IRIS Explorer User's Guide
Release 3

Document Number NP2872
ISBN 1−85206−110−3

Chapter 6

Scripting

This chapter describes how to use the scripting language, Skm, to take full advantage of the command interface in IRIS Explorer. It covers these topics:

- running IRIS Explorer with a script
- examples of its use
- the structure and syntax of Skm commands
- the Skm Editor window

6.1. Overview

IRIS Explorer provides two user interface mechanisms: the graphical user interface (GUI) which is described in the previous chapters, and a command interface. The command interface allows you to run IRIS Explorer using text−based commands. You can issue these commands directly from the keyboard, or you can write a script that IRIS Explorer runs, using the IRIS Explorer scripting language called Skm (pronounced "scheme").

The scripting language is useful for three reasons:

- You can set up a series of commands for testing new modules, maps, or applications.
- You can run IRIS Explorer automatically by piping information into it, which is handy for running batch jobs and test suites.
- You can run IRIS Explorer remotely from another system, such as a personal computer, and you can write the results to a file.

To create a script, you use the IRIS Explorer scripting commands. The commands are described in **"Command Syntax."**

6.1.1. Running IRIS Explorer by Script

To run a script when IRIS Explorer starts up, at the shell prompt (%), type:

```
explorer −script scriptfilename
```

A map file and a script file can be specified on the command line. For example:

```
explorer −map mymap −script scriptfile.skm
```

To set up IRIS Explorer for interactive scripting, at the shell prompt (%), type either:

```
explorer −script −
```

or

```
explorer −script %
```

Both of these puts you into the Skm interpreter, which displays this start up message and prompt:

```
IRIS Explorer Version 3.0

skm>
```

If you use the "explorer −script −" option this prompt appears in the shell window you invoked explorer from. If you use the "explorer −script %" option then this message and subsequent scheme output appears in a dedicated scheme window (see later).

At this prompt, you can type Skm commands as you go and have them executed in IRIS Explorer immediately.

You can pipe commands to IRIS Explorer using "explorer –script –" with a UNIX pipe. For example:

```
cat myFile.skm | explorer –script –
```

If you do use a pipe, although Skm output will appear as normal, subsequent commands cannot be entered from the keyboard as IRIS Explorer will still be receiving its input from the pipe. A pipe sends the output of one command into a second command. For more information, refer to the UNIX documentation.

6.1.2. Creating Simple Scripts

This section gives examples of how to use the Skm scripting commands. If you are entering commands interactively, you type each command on a separate line after the Skm prompt (**skm>**). If you are creating a script file, simply type the commands, each on a separate line.

The *start* command is used to start modules. For example,

```
skm>(start "ReadImg")
```

launches the *ReadImg* module in the Map Editor. The name of the module is enclosed in double quotes. The command:

```
skm>(start "SharpenImg" "at" 200 300)
```

launches the *SharpenImg* module in the Map Editor at the specified location. The coordinates specify the position of the top left–hand corner in pixels. Both the name of the module and the argument "at" are enclosed, separately, in double quotes.

Modules can be destroyed using the *destroy* command. For example,

```
skm>(destroy "ReadImg")
```

destroys the *ReadImg* module that was previously launched.

Maps are started by using the *start–map* command. For example,

```
skm>(start-map "cfd")
```

launches the *cfd* map.

The *connect* command specifies two ports to be connected. The command lists the output module name, the output port name, the input module name, and the input port name. For example,

```
skm>(connect "ReadImg" "Output" "SharpenImg" "Img In")
```

connects the "Output" port of *ReadImg* to the "Img In" port of *SharpenImg.*

Defining Command Variables

You can use the *define* command to define a variable for use in other commands. For example, you can use this command:

```
skm>(define mod1 (start "ReadImg"))
```

to start the module *ReadImg* and create a reference to the module in a variable called *mod1*. Then you can destroy *ReadImg* by giving the command:

```
skm>(destroy mod1)
```

You can substitute the variable *mod1* for `"ReadImg"` whenever you choose. This is important when you launch more than one copy of the same module (see **"Creating More Complex Scripts"**).

Similarly, you can save information about a *connect* command in the variable *c1*:

```
skm>(define c1 (connect "ReadImg" "Output" "SharpenImg" "Img In"))
```

This particular connection can be referenced later as `c1`. For example, you can break the connection using either of these commands:

```
skm>(disconnect "ReadImg" "Output" "SharpenImg" "Img In")
```

or

```
skm>(disconnect c1)
```

Variables containing module and connection information remain even after the IRIS Explorer objects to which they refer are gone, so you can reconnect the modules just disconnected with the command:

```
skm>(connect c1)
```

Example 6–1 An Example Script

Here is an example of a script that demonstrates the use of *define* in simplifing module connections. These modules do not currently exist, but you can use the script as a template and substitute your own module and port names in the appropriate places.

```
(define m1 (start "GenLat"))
(define m2 (start "testLattice"))
(define m3 (start "testLatticeFortran"))
(define m4 (start "testLatticePrint"))
(define m5 (start "ReadtstLattice"))
(define m6 (start "WritetstLattice"))
(set-param m6 "Filename" "foo.foo")
(define c1 (connect m1 "Output" m2 "Input"))
(define c2 (connect m2 "new" m6 "SuperLattice"))
(define c3 (connect m2 "new" m4 "SuperLattice"))
(define c4 (connect m1 "Output" m3 "Input"))
(define c5 (connect m3 "new" m6 "SuperLattice"))
(define c6 (connect m3 "new" m4 "SuperLattice"))
(define c7 (connect m5 "SuperLattice" m4 "SuperLattice"))
(define c8 (connect m6 "Filename" m5 "Filename"))
(disconnect c8)
(fire m1)
(fire m5)
```

Saving a Script

To use the script, save it with the *.skm* suffix, for example, *Testlat.skm*, then start IRIS Explorer with the command:

```
explorer -script Testlat.skm
```

6.1.3. Advanced Uses

Skm is a comprehensive language that provides the capability for creating much more sophisticated scripts than are described in this chapter. For more information on the Skm language syntax, read **Appendix B, "The Skm Language"**.

6.2. Using the Skm Commands

You can use the Skm commands to manipulate modules and maps in the Map Editor. Some of the commands let you emulate options available from the Map Editor and module pop–up menus. Using a Skm script, you can:

- launch a module and start maps and applications
- fire a module
- enable and disable selected modules or complete maps
- switch execution highlighting on and off
- destroy selected modules or complete maps
- make and break connections between modules
- edit a map, including cutting, pasting and duplicating selected modules
- manipulate module parameters
- save maps
- quit IRIS Explorer

Table B−1 in Appendix B lists, in alphabetical order, the Skm commands you can use to perform functions in IRIS Explorer.

The structure of these commands, plus any arguments they may take is listed in **Table B−2**, of Appendix B.

6.2.1. Command Syntax

In the Skm scripting language, all commands precede arguments and operands, and all expressions are enclosed in parentheses. If you want to say "do something," the Skm expression is:

```
(do something)
```

For example, to add two numbers, the addition operator precedes the arguments. Numeric constants do not require quotes:

```
skm> (+ 1 2)
3
```

White space separates operands and operators, as well as commands and arguments. You can use the space character, a tab, or a "newline" character. Non−numeric constants without spaces are preceded by a single quote, for example:

```
'all
```

Character sequences that include spaces should be enclosed by double quotes.

6.2.2. Interactive Help

To obtain details on the syntax of a command from within Skm itself, type **`(help)`** at the Skm prompt (**`skm>`**). You can also type **`(help−command "prefix")`** and you will get a list of all commands beginning with the given prefix.

Example 6−2 Using Skm Commands

You can use the commands in several different ways (see **Table 6−3**).

Command	Action
(copy "Contour")	Copies one module
(copy '("GenLat" "Contour"))	Copies a symbol
(copy (all−mods))	Copies the list of all modules

Table 6−3 Using Commands

6.3. Creating More Complex Scripts

This section describes the Skm command syntax in more detail to help you create flexible scripts for performing activities in IRIS Explorer.

6.3.1. Specifying Symbols

Symbols are identifiers that are preceded by a single quote. For example:

```
(maxi "modulename" 'at X Y 'size W H)
```

You can then define `var` as a symbol:

```
(define var 'size)
```

Setting Off Comments

The semicolon is the comment character in Skm. It comments out the remainder of the line on which it is located, for example:

```
; launch the Pick map and list the active modules
```

Using the Define Command

You assign values to a variable using the *define* command. For example, to define a variable called `myVar` and assign it the value `simple`, you write:

```
(define myVar simple)
```

To display the value of a variable, type the name of the variable at the Skm prompt without using parentheses, for example:

```
skm> myVar
simple
```

The assignment of a module to a variable is important because the actual name of the module depends on whether or not there are other copies of the module running.

If, for example, you use the command *start* to launch a copy of *ReadImg*, but there is already another active copy of *ReadImg* in the Map Editor, your module will be named *ReadImg<2>*. If you then use a *connect* command that explicitly refers to *ReadImg*, you will be referencing the wrong module.

The solution is to avoid referring to modules by the name used in the *start* command. Instead, assign the result of the *start* command to a variable and use the variable for subsequent references to the module.

For example,

```
(define mod1 (start "ReadImg"))
(connect mod1 "Output" "SharpenImg" "Img In")
```

6.3.2. Testing Variable Types

Skm provides the commands *module?* and *connection?* to test the type of a variable. They return the value `()` for false or `t` for true. For example:

```
skm> (module? m1)
t
```

means that the variable `m1` contains information about a module started some time in the past. It does not, however, necessarily mean that the module still exists.

Similarly, when *connection?* returns a value of `true`, it indicates that a particular connection once existed and information about that connection persists, but the connection itself may possibly have been broken.

Creating a Procedure

You can create a procedure in Skm with the *define* and *procedure* commands. This example demonstrates how to define a procedure that takes a single argument, declared formally as *a*, and adds 1 to it. The value returned by the procedure, if any, is the last value of the procedure.

```
(procedure (a) (+ a 1))
```

This procedure does not have a name yet (procedures can be "anonymous" in Skm). To give the procedure the name *plus1*, type:

```
(define plus1 (procedure (a) (+ a 1)))
```

You can invoke this procedure as follows:

```
skm> (plus1 3)
4
```

Using a Procedure

This example shows how to carry out the action of connecting image processing modules in IRIS Explorer. Image processing modules that operate on a single input typically have an input port called "Img In" and an output port named "Img Out."

First you define a procedure that will connect specified ports in two different modules. This procedure simplifies the connection of the modules by supplying the port names automatically:

```
(define image-conn
(procedure ( mod1 mod2 )
(connect mod1 "Img Out" mod2 "Img In")))
```

Then you invoke the procedure and provide the names of the modules to be connected:

```
(image-conn "SharpenImg" "DisplayImg")
```

6.4. Output from Skm

You can use the *print* command to print the output from a Skm command or script to the "IRIS Explorer console" (either the shell from which IRIS Explorer was launched or the Skm Editor Window, depending if IRIS Explorer was started with "−script −" or "−script %"). *print* takes one or more arguments; here are two examples of its use:

```
skm> (define v1 "abc")
skm> (define v2 123)
skm> (define m1 (start "ReadImg"))
skm> (print v1 v2)
abc123
```

and

```
skm> (print "My module is " m1)
My module is #< IRIS Explorer Module Ref: ReadImg >
```

6.5. The Skm Editor Window

If you started Explorer with the "explorer −script %" option, Explorer will open a dedicated window in which Skm interactions can take place, see **Figure 6−1**. The Skm Editor Window can be opened and closed by toggling the "Skm

Window" option from the Map Editor's Layout menu.

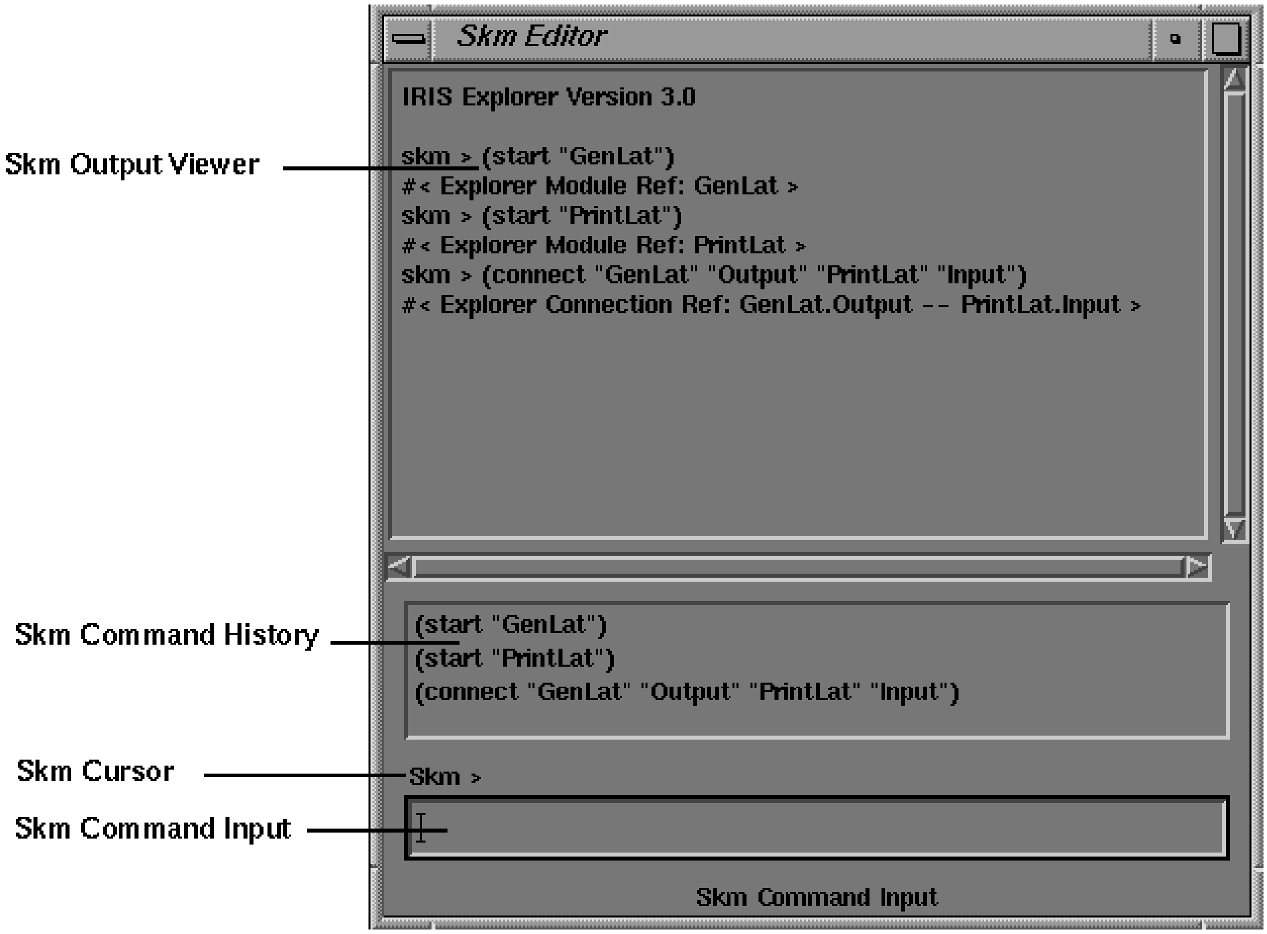

Figure 6–1 The Skm Editor Window

This main window is divided into four key areas described in the following subsections.

6.5.1. The Skm Output Viewer

All output from Skm is directed here, including the commands you type and the Skm cursor. One of the main advantages of using IRIS Explorer in this mode is that Skm output is separated from general IRIS Explorer standard output, for instance from modules. This makes it easier to keep track of what you have typed in Skm.

6.5.2. The Skm Command History Window

This window stores a list of commands previously entered in the Skm Command Input window. You can use the arrow keys to scroll up and down the list (you can also use the arrow keys, with the same effect, in the Skm Command Input window). When using the Command History Window, the currently selected item in the history list will be replicated in the Skm Command Input window. To re–run this command you can either "double click" on the return/enter key here or alternatively a single return/enter in the Skm Command Input window will execute the command.

6.5.3. The Skm Cursor

This tells you what state Skm is currently in, the possibilities are:

- **Skm>**
  This means Skm is ready to accept input. Type away!
- **Skm** (<integer>) **>**
  This means you have already typed something into Skm put have not finished entering the Skm command. A Skm command is complete when the number of left parenthesis match the number of right parenthesis. The integer in the bracket represents the number of unclosed right parenthesis.
- **Skm ...**
  This means Skm is busy. Further input will not be accepted until the interpreter is ready.

6.5.4. The Skm Command Input

This is where you type Skm commands. Like other text input areas you can "cut and paste" text out of and into the Skm Command Input. Press "return" or "enter" to pass the command input to the interpreter. Each command line will be stored in the Skm Command History window (see above). A single Skm command can be split over more than one command line.