+50
```

Note that R=, XS=, and XE= are command keywords, not variables, and the equals signs following them are not interpreted as arithmetic operations. However, since everything following the first = is an arithmetic expression, you could save the "value" of, for instance, XE, by doing the following:

```
PLOT 4 R=200 XS=X XE=XLAST=X+50
```

The variable XLAST takes on the value of X+50.

IMPORTANT!!!

Variable names or arithmetic expressions may be used **anywhere** that an explicit constant may be used as the value of a keyword.

8.2 SET: define a Prospero variable and give it a value

Usage: SET *VAR_NAME=value* [*VAR_NAME=value*]

where:

VAR_NAME is the name of the variable being defined, and
value is its value.

SET defines *Prospero* variables in terms of numerical constants, other variables, or the result of arithmetic operations between other variables. The name of a *Prospero* variable is any alphanumeric string. The value of the variable is always a floating-point number. *Prospero* supports an internal variable table that holds variables defined by you or as the output of a program. These variables can be used to pass the results of arithmetic calculations to

keywords, to control the flow of a procedure in IF tests or DO loops, or to store convenient numbers in symbolic form.

Each SET command can handle up to 15 definitions at once. Each definition must include an "=" sign with the name of the new variable to its left, and a defining expression to its right. The expression on the right may be any proper *Prospero* arithmetic expression (see §8.1 for rules on expressions). All operations are done in double precision floating point.

Examples:

```
SET Q=6
```

Sets Q to have the value 6

```
SET A=1 B=3 C=D=6
```

Sets several variables at once

```
SET V=SIND[45]
```

Functions may be used

```
SET B=3.1415926^0.5+4
```

Any arithmetic expression may be used.

```
SET C=LOG10[@FILE.1]
```

References to data from files may be used.

Note: There must be no spaces between the beginning of "VAR_NAME" and the end of "value".

Note that the SET command may in general be omitted when defining variables. Use of this "implicit SET" feature will save time and typing. The SET command has been retained to ensure backward compatibility with older versions of *Prospero* (and *Vista*).

IMPORTANT!!!

All *Prospero* variables are *GLOBAL* floating-point variables. This means that if you use the same variable name in two different subroutines with one called by the other, *Prospero* will assume they are the *same* variable.

8.3 TYPE: evaluate an expression and print it

Usage: TYPE *expression* [*expression*] [*expression*] ...

This command can be used to print out arithmetic expressions. Up to 15 expressions may appear at one time on the command line.

Examples:

```
TYPE X
```

Evaluates X (a variable) and prints the value.

```
TYPE X+0.5^3.4
```

Evaluates the expression shows and prints the value.

```
TYPE X X*2 X*4
```

Evaluates the 3 expressions & prints the results

NOTE: The expressions that TYPE will accept must have at least one non-numeric character in them. This means that the expressions must contain at least one variable name OR at least one arithmetic operator.

Thus, something like

```
TYPE 6.3
```

will not work.

The command PRINTF (§8.7) is used to print expressions in a particular format.

8.4 ASK: prompt for a variable on the console

Usage: ASK [*'An optional prompt in quotes'*] *VAR_NAME*

This command can be used to request the input of variable values during the execution of a procedure. When the ASK command is executed, the prompt will be displayed at the terminal until the requested value is typed in. If no prompt is given, the command will respond with "`ENTER VAR_NAME:`" and wait for you to respond to the prompt by typing a value and hitting the return key. Only one value can be requested per ASK command.

Examples:

```
ASK BCKGND
```

will print "`ENTER BCKGND :`" on your screen. When you enter an number and hit RETURN, the value of BCKGND will be set to the number you specified.

```
ASK 'Enter an estimate for the background >> '  BCKGND
```

will type the prompt "`Enter an estimate for the background >>`" on your terminal, and wait for you to enter an expression; the value of BCKGND is the value of that expression.

Note that the reply typed in response to ASK can be any valid expression.

8.5 YORN: Ask a "Yes or No" question.

Usage: YORN [*'An optional prompt in quotes'*]

YORN prompts the user to answer a Yes or No (YorN) question. YORN takes a single argument, a question prompt that must be enclosed in single quotes ('s).

The response to the question is case-insensitive, and only the first character of the answer is tested. Thus

"YES" = yes, Yes, y, Y, etc.

"NO" = No, no, n, N, etc.

are all valid responses. If an invalid response is given, it will ask the question again.

On answering either Yes or No, YORN sets the Prospero variable YORN to either 1 (Yes) or 0 (No). Recall that 1 is a logical "TRUE" in Prospero, while 0 is a logical "FALSE". An example of how YORN is used in a script is as follows:

```
YORN 'Do it'
if (YORN)
   printf 'doing it…'
else
   printf 'not doing it…'
end_if
end
```

When this script is run, the observer sees:

```
Do it <Y|N> ?
```

Responding with "Y" will print

```
doing it…
```

Responding with "N" will print

```
not doing it…
```

A common mistake is to forget to put quotes around the prompt string. This will cause only the first word of the prompt to appear.

8.6 Using variables to substitute for numerical command-line arguments

Usage:

COMMAND *$varname* [rest of command]
COMMAND *%varname* [rest of command]

Numerical arguments by themselves on the command line (that is, numbers given as arguments of a command without being part of a "KEYWORD=VALUE" token) are used by a number of commands, and may be either integers or floating-point numbers. Examples are integer device positions (e.g., FILTER 2) and decimal parameter values (e.g., EXPTIME 4.3).

In order to use a variable to substitute for a bare numerical argument, it needs to be "qualified" as either an integer or floating-point number. If the variable is to be an integer, a $ (dollar sign) is prepended to the variable name to qualify it as an integer, hence:

```
RD $J FILE=950101.001
```

reads image 950101.001 into image buffer given by integer variable J, whereas

```
FILTER $F
```

sets the filter to the position given by the value in integer variable F.

If a floating-point number is required (e.g., a decimal exposure time), then a % (percent sign) is prepended to the variable name to qualify it as a float, hence:

```
EXPTIME %T MINUTES
```

sets the exposure time to T minutes, where T is a floating-point variable. Similarly

```
GRTILT %ANG
```

tilts the grating by ANG degrees, where ANG is a floating-point number.

Note that you can also use $ or % to qualify any valid arithmetic expression as an integer or floating argument, hence

```
RD $J+1 FILE=950101.001
```

or

```
EXPTIME %(T+2*DT) MINUTES
```

are both valid commands, provided the variables being used have all been previously defined. The expressions are evaluated before being qualified and substituted onto the command line.

NOTES:

1. $ and % may not be used in "KEYWORD=VALUE" tokens. For example,

   ```
   TV 2 Z=%F L=5.0
   ```

   is invalid and will result in an error. The correct syntax is

   ```
   TV 2 Z=F L=5.0
   ```

2. $ and % may not be used *inside* numerical expressions, thus

   ```
   RD ($I+1) FILE=960101.023
   ```

   is invalid. By enclosing the $I in ()'s you are attempting to evaluate $I, which is unrecognized by the parser.

3. $ and % may not be used when passing variables to subroutine scripts via the CALL/PARAMETER mechanism, thus

   ```
   CALL DOEXPOSE $n %t
   ```

   is invalid, and will result in an error. The correct calling syntax is

   ```
   CALL DOEXPOSE n t
   ```

   With the variables n and t passed "unqualified".

If the integer part of a variable is required in an expression, the arithmetic functions INT[] and IFIX[] are provided to return the nearest integer and integer part (truncation), respectively. All XVista variables are "floats" by definition.

8.7 PRINTF: formatted output of strings & arithmetic expressions

Usage: PRINTF *'Format string'* [*expressions*] [*redirection*]

This command displays character strings and variables in specified formats, thus producing tables of results.

The simplest form of PRINTF is

```
PRINTF 'string'
```

which prints the string enclosed within single quotes.

Examples:

```
PRINTF HELLO
```

prints `HELLO` on the terminal screen, and

```
PRINTF 'Hello, world'
```

Prints the string `Hello, world` on the terminal screen.

You can print the values of arithmetic expressions by using a format specifier in the character string, followed by the names of the expressions to be evaluated and printed. The % character is used to denote a format specifier within a string. The format specifiers use a C-like syntax with certain affinities to Fortran (*Prospero* has a decidedly mixed heritage).

Examples:

Suppose we have two variables: A with value 1.0, and PI with value 3.14159. Then

```
PRINTF '%F4.1 %F9.4' A PI
```

prints

```
  1.0    3.1416
```

while

```
PRINTF '%I6 and %F9.5' A PI
```

prints

```
     1 and 3.14159
```

and finally,

```
PRINTF 'The value of pi is %F9.7' PI
```

prints

```
The value of pi is 3.1415900
```

Note that all spaces between the % format specifiers are printed.

The output of PRINTF can be redirected. For example:

```
A=45
PRINTF 'The sine of %I2 degrees is %F9.7' A SIND[A]
```

To force newlines in the printing, use the pattern "\N" or "\n" in the format statement. An example is:

```
A=5.1234
PRINTF 'The value of A is \n %F9.3' A
```

which prints

```
The value of A is
5.123
```

See Section 9.3 for how to print string variables.

Chapter 9: String Variables in *Prospero*

9.1 STRING: define a string variable

Usage:

STRING *name* [*'format string'*] [*expressions*]

STRING *name '?query'*

STRING defines string variables. String variables are names to which a character string is associated.

STRING works like PRINTF (§8.7), except that the name of the string variable being defined appears between the command STRING and the format string. As in PRINTF, you can define strings directly, as in:

```
STRING EXPR 'This is a string with seven words.'
STRING HELLO 'Hello, world'
```

Remember that multiple words to be considered as one string must be enclosed in quotes. If you wish to define a blank string, you must define it as a single blank in quotes, as in:

```
STRING NAME ' '
```

You may also define strings using expression evaluation, for example:

```
J=7
STRING FNAME 'FILE%I3.3' J
```

This evaluates the numeric variable J, and substitutes its value into a string named "FNAME" that begins with the characters FILE. In this example, the value of the string variable FNAME will become "FILE007".

If the first character of the format statement is a question mark (?), STRING will use the remainder of the statement as an input prompt, and wait for user entry at the keyboard. This allows the interactive entry of strings. If the format string contains only a question mark, the default prompt will be `Enter` followed by the name of the string variable to be defined.

Examples:

```
STRING FILE '?Enter a filename for this image. >> '
```

will print "`Enter a filename for this image. >>`", and then pause while you enter a name. Your reply will be loaded into the character string FILE.

```
STRING HEADER ?
```

will print "`Enter HEADER`" then accept a string.

Note: You *cannot* evaluate arithmetic expressions when defining strings interactively. Thus

```
STRING FILE '?Enter a file for image %I2' J
```

will not work. The "`\n`" newline character is also not allowed inside a prompt.

You can list all currently defined strings with the command PRINT STRINGS.

A string variable can be substituted into a command line by enclosing the name of the string in braces ({}'s). See below for more information.

String variables are stored in a different stack from numeric variables. Thus it is possible to have a string variable and a numeric variable with the same name.

9.2 Substituting String Variables into a Command Line

To substitute the value of a string into a command line, enclose the name of the string in braces. As an example, the command RD reads a file from disk. Its syntax is

```
RD buf filename
```

which reads a disk file "filename" into buffer number "buf".

Suppose that the string `FNAME` has been defined to be "`./myfile.dat`", then

```
RD 2 {FNAME}
```

will execute the command

```
RD 2 ./myfile.dat
```

If the string to be substituted has more than one word separated by spaces, then the substitution command must appear in single quotes, thus if the string `OBJNAME` has been defined to be "`NGC 1068 @ K`", then

```
OBJECT '{OBJNAME}'
```

will execute the command

```
OBJECT 'NGC 1068 @ K'
```

Without the quotes around `{OBJNAME}`, it would have recognized only the first word in the string.

9.3 Printing string variables

There are three ways to view the values of string variables:

1. Use PRINT STRINGS to show all currently defined strings.
2. Use PRINTF and substitute the string as the format string.
3. Use PRINTF with the %A format specification. As in Fortran or C, the number of characters to print may given.

Examples:

```
PRINTF '{STRING}'
```

Will print the named string on the terminal.

If FNAME is a string previously defined to be "ngc1068k.fits", then

```
PRINTF 'Writing file %A' '{FNAME}'
```

will print

```
Writing file ngc1068k.fits
```

on the terminal screen. Note that the %A format statement suppresses leading spaces.

If instead, you were to type:

```
PRINTF 'Writing file %A20 ' '{FNAME}'
```

it would print:

```
Writing file        ngc1068k.fits
```

forcing PRINTF to format the string with 20 characters, in this case resulting in a number of leading spaces. The formatting would be useful if the output were being redirected to an output file to make a column-formatted table (for example).

9.4 Getting values out of the FITS headers

The substitution mechanism can be used to copy the value of a FITS header card into a string or a numerical variable. The syntax for this is:

```
{?BUFFER:CARDNAME}
```

where:

BUFFER is the image buffer number
CARDNAME is the name of a FITS header card.

The value of the FITS card is substituted into the command line where indicated by the {?BUFFER:CARDNAME} construction. If the buffer number is incorrect, there is nothing in the listed buffer, or if the named card does not exist, an error message is printed and a blank string is loaded. Leading blanks and comments are stripped off. Use the STRING command to load a string with a FITS character card. Use a direct assignment to load a numerical FITS card into a *Prospero* variable.

Examples:

```
STRING OBJ '{?23:OBJECT}'
```

Loads the OBJECT card in the FITS header of the image in buffer 23 into string {OBJ}.

```
FOC={?1:FOCUS}
```

Gets the value of the FOCUS card (a number) and loads the numerical value into the *Prospero* variable FOC.

9.5 Advanced Examples of String Substitution

This section gives examples of more advances use of string substitution as part of command procedure scripts.

In this procedure fragment, the observer is asked to type the filename of an image that is to read from the disk and processed using a subroutine. The processed image is then written out with the same filename.

```
STRING FILE '?Enter the file to process. >> '
RD 1 {FILE} # Read image
CALL PROCESS # Process it
WD 1 {FILE} # Write out
```

As the procedure is run, the prompt “Enter ...” appears on the screen. The reply is loaded into the string variable FILE. Suppose the reply was “/usr1/data1/hd183143.001”. Then the next command, which reads an image from the disk, uses the string substitution mechanism to insert the string FILE into the command. The actual command executed is

```
RD 1 /usr1/data1/hd183143.001
```

Similarly, the last command in the procedure is

```
WD 1 /usr1/data1/hd183143.001
```

The following loop defines the string NAME. The value of NAME is successively FILE001, FILE002, FILE003, FILE004, through FILE100

```
DO INDEX=1,100
   STRING NAME 'FILE%I3.3' INDEX
END_DO
```

The string substitution mechanism can be used to read text from an opened ASCII data file, allowing you (for example) to write a script that reads object names and exposure times from each line of a file. See the discussion for the OPEN and READ commands for more details on using ASCII files.

An oddity: You cannot have a % as the first character in a format statement in string substitutions. For example, you want to generate object names of the form "Object @ filter", using a common object name previously defined as objname. The obvious syntax:

```
STRING FULLNAME '%A at V' '{objname}
OBJECT '{FULLNAME}'
```

will fail, as the STRING command will cause a parser error when it tries to interpret %A as a floating variable substitution! The reasons are obscure. You can accomplish the same thing by including the string to be substituted in the format statement proper, hence:

```
STRING FULLNAME '{objname} at V'
OBJECT '{FULLNAME}'
```

will work. It looks kind of odd (think if it as string concatenation), but it does the job.

Chapter 10: External ASCII Files

Prospero provides a facility for reading and writing ASCII text files. Uses of ASCII files within procedures include lists of images for processing, procedure log files, data to help control a procedure, (e.g., coordinates of an image mosaic) etc.

IMPORTANT!!!

All external ASCII files must have names of the form "`basename.ext`", where "`ext`" is a file extension. While Unix allows filenames without extensions, *Prospero* does not.

10.1 OPEN: open an ASCII data file for reading

Usage: OPEN *logical_name filename*

where:

logical_name Is the logical name you assign to the file
filename Is the disk filename.

OPEN opens the specified ASCII file read-only. Such files must be normal sequential files as might be generated by the editor or various commands. If the file is successfully opened the LOGICAL_NAME is then assigned to the opened file and all further references to the file are made using the logical name. If an OPEN is done using a logical name that is already assigned to an opened file, this old file is closed and the new file is opened. When a file is first opened, the first line of the file is ready to be read. A detailed example of how to use OPEN and the logical file name is given in the description of the READ command.

A maximum of up to FIVE (5) files may be open at one time. It is a good practice to get in the habit of using CLOSE to close files opened by a procedure script, as files are *not* closed automatically upon script termination.

Example:

```
OPEN DATA ./mydatafile.dat
```

Opens the file for reading and assigns the logical name DATA to the file.

10.2 CLOSE: closing an opened ASCII data file

Usage: CLOSE *logical_name*

where:

logical_name is the logical name of a file previously opened file.

CLOSE allows you to close one of the ASCII text files that you have previously opened with the OPEN command. You should regularly pair OPEN and CLOSE in scripts to make sure you close files you are done with as only five files may be OPEN at one time.

Unlike normal programs, OPEN files are **not closed automatically** when the procedure finishes execution. Since only five ASCII files may be open at one time, if your procedure uses an OPEN command, you should *always* include a CLOSE command before the end of the procedure.

10.3 READ: read the next line of an ASCII data file

Usage: READ *logical_name*

where:

logical_name is the logical name of a file previously opened file.

The READ command causes the next line of the named file to be read. This line then becomes the 'current' line for the file and all subsequent references to the file in arithmetic expressions use the current line. Each READ causes a new line to be read in the order in which they appear in the file. However, it is possible to skip specified lines in the file using the SKIP command. The following example shows how to use the OPEN, and READ commands in conjunction with arithmetic expressions.

Suppose you have a file called PHOTOMETRY.DAT containing the following four lines of data:

```
B V
100.5 150.3
110.4 164.9
75.3 113.6
```

We could compute B-V using the following simple procedure.

```
OPEN PHOT PHOTOMETRY.DAT
SKIP PHOT 1
DO I=1,3
      READ PHOT
      BV=2.5*(LOG10[@PHOT.1]-LOG10[@PHOT.2])
      PRINTF '%I2 %F10.3' I BV
END_DO
END
```

The first line opens the file and gives it the name PHOT. Since the first line of the file does not contain numeric data and is just a header for the columns of data, we use the SKIP command to label line 1 as a line to be skipped. We then begin to read the real data. The first time line 4 is executed it reads line 1 of the file. It finds that we have marked line 1 as a line to skip so it then reads the next line.

Line 5 then makes two references to the file. The construction @PHOT.1 is interpreted in the following way: In the 'current' line of PHOT (the one we just got with the READ), extract the first word of the line and convert it into a numeric value. And, of course, @PHOT.2 refers to the second word on the current line. The 'word' indicator can be either a constant as shown or it can be a variable. So if B=1 and V=2 we could have said @PHOT.B and @PHOT.V. Expressions are not allowed: @PHOT.(B+1) is illegal and @PHOT.B+1 means "add 1 to the value of @PHOT.B".

10.4 SKIP: skip over lines in an ASCII data file

Usage: SKIP *logical_name line#* [*line1,line2*]...

where:

logical_name is the logical name of a previously opened file
line# one line in the file to skip.
line1,line2 is a range of lines in the file to skip.

SKIP builds a table of lines to be skipped for the named file. There is enough room in the table for 50 skip specifications. Each individual line skipped counts as one specification and each range of lines skipped counts as two specifications. Whenever a file is opened, its skip table is cleared. Rewinding a file (see the REWIND command) does not clear the skip table. If you just type SKIP LOGICAL_NAME without any lines to skip then the table of skipped lines for the named file is printed.

Lines that are skipped in a file can not be read with the READ command or by string substitution and are not used by the STAT command. In particular, note that the number of lines in the file as returned by the STAT command is the actual number of lines minus any skipped lines.

Examples:

```
SKIP PHOT 1
```

Marks line 1 of file PHOT to be skipped.

```
SKIP PHOT 100,120
```

Marks 100 to 120 (inclusive) to be skipped.

```
SKIP PHOT 1 100,120
```

Marks lines 1 and 100-120 to be skipped.

```
SKIP PHOT
```

with no arguments will print the current skip table for the file labeled `PHOT`.

See the example under the READ command (§10.3) for another use of the SKIP command.

10.5 REWIND: position an open file to the beginning of the file

Usage: REWIND *logical_name*

where:

logical_name is the logical name of a previously opened file.

The REWIND command repositions the named file back to the beginning of the file. The file must already be opened for reading using the OPEN command. Lines in the file which are marked for skipping using the SKIP command will continue to be skipped.

10.6 STAT: find the properties of a file

Usage: STAT *variable=function[expression]*

where:

variable is a *Prospero* math variable in which the value of the statistic is stored.
expression is an arithmetic expression that involves at least one reference to data in an open ASCII file.
function is one of the following:

MAX	Find the maximum value of the expression.
MIN	Find the minimum value of the expression.
FIRST	Finds the first value of the expression.
LAST	Find the last value of the expression.
COUNT	Counts the number of lines in the file. In this case 'expression' is really just a logical file name.
LOAD	Loads the arithmetic expression from each line in the input file into a specified buffer using STAT N=LOAD[buffer,expression]

The STAT command can be used to determine information about the data values in an ASCII file. For the MAX and MIN functions, the given expression is evaluated for each line in the file. For the FIRST function, the expression is evaluated for the first line in the file and for the LAST function the expression is evaluated for the last line in the file. The COUNT function merely counts the lines in the file. Remember that skipped lines (see the SKIP command) are never included in the calculations. These STAT functions are not the same as the normal *Prospero* math functions and can not be included in other mathematical expressions.

The LOAD function allows the observer to load data from an input ASCII file into a *Prospero* image buffer. Arithmetic operations may be performed on the input data before loading into the buffer. Simply specify the desired buffer and the arithmetic expression to load. The new buffer will automatically be created.

Examples:

```
STAT NOBJS=COUNT[TARGLIST]
```

Counts the number of lines in the file designated by the logical name TARGLIST (see OPEN), and assigns this value to the variable NOBJS. The file associated with the logical name TARGLIST must have been opened with the OPEN command. Skipped lines are not counted.

```
STAT MAXVAL=MAX[2.5*LOG10[@PHOT.2]
```

Evaluates the expression "2.5*LOG10[@PHOT.2]" for each line in the file PHOT and sets MAXVAL to have the maximum value. The file PHOT will be left repositioned to the beginning of the file after the STAT command completes.

```
STAT N=LOAD[1,@PHOT.1*@PHOT.2]
```

Loads the product of the values in the first and second columns of the input file PHOT into *Prospero* buffer number 1.

10.7 Implicit Reading: substituting a file line onto the command line

String substitutions, using the {STRING_NAME} construction, can also substitute lines from an opened file. To do so use the form {LOGICAL_NAME}, which does an implied READ of the named file and substitutes the entire line from the file into the command line. You can also substitute particular words using the form {LOGICAL_NAME.WORD_INDICATOR} where the word indicator is an expression giving the word number. These substitutions always do a new implicit READ for each substitution. The following example shows how to use these implied READ string substitutions.

In this procedure, the observer is asked to give a filename. The file contains a list of disk file names for images that are to be processed in some standard way. The processed image is written out with the same name. There is one disk file name per line.

```
STRING FILE '?Enter image name file. >> '
OPEN IMAGES {FILE}
STAT LINES=COUNT[IMAGES]
DO I=1,LINES
      STRING DISKIM {IMAGES}
      RD 1 {DISKIM}
      CALL PROCESS
      WD 1 {DISKIM}
END_DO
END
```

10.8 Writing to Files using Output Redirection

Many (but not all!) programs that produce large amounts of information may have their output redirected by the observer. The output from these programs normally goes to the terminal, but instead can be written to a file or to the line printer.

To redirect the output, you must use the open/write (“>”) construct or the open/append (“>>”) construct, both analogous to the same Unix constructs. These must appear at the END of a valid command.

They work like this:

`command >filename`

writes the output to the specified file, creating a new version of that file.

`command >>filename`

appends the output to the specified file. If that file does not exits, it is created.

Examples:

```
PRINTF '%I %I' NUM PHOT >first.lis
```

Prints the values of the variables NUM and PHOT into the new file “first.lis”, converting them to integers. The file will be located in your current default directory. Note that since we are working within the Unix operating system, filenames are case sensitive.

```
PRINTF 'FILE {IMFILE} has mean of %F' MNFILE >>redux.log
```

Inserts the string `IMFILE` into the line of text and evaluates the value of `MNFILE`, appending this text string into the ASCII file `redux.log` in the current working directory. If this file does not exist, it creates it first. Note the use of `{}`’s around `IMFILE` which specify that this string is to be *substituted* in the command as shown.

Chapter 11: Sample Procedure Scripts

The following are examples of simple *Prospero* scripts. These are meant to be illustrative, rather than to suggest the only way to do things. Within each group of examples, you will notice a steady progression from simple to more complex implementations.

11.1 An Image Sequence (Part I)

In this script, the observer wishes to take a sequence of images through each of three filters (numbered 1–3 in the instrument filter wheel). This script is bare bones, and must be edited if the observer wishes to change any of the image parameters.

The script below does the following:

1. Takes 5 images of 30-seconds each through a J-band filter in filter wheel slot 1.

2. Takes 3 images of 60-seconds each through an H-band filter in slot 2.

3. Takes 5 images of 40-seconds each through a K-band filter in slot 3.

The object title is changed each time to reflect the new filter, and the `MGO` command is used to take the multiple integrations.

```
filter 1
exptime 30
object 'UGC 12176 @ J'
mgo 5
filter 2
exptime 60
object 'UGC 12176 @ H'
mgo 3
filter 3
exptime 40
object 'UGC 12176 @ K'
mgo 5
end
```

The script above is simply a list of the commands the observer would have typed if they were doing the task by hand. This is the most common type of script written by observers.

11.2 An Image Sequence (Part II)

Suppose that the observer wants to take 3-filter sequences for all of the objects on their target list. In this case, editing the script above to change the object name and integration times for each object would be both tedious and dangerous as there are more opportunities to make mistakes. A more general script can be written by making use of the `PARAMETER` command

(§6.5) and the various numerical variable and string handling utilities of *Prospero* to pass arguments to be used. By then making the script an alias, it can be treated as a custom *Prospero* command that takes arguments on the command line to set the object name an integration times each object.

This script will be stored in the procedure directory as filename "`dojhk.pro`", and executed via an alias that uses the `CALL` command (§6.1). We also add some other features as noted below.

```
#
# dojhk - take a BVR image sequence
#
# Usage: call dojhk 'object name' Jexp Hexp Kexp
#
# where: 'object name' is the object title in quotes
#        Jexp, Hexp, & Kexp are the integration times for
#           each filter band, in seconds.
#
# Will take 5 J images of Jexp sec, 3 H images of Hexp,
# and 5 K images of Kexp, changing the name each time.
#
# 1998 July 8 [rwp/osu]
#
parameter string=objid Jexp Hexp Kexp
#
# J-band sequence: 5 images of Jexp seconds each
#
filter 1
   exptime %Jexp
   string newname '{objid} @ J'
   object '{newname}'
   printf 'Taking 5 J-band images of %f5.1 sec each' Jexp
   mgo 5
#
# H-band sequence: 3 images of Hexp seconds each
#
filter 2
   exptime %Hexp
   string newname '{objid} @ H'
   object '{newname}'
   printf 'Taking 3 H-band images of %f5.1 sec each' Hexp
   mgo 3
#
# K-band sequence: 5 images of Kexp seconds each
#
filter 3
   exptime %Kexp
   string newname '{objid} @ K'
   object '{newname}'
   printf 'Taking 5 K-band images of %f5.1 sec each' Kexp
   mgo 5
string hey 'All images of {objid} are done'
alert '{hey}' 1 0
end
```

To use this script, you would write it out into a procedure script file named `dojhk.pro` with the command

```
wp dojhk
```

and then define an alias, `dojhk` to use as a custom command:

```
alias dojhk 'call dojhk'
```

To execute this script, you would type the `dojhk` alias with the necessary arguments. For example, to take JHK images of Mrk 35 with integration times/image of 60, 90, and 45 seconds in J, H, and K respectively, you would type:

```
dojhk 'Mrk 35' 60 90 45
```

The result will be five 60-second J-band images, three 90-second H-band images, and five 45-second K-band images of Mrk 35, each labeled appropriately.

We've used a number of different features in this script to enhance its usefulness as a general "custom command":

1. The # mark for embedding comments. These are really useful for remembering later what your script actually does (including later in the same night...)

2. The `%` qualifier when using a variable to substitute for a floating-point numerical argument on the command line (used here with the `EXPTIME` command to set the integration time from values passed by the `PARAMETER` command). See §8.6 or the online help page for `NUMBERS` for details on substituting variables for numerical command-line arguments.

3. The `STRING` command and the string substitution utility (the `{}`'s) to change the object name using the `OBJECT` command by appending the filter band name onto a user-provided string (the objid string passed by the `PARAMETER` command).

4. The `PRINTF` command to print status messages as the script proceeds.

5. The `ALERT` command to ring the bell and print a "done" message at the end.

Scripts of this kind are very useful if doing very repetitious observing programs, for example imaging surveys of many objects.

11.3 Simple Camera Focus Script

Below is a simple focusing script that uses a loop to change the re-imaging camera focus, taking an image of a slit mask at each step. This example illustrates:

1. Use of DO-loops.

2. Use of the PAUSE command to temporarily interrupt a script to let the user check settings.

Scripts of this kind might be found in a suite of canned procedures associated with setups for a particular instrument.

```
#
# dofocus - do a focus sequence
#
# Usage: call dofocus itime sfoc efoc fstep
#
# where: itime = integration time in seconds
#        sfoc  = starting camera focus value
#        efoc  = ending focus value
#        fstep = focus step
#
parameter itime sfoc efoc fstep
exptime %itime
pause 'Check the setup, then hit C to continue'
do foc=sfoc,efoc,fstep
      camfocus $foc
      string focid 'Focus=%i4' foc
      object '{focid}'
      printf 'Doing camfocus %i4' foc
      go
end_do
printf 'Focus sequence done...'
end
```

When executed dofocus sets the integration time and then pauses to remind the observer to make sure the camera, slit, and filter selections are OK and gives them a chance to make the necessary settings before proceeding further. When everything is set, the observer types C followed by the Return key to resume execution of the script. On each pass around the DO-loop, the image "object" name is changed to include the focus value, and the current focus value is printed for the observer to keep track of the sequence. For example:

```
call dofocus 2 100 300 20
```

Will take a sequence of 2-second focus images with camera settings running from 100 to 300 in steps of 20 (11 images).

11.4 More Complex Camera Focus Script

The next script illustrates how to acquire a sequence of images using the various flow-control commands (DO/END_DO, IF/THEN, and GOTO). In this case, our goal is a general script for taking a series of focus images of a slit mask to determine the optimal re-imaging camera lens focus for a particular filter band. Scripts of this kind might be found in the toolkit of an instrument support astronomer.

```
#
# cfoc - take a set of camera focus frames
#
# 1999 Feb 12 [rwp/osu]
#
# Ask for the camera to focus
getcam:
ask 'Which camera (f/2.8=0, f/7=1) >>' camid
if (camid<0)&(camid>1)
```

```
   printf 'Must choose 0 or 1'
   goto getcam
end_if
camera $camid
if camid==1
   string cstr 'f/2.8 Camera @'
else
   string cstr 'f/7 Camera @'
end_if
# ask for the filter
getfilt:
ask 'Which filter (J=1, H=2, K=3) >>' filtid
if (filtid<1)&(filtid>3)
   printf 'Must choose 1, 2, or 3'
   goto getfilt
end_if
filter $filtid
if filtid==1
   string fstr 'J Focus'
else_if filtid==2
   string fstr 'H Focus'
else
   string fstr 'K Focus'
end_if
#
# get the starting focus, ending focus, step size, and exp time
#
ask 'Starting Focus Value >> ' sfoc
ask 'Ending Focus Value >> ' efoc
ask 'Focus Step Size >> ' fstep
ask 'Integration Time >> ' itime
exptime %itime
#
# get the filename to use
#
string ffile '?Starting Filename (e.g. kfoc.001) >> '
filename '{ffile}'
#
# loop over camera focus, taking an image of the pinhole at each
setting
#
do foc=sfoc,efoc,fstep
   camfocus $foc
   string focname '{cstr} {fstr} Focus=%i4' foc
   printf 'Doing CAMFOCUS=%i4' foc
   object '{focname}'
   go
end_do
printf 'Focus run completed...'
end
```

This is a bit more sophisticated example that shows how to make a script interactive. It prompts for both numerical and string variables to use in setting the behavior of the script.

Some things to note:

1. If a variable is to take the place of a floating-point number on a command line you have to use the `%VARNAME` construct (e.g., as in "exptime %itime" above), as described in §8.6.

2. If a variable is to be substituted for an integer command-line argument, it needs to be qualified using the `$` qualifier (e.g., as in "`camfocus $foc`" above). See §8.6

3. The `STRING` command can either define a string directly, or if the ? appears first inside the quotes, it will prompt for the string value. See the online help file for `STRING` for details.

4. No spaces must appear anywhere within a logical expression, for example in "`if (filtid<1)&(filtid>3)`" above. Spaces are used to separate arguments on a command line, and if used in expressions, would treat it as two arguments where only one is expected (the parser is good, but not *that* good).

5. A jump-point label must end in a colon "**:**", but when the label is used in a `GOTO` statement, the colon is omitted (as in "`getcam:`" and "`goto getcam`" above).

11.5 IR Image Mosaic Script (Part I)

This script is intended to take a 2×2 IR image mosaic of a region with extended emission (e.g., a galaxy or a nebula), and therefore includes sky chopping in the usual ABBA fashion. The array is assumed to subtend a field somewhat larger than 100x200 arcseconds on the sky. The resulting field of this mosaic is approximately 200×400 arcseconds. The telescope should be centered on the object at the start, and at the end the telescope will be sent back to the starting point.

NOTE: the script and the two that follow will only work with those telescopes that provide the ability to command pointing offsets remotely.

```
#
# mos - take a 2x2 IR mosaic with 512x1024 InSb Array
#
# Usage: call mos
# 1997 Jan 11 [plm/osu]
#
exptime 5
#
# chop to sky
#
east 500 twait=10
object 'Sky 1'
avego 10
west 500 twait=10
#
# go to first position
#
offset dec=100 ra=50 twait=10
object 'Position 1'
mavego 2 10
```

```
#
# chop to sky
#
east 500 twait=10
object 'Sky 2'
avego 10
west 500 twait=10
#
# go to second position
#
south 200 twait=10
object 'Position 2'
mavego 2 10
#
# chop to sky
#
east 500 twait=10
object 'Sky 3'
avego 10
west 500 twait=10
#
# go to third position
#
west 100 twait=10
object 'Position 3'
mavego 2 10
#
# chop to sky
#
east 500 twait=10
object 'Sky 4'
avego 10
west 500 twait=10
#
# go to fourth position
#
north 200 twait=10
object 'Position 4'
mavego 2 10
#
# chop to sky
#
east 500 twait=10
object 'Sky 5'
avego 10
west 500 twait=10
#
# go back to start
#
offset ra=-50 dec=-100 twait=10
#
printf 'All done!'
end
```

This is just the sequence of commands you would type if you were doing this by hand (pretty tedious). As the mosaic gets more complex, the script has to similarly grow in size. Note that

for each offset we use the `TWAIT=10` keyword to tell the system to wait 10 seconds after each move for the telescope to settle. Actual settling times depend on the telescope control system and the size of the offset. Note also that we have used both the specific `NORTH/SOUTH/EAST/WEST` directional offsetting commands, as well as the generic `offset` command to accomplish the offsets.

The next script shows a more sophisticated way to do the same thing that is more easily extensible.

11.6 IR Image Mosaic Script (Part II)

Here we do the same thing as above, only now we embed the repeated operations of sky chopping inside of a subroutine script, and add a dithering subroutine as well. Thus there are three files needed for this example:

1. mos2.pro: the main procedure script file.

2. sky.pro: the sky chopping script file, called by mos2

3. dith.pro: the dithering script file, called by mos2 and sky

In this example we create a 2×2 mosaic as above, but at each "object" and "sky" position we "dither" the images around a four-corner pattern to help avoid bad pixels on the array. (Note: if you are acquainted with standard IR imaging practices, this should all sound familiar. If not, finding an IR imager instrument manual will help clarify why we do this, we won't discuss IR imaging practices here, only give a script that implements those practices with *Prospero*).

The *mos2.pro* script file:

```
#
# mos2 - take a 2x2 mosaic with TIFKAM
#
# Advanced version of mos.pro. Calls the subroutines
# sky.pro and dith.pro (which must be in the procedure
# directory for this script to work).
#
# Produces a mosaic field approximately 200x400 arcseconds
# in size. Uses an ABBA sampling pattern.
#
# Usage: call mos2.pro objname exptime coadds
#
# 1997 Jan 11 [plm/osu]
#
parameter string=objname etime coad
snooze=10  # time to wait for telescope to settle
if etime==0
  ask 'Exposure Time: ' etime
end_if
if coad==0
  ask 'Number of Coadds: ' coad
end_if
```

```
do imos=1,5
   string FULLNAME '{objname} Sky %i2' imos
   object '{FULLNAME}'
   call sky etime coad
   printf 'Finished sky %i2' imos
   if imos==1
      offset ra=50 dec=100 twait=snooze
   else_if imos==2
      south 200 twait=snooze
   else_if imos==3
      west 100 twait=snooze
   else_if imos==4
      north 200 twait=snooze
   else_if imos==5
      goto done     # just want a final sky
   end_if
   string FULLNAME '{objname} Position %i2' imos
   object '{FULLNAME}'
   call dith etime coad 2     # dither with mgo 2
   printf 'Finished set %i2 of 5' imos
   done:
end_do
#
# recenter telescope at original position
#
offset ra=50 dec=-100 twait=snooze
#
alert 'Attention: 2x2 mosaic sequence completed' 2 1
end
```

The *sky.pro* script file:

```
#
# sky.pro -- chop to a sky positon and dithing using dith.pro,
#            returning the telescop to the original position.
#
# Usage: call sky exptime coadds
#
# 1997 Jan 11 [plm/osu]
#
parameter etime coad
chop=500
#
south $chop
call dith etime coad 1
north $chop
return
```

The *dith.pro* script file:

```
#
# dith.pro -- dithering around the four corners of a
#             square 'side' arcseconds on a side
#
# Usage: call dith exptime coadds mgos
#
# 1997 Jan 11 [plm/osu]
```

```
#
parameter etime coad mg
side=10    # size of the dither square in arcseconds
snooze=10  # seconds sleep after offsetting
#
exptime %etime
#
do jd=1,4
   if jd==2
      west $side twait=snooze
   else_if jd==3
      north $side twait=snooze
   else_if jd==4
      east $side twait=snooze
   end_if
   mavego $mg $coad
end_do
#
# return to start of dither pattern
#
south $side twait=snooze
#
return
```

Some things to note:

1. For these scripts to work, they must all be in the default procedure directory, with the filenames given. With the `CALL` statement, the `.pro` filename extension is implicit.

2. The subroutines sky and dith end with the `RETURN` command instead of `END`.

3. This example (and the one before it) used the `TWAIT=` keyword with the offsetting commands to insert a timed pause in the procedure. This is to allow the telescope to stop bouncing after the offset.

4. The `ALERT` command is used at the end to wake up the observer, as this script could take a while to execute.

5. If a subroutine and its calling script both use the *same* counter variable in a DO-loop (e.g., both use variable I as the DO-loop counter), then things get confused. Try to use *different* counter variables when doing subroutines (see above).

6. Variables passed as arguments of sub-scripts using the `CALL` statement are always passed "unqualified" (i.e., without the `%` or `$` qualifiers), whereas variables used to substitute for numerical arguments in normal commands (e.g., `EAST` or `WEST`) must be properly qualified (see §8.6 for details).

11.7 IR Image Mosaic Script (Part III)

Now we generalize one step further. In `mos2.pro` above, we used a large IF/ELSE block to build the mosaic using hardwired offsets that gave coordinates relative to the previous tile of

the mosaic. In this example, we now specify the coordinates of the center of each tile of the mosaic relative to a (0,0) position defined by the observer to be the origin of the final mosaic. The mosaic may now be any size, breaking out of the limitation of a 2×2 mosaic as in the examples above. We will also introduce two new procedures, sky2.pro and dith2.pro that take additional arguments to give us more freedom in choosing the chopping and dithering parameters.

We emphasize that this is not the only way (or the best way) to do this, but illustrates how external ASCII data files can be incorporated into a relatively sophisticated procedure script.

There are four files needed for this example:

1. mos3.pro: the main procedure script file (see below)

2. sky2.pro: the sky chopping script file (see below)

3. dith2.pro: the dithering script file (see below)

4. mosaic.dat: an ASCII data file with the mosaic pattern

Below we give the mos3.pro script and a sample mosaic.dat file that replicates the 2x2 mosaics in the previous examples. Note here that we use the more general `OFFSET` command instead of the subset of directional commands (e.g., `NORTH`, `EAST`, etc.) used in the previous examples.

The *mos3.pro* script file:

```
#
# mos3 - Take an image mosaic with TIFKAM
#
# Builds an image mosaic using offsets given in the
# an external ascii file.  This file should contain
# absolute offsets from the initial telescope position.
#
# MOS3 dithers at each object and sky position, and
# sky chops following an ABBA pattern.
#
# This script calls the subroutines sky2.pro and
# dith2.pro, which must be in the procedure directory.
#
# Usage: call mos3 'object name' offlist exptime coadds
#
# The file 'offlist' has a 2-column format:
#
#        dra1 ddec1
#        dra2 ddec2
#         ... ...
#        draN ddecN
#
# where: +dra is east, -dra is west in arcseconds
#       +ddec is north, -ddec is south in arcseconds
#
# 1997 Jan 14 [plm/osu]
```

```
#
parameter string=objname string=offlist etime coad
if etime==0
  ask 'Exposure Time: ' etime
end_if
if coad==0
  ask 'Number of Coadds: ' coad
end_if

# set the sky chopping angle to 500 arcseconds N-S

rachop=0
decchop=500

# open the file containing the absolute offsets

open offsets {offlist}
stat noff=count[offsets]

roff=0
doff=0
do i=1,noff+1
  string FULLNAME '{objname} Sky %i2' i
  object '{FULLNAME}'
  call sky2 etime coad rachop decchop

# set some position tracking variables

  if i==1
    ralast=0
    declast=0
  else
    ralast=rtmp      # last ra offset
    declast=dtmp     # last dec offset
  end_if
  if i==noff+1
    goto done
  end_if

  read offsets       # read coords of the next mosaic cell
  rtmp=@offsets.1    # ra offset in column 1
  dtmp=@offsets.2    # dec offset in column 2
  roff=rtmp-ralast   # size of the next ra offset
  doff=dtmp-declast  # size of the next dec offset

# do the offset, waiting 10 sec for the scope to settle

  offset ra=roff dec=doff twait=10
  string FULLNAME '{objname} Position %i2' i
  object '{FULLNAME}'
  call dith2 etime coad 2 side  # dither with mgo 2
  printf 'Position %i2 of %i2 completed' i noff
  done:
end_do

# restore the telescope to the original position
```

```
offset ra=-rtmp dec=-dtmp twait=10

# close file and wake up observer

close offsets
alert 'Attention: mosaic sequence completed' 2 1
end
```

The *sky2.pro* script file:

```
#
# sky2.pro - chop to a sky positionand dithing using dith.pro,
#            returning the telescope to the original position.
#
# Usage: call sky2 exptime coadds rachop decchop
#
# where:
#  rachop  = N-S chop in arcseconds (north positive)
#  decchop = E-W chop in arcseconds (east positive).
#
# 1997 Jan 14 [plm/osu]
#
parameter etime coad rac decc
#
roff=rac     # size of the N-S chop
doff=decc    # size of the E-W chop
offset ra=roff dec=doff twait=20
call dith2 etime coad 1
offset ra=-roff dec=-doff twait=20     # reverse the chop
return
```

The *dith2.pro* script file:

```
#
# dith2.pro - dither around the four corners of a square
#             'side' arcseconds on a side and performing a
#             certain number of coadds and mgos there.
#
# Usage: call dith exptime coadds mgos side
#
# 1997 Jan 14 [plm/osu]
#
parameter etime coad mg side
snooze=15  # seconds to sleep after offsetting
#
exptime %etime
#
oside=-side
do jd=1,4
   if jd==2
      offset ra=oside twait=snooze
   else_if jd==3
      offset dec=side twait=snooze
   else_if jd==4
      offset ra=side twait=snooze
   end_if
   mavego $mg $coad
```

```
end_do
#
# return to start of dither pattern
#
offset dec=oside twait=snooze
#
return
```

Notes:

1. The offsets file, mosaic.dat is hardwired into the `OPEN` command, and has a "./" prepended to specify that the file is in the current working directory.

2. The `OPEN` command is balanced by a `CLOSE` command near the end of the script.

3. The cumulative offset is tracked with the `ROFF` and `DOFF` variables to be able to restore the telescope position at the end.

A sample `mosaic.dat` file would be:

```
50  100
50 -100
-50 -100
-50  100
```

Two things to note:

1. The file must reside in the current working directory, not the procedure directory.

2. This particular example replicates the 2×2 mosaic pattern executed by the mos.pro and mos2.pro example scripts.

Note that the coordinates of each tile of the mosaic are given relative to a common center (coordinates [0,0]), unlike the previous two examples where the offset coordinates are given relative to the previous tile of the mosaic.

Because we count the number of lines in the mosaic.dat file, we can make a mosaic of any size and pattern with this script.

This is clearly not the only way, or even the best way to do this observing procedure. We include it here as a possible solution. If someone comes up with a good mosaic script that works well in actual practice, please send it to us and we'll include it in future editions of this guide.

Chapter 12: Command Summary

This section provides a quick summary of all procedure scripting commands. Please see the text or the online help files for a detailed description of their function and use.

12.1 Editing, Reading, Writing, and Executing Procedures

PEDIT	Edit the procedure buffer
RP filename	read a procedure file from disk
SHOW [output redirection]	print the contents of the procedure buffer
WP filename	Write the procedure buffer to disk
RUN [arg1] [arg2] ...	Run the procedure buffer
CALL filename [arg1] [arg2] …	Execute a procedure file. CALL is also used to execute procedure scripts as subroutines of larger scripts.
VERIFY [Y|N]	Trace procedure execution line-by-line
PARAMETER [var1] [var2] [STRING=str1]	Evaluate command-line arguments passed to a procedure. If a procedure uses command-line arguments, PARAMETER must appear in the **first** executable (non-comment) line of the procedure.
END	End a procedure and return to the command prompt END must appear in the very last line of all procedures.
RETURN	Return from a procedure called as a subroutine
#	Insert a comment line into a procedure

12.2 Numerical Variables and Arithmetic Expressions

VARIABLE=value

The value assigned may to a variable be a number or an arithmetic expression. Up to 15 variables may be define/evaluated on a single command line.

Using numerical variables as command-line arguments:

Integer Arguments:	COMMAND $variable
Real Argument:	COMMAND %variable
Keyword Argument:	COMMAND KEYWORD=variable

Arithmetic Operators & Expressions:

+	addition	`B+C`
-	subtraction	`X-Y`
*	multiplication	`X1*X2`
/	division	`A/B`
-	negative sign	`-X`
^	exponentiation	`A^0.5`
=	equate	`A=B`

NOTE: No spaces may appear anywhere in arithmetic expressions.

Arithmetic Functions:

INT[E]	nearest integer to the expression E
ABS[E]	absolute value of E
MOD[E,I]	E modulo I (remainder of E/I)
IFIX[E]	integer part of E (truncation)
MAX[E,F]	the larger of E or F
MIN[E,F]	the smaller of E of F
LOG10[E]	Log base 10 of E
LOGE[E]	Log base e ("natural log") of E
EXP[E]	e raised to the power E
SQRT[E]	square root of absolute value of E

Trigonometric Functions:

SIN[E]	sine of E (E in radians)
SIND[E]	sine of E (E in degrees)
COS[E]	cosine of E (E in radians)
COSD[E]	cosine of E (E in degrees)
ARCTAN[E]	arctan of E, returns radians
ARCTAND[E]	arctan of E, returns degrees

12.3 String Variables

STRING strname 'the string'	Define a string.
STRING strname 'format statement' expr	Create a string that includes numbers input from variables or expressions
COMMAND {strname} [rest of command]	Substitute a string onto the command line

12.4 Printing & Prompting for Input

TYPE expr1 [expr2] ... [expr15]	Unformatted printing
PRINTF 'Format string' [exprs]	Formatted printing
PRINTF 'Format string' [exprs] >filename	Send formatted output to an external ASCII file.
PRINTF 'Format string' [exprs] >>filename	Append formatted output to an existing ASCII file:
ASK ['prompt in quotes'] variable	Prompt the user for a numerical value from the keyboard
YORN ['prompt in quotes']	Ask the user a Yes/No question. YORN sets the variable YORN to 1 if Yes, 0 if No.
STRING strname '?prompt in quotes'	Prompt the user for a string

PRINTF and String Format Syntax:

%F	unformatted floating-point number
%F4.1	4 digit float with 1 digit of precision (e.g., 123.4)
%I	unformatted integer
%I6	6-digit integer, no leading zeros (e.g., 1, 123, 142212)
%I3.3	3-digit integer with leading zeros (e.g., 152, 001, 015)
%A	unformatted string
%A10	10-character string

12.5 Flow Control in Procedures

STOP ['A message']	Stop procedure execution and return to the command prompt
PAUSE 'pause message'	Pause the procedure and return to the command prompt
CONTINUE -or- C	Resume a paused procedure
SLEEP tsec ['optional sleep message']	Put a procedure to sleep for tsec seconds
WAIT ['optional wait message to print']	Suspend a procedure until the user hits the RETURN key
ALERT 'alert message' ntimes tdelay	Print an alert message to the screen. The message is repeated *ntimes* with a pause of *tdelay* seconds between alerts.
GOTO label	Jump to a labeled line in a procedure
label:	Label a line as a GOTO jump-point
ERROR command	Execute “command” on Error
ERROR GOTO label	GOTO “label:” on ERROR
EOF command	Execute “command” on End-of-File (EOF)
EOF GOTO label	GOTO “label:” on EOF

12.6 DO Loops:

```
DO var=from,to,step
   execute these commands
END_DO
```

NOTE: No spaces are allowed between the arguments in the DO command.

12.7 Conditional (IF) Branching:

Logical & Boolean Operators:

>	greater than	A>B, A>2.5
<	less than	A<B, A<100
==	equal to	A==B, A==10
~=	not equal to	A~=B, A~=10
<=	less than or =	A<=B, A<=5
>=	greater than or =	A>=B, A>=3
&	Boolean AND	(A>B)&(C==0)
|	Boolean OR	(C>=2)|(B<A)

NOTE: No spaces are allowed anywhere in logical expressions.

IF/END_IF: Simple logical test:

```
IF (logical test)
   Execute these lines if the test is true.
END_IF
```

IF/ELSE/END_IF: IF/ELSE logical test:

```
IF (logical test)
   Execute these lines if the test is true.
ELSE
   Execute these lines if the test is false.
END_IF
```

IF/ELSE logical test after a Yes/No question:

```
YORN 'Do something'
IF (YORN)
   Execute these lines if yes
ELSE
   Execute these lines if no
END_IF
```

Multi-level IF-Test Block:

```
IF (logical test 1)
   lines to be executed if test 1 is true.
ELSE_IF (logical test 2)
   lines to be executed when test 1 is
   false but test 2 is true.
     .
     .
     .
ELSE_IF (logical test N)
   lines to be executed when all conditions
   are false except test N.
ELSE
   lines to be executed if and only if
   all other conditions are false.
END_IF
```

12.8 External ASCII Data Files

Up to 5 ASCII files may be opened at one time. All files are readonly. User-assigned logical names are used to distinguish the currently open files.

OPEN logname filename	Open the ASCII file *filename* and assign it logical name *logname*
CLOSE logname	Close *logname*
READ logname	Read the next line from *logname*
STRING strname {logname}	Read then next line of *logname* a string
variable=@logname.n	Extract numbers from column n of the last line read
SKIP logname line [line1,line2]	Skip over selected lines in *logname*
REWIND logname	Rewind *logname* to the first line
STAT variable=COUNT[logname]	Count the number of lines in *logname*
STAT variable=MAX[@logname.n]	Find the maximum data value in column n
STAT variable=MIN[@logname.n]	Find the minimum data value in column n
STAT variable=FIRST[@logname.n]	Find the first data value in column n
STAT variable=LAST[@logname.n]	Find the last data value in column n

Chapter 13: Differences from *XVista*

In stripping *XVista* down to its parser to form a foundation for *Prospero*, we made a few, syntactic changes and additions to the *XVista* scripting language. Since we treat *Prospero* and *XVista* as separate (if related) programs, we felt justified in making deep changes that make sense in the context of Prospero's data acquisition function without regard to XVista tradition or convention if it best suited our needs. We note that about 90-odd% of Prospero's command syntax is backward compatible with *XVista* scripting conventions, but there are some important differences that might throw unwary *XVista* users.

The simple summary is: however much it may look like *XVista*, *Prospero* is NOT XVista by another name. Especially in three important areas:

13.1 No GO

The GO command in *Prospero* means "start an integration." This intuitive and deeply rooted in the syntax of nearly every data-taking system used in astronomy, and it was not considered worthwhile to make *Prospero* behave differently. Thus, to execute commands in the procedure buffer, *Prospero* uses the RUN command in place of the *XVista* GO command.

13.2 Comments

We adopt the # (hash) character as the comment character, This is a syntax common to Unix shell scripts, *IRAF* cl scripts, and is the comment character used by configuration files in the data taking system. First-time users with no experience of *XVista* would guess # as a comment more readily than the ! character used in *XVista*.

XVista users should note, however, that the ! character is also treated as an alternative comment character in *Prospero*, even if it is not documented as such.

13.3 Integer and Floating-Point variables as command-line arguments

Some *Prospero* commands require floating-point arguments alone on the command line (e.g., the EXPTIME command accepts fractional exposure times). We thus have introduced the %VARNAME floating-point qualifier syntax to *Prospero*, taking its place along side the $VARNAME integer qualifier syntax adopted from the original *XVista* parser. See §8.6 for details.

As in *XVista*, the $VARNAME qualifier converts the variable VARNAME to the *nearest* integer value (via the Fortran nint() intrinsic function) before substituting it on the command line.