5 Scripts

This chapter describes the syntax for SciAn scripts and explains every script command.

Syntax

A script consists of a sequence of text lines. Each line contains a single script command. If a line ends with a backslash (\), it is continued on the next line.

A line beginning with a hash sign (#) is a comment and is not parsed or interpreted. Comments are passed on to log files when SciAn is reading and writing a script.

Script lines in a command are parsed from left to right. The first token is always a keyword which specifies the command. Subsequent tokens are arguments which may either be keywords or values depending on the command. Tokens are normally separated by spaces. The elements in a list of objects are separated by commas.

Arguments which are not keywords are values. There are many types of values with syntax given by the table at the top of the next page.

Simple names which appear in a `window` command set the current script window to the named window. Elsewhere, simple names are considered names of objects in the current window.

Numbers are normally treated as real values. To use a number as an integer, use an integer type cast. For example, `(int) 2` is the integer 2, while `2` is the real number 2.0.

Type	Example	Description
Number	`-4.6e9`	Standard C syntax for real numbers with optional exponent. Generally used for values of numeric controls. There is no syntactic difference between a real number and an integer; an integer merely doesn't have a fractional component.
Simple Name	`Palette_1`	An identifier which specifies a name of a window or an object within a window. Use the backslash as an escape character to include characters other than letters and digits in a name. Use the underscore to represent spaces.
Window Name	`Window_7:Line_1`	A window name specifies an object within a particular window. It consists of the window, a colon, and a short name specifying the object within the window.
Database Name	`dataset@X`	A database name specifies an object of a certain type within SciAn's internal database. It consists of the type name, an at symbol (`@`), and a short name specifying the object.
Null object	`NULLOBJ`	The null object. Some controls, such as the color wheel, can take the null object as a value.
String	`"Hey, \"Bob\""`	A quoted string. Double quotes can be included using the backslash as an escape character.
Vector	`[0.0 1.0 2.0]`	A series of numbers within brackets. Used for some controls and position values.
Array	`[[1 2] [3 4]]`	A vector of vectors, or a vector of vectors of vectors, or as far as you need to go.

Script commands

This section gives all the script commands in alphabetical order. Portions of the commands which must be entered exactly appear `in this type face`. Portions which must be replaced with arguments *appear in italics*. In some cases, such as the `set value` command, an argument can be one of many types. In other cases, only one type is accepted. Portions of the syntax of the argument, such as the quotes around strings, may appear in the descriptions as a reminder of which type is accepted.

`activate` *object*

Activates *object*. Windows can only have one object of a certain class active at one time. Activating one object automatically deactivates all others. Currently, this is only used to activate video recorder drivers.

`annotation` *name left right bottom top*

Creates a new annotation with the given name and bounds and adds it to the space panel of the current window. This command has the same effect as drawing an annotation.

`begin recording` *time*
`...`
`end recording`

The `begin recording` and `end recording` commands enclose a block of commands to execute while recording to a video recorder. Replace *time* with a number of seconds larger than the duration of the recording. This number is used to find space on videodisc recorders before beginning recording, so it should be comfortably larger than the actual recording duration without being huge. Within this `begin/end` block, the `record` and `snap` commands will cause frames to be recorded.

`begin setup`
`...`
`end setup`

The `begin setup` and `end setup` commands enclose a block of commands which are used to set up a visualization. Within the block, only a minimal amount of redrawing will be done. The screen will look funny until the `end setup` command has been read. However, the commands will execute much more quickly than they would if all the redraws had to be done. This block should be added by hand to production scripts for movies, enclosing all but the recording block.

`begin snapshot` *object*
`...`
`end snapshot`

The `begin snapshot` and `end snapshot` commands enclose a block of commands which define a variable snapshot for *object*. Within the block, only `set variable` commands are valid. Snapshot blocks are produced when some objects with several coupled variables, such as observers, are logged. The snapshot block allows you to change any variable of any object, which makes it very powerful and very dangerous.

`bring to top` *name1*`,` *name2 ...*

Brings the named objects, which must be drawings on a space panel, to the top of the space panel.

`close`

Closes the current window. After the `close` command completes, the current window will be undefined. The command should immediately be followed by another `window` command.

`delete` *name1, name2 ...*

Deletes the named objects from the current window. This command has the same effect as the Delete item in the Object menu.

`deselect` *name*

Deselects the object named *name* without affecting the selection of any other objects. Selection commands do not occur in scripts unless the `-S` option is included on the command line.

`deselectall`

Deselects all the selected objects. Selection commands do not occur in scripts unless the `-S` option is included on the command line.

`drag` *name1, name2 ...*

Finds the named objects and prepares them to be dragged to another window. The objects which have been dragged are dropped into a window using the `drop` command.

`drop` *xPosition yPosition*

Drops all of the icons accumulated in the drag buffer into the current window. The first icon in the drag buffer is dropped at the location given by *xPosition* and *yPosition* in pixels, and the other icons are arranged around it according to their positions in the original window. The correct way to use `drag` and `drop` involves four steps: selecting the source window, doing a `drag` on each icon, selecting the destination window, and doing a `drop`. Because getting the positions right is tricky, it is a good idea to use the logging facility of SciAn to generate the `drag` and `drop` commands.

`duplicate` *name1, name2 ...*

Duplicates the named objects. This command has the same effect as the Duplicate item in the Object menu.

`edit palette` *name1, name2, ...*

Finds the named objects, which are presumed to be datasets, and opens windows to edit their color palettes.

`effect` *whichEffect*

If the selected window is a palette window, selects a color effect, where *whichEffect* is the number of the effect. This command is produced in logs but should never be written by hand.

`exit`

Exits SciAn. Normally, coming to the end of a script does not cause SciAn to quit. The `exit` command is useful at the end of a video script run remotely. It is identical to the `quit` command.

`eyeposn` [*x y z*]

If the current window is a visualization window, this command sets the position of the observer's eye to the specified coordinates. The `eyeposn` command is obsolete and has been replaced by the observer snapshot.

`fullscreen`

This command resizes the current window to fill the entire screen. The `fullscreen` command will never be logged from SciAn; the `locate` command will be used instead. The `fullscreen` command is there for convenience when writing a script by hand.

`hide frame`

Hides the frame around the current window. This only works if the windowing system has the capability of hiding the window frame.

`hide panel`

Hides the control panels of a visualization window. This command expects the current window to be a visualization window. See `show panel`.

`keep`

If the front panel is a palette window, keeps the changes to the palette.

`line` *name x1 y1 x2 y2*

Creates a line object named *name* from points *(x1 y1)* to *(x2 y2)* in the front space panel of the current window. This has the same effect as drawing a line using the line drawing tool.

`localcopy` *name1, name2 ...*

Makes local copies of the named objects. This has the same effect as choosing Make Local Copy from the Object menu.

`locate` *left right bottom top*

Locates the current window within the screen rectangle given by *left, right, bottom,* and *top* in pixels.

`model` *whichModel*

If the selected window is a palette window, selects a color model, where *whichModel* is the number of the color model. This command is produced in logs but should never be written by hand.

`modify` *name1, name2 ...*

Creates all possible modifications of the named datasets. This command has the same effect as the Modify button in the Datasets window.

`move to back panel` *name1, name2 ...*
`move to front panel` *name1, name2 ...*

Moves the named screen drawing objects to the back or front space panel. These commands have the same effect as the Move to Back Panel and Move to Front Panel items in the Arrange submenu of the Object menu.

`open` *name1, name2 ...*

Opens the named files and reads their contents into the Datasets window.

`pitch` *radians*

If the current window is a visualization window, this command sets the pitch of the observer's eye to the specified number of radians from horizontal. This command is only intended to be logged from SciAn; it should not be modified by hand.

`pushwindow`

Pushes the current window behind all other windows. This is sometimes useful instead of close when, in a recording script, you want to bring up a window for controls but push it behind all the other windows so that it cannot be seen.

`push to bottom` *name1, name2 ...*

Pushes the named objects, which must be drawings on a space panel, to the bottom of the space panel. Compare with the `bring` command.

`quit`

Quits SciAn. Normally, coming to the end of a script does not cause SciAn to quit. The `quit` command is useful at the end of a video script. It is identical to the `exit` command.

`record` *time*

Records *time* seconds of video on the video recorder. This command only works within a `begin recording` block. The number of frames recorded is given by the time multiplied by the number of frames per second (see `set fps`).

`rectangle` *name left right bottom top*

Creates a rectangle with the given name and bounds in the front space panel of the current window. This has the same effect as dragging out a rectangle using the rectangle drawing tool.

`revert`

If the selected window is a palette window, reverts to the last kept palette.

`roll` *radians*

If the current window is a visualization window, this command sets the roll of the observer's eye to the specified number of radians from upright. This command is only intended to be logged from SciAn; it should not be modified by hand.

`rotate x` *degrees*
`rotate y` *degrees*
`rotate z` *degrees*

Rotates the space in the current window *degrees* degrees about the specified axis, relative to the objects in the space. This command can used in video sequences in conjunction with the snap command to provide a sequence of views each rotated by a small amount.

`save controls`

If the current window is a control panel this saves the settings in the control panel to a file. This only works on palette control panels and other control panels containing a Save All Settings button.

`scrsave` *fileName left right bottom top*

Saves the screen section defined by the rectangle *left*, *right*, *bottom*, and *top* into file *fileName*. This command only works on the Silicon Graphics IRIS and behaves exactly as the `scrsave` command on that workstation.

`select` *name*

Selects the object named *name*. This command does not deselect any icons that are already selected but instead extends the selection to include *name*. Also see the `deselect` command. Selection commands do not occur in scripts unless the `-S` option is included on the command line.

`selectall`

Selects all the icons in the current window's main icon corral.

`set alignment` *textBoxName alignment*

Finds a text box in the current window with name *textBoxName* and sets its text alignment to *alignment*. Integer 1 means left alignment, 2 means center alignment, and 3 means right alignment. This is usually used for annotations and clock readouts in space panels.

`set bounds` *objectName* `[`*left right bottom top*`]`

Finds an object in the current window and sets its bounds to a rectangle given by *left*, *right*, *bottom*, and *top* in pixels. This is mostly used to move and resize annotations, legends, and rectangles in space panels.

`set color` *controlName colorNumber rComp gComp bComp*

Finds a color bar named *controlName* and sets color *colorNumber* to color components *rComp, gComp,* and *bComp*. This is a special purpose command which should, in general, not be written by hand.

`set endpoints` *name x1 y1 x2 y2*

Finds a line drawing object named *name* in the window and sets its endpoints to (*x1 y1*) and (*x2 y2*).

`set font` *textBoxName* `"`*fontName*`"`

Finds a text box in the current window with name *textBoxName* and sets its text font to *fontName*. This is usually used for annotations and clock readouts in space panels.

`set format` *name1, name2 ...*

Finds the named files and opens a Set Format dialog window to set their format.

`set fps` *framesPerSecond*

Sets the number of frames per second for subsequent `begin recording` blocks to *framesPerSecond*, which must be an integer. Because each recorder has a default number of frames per second, this command must occur after the `set recorder` command, if any. This command has been superseded by the recorder driver controls.

`set functionbox` *controlName* `[`*left right bottom top*`]`

Finds a color bar named *controlName* and sets its function box to *left*, *right*, *bottom*, *top*. This is a special purpose command which should, in general, not be written by hand.

`set location` *name* `[`*x y z*`]`

Finds an object with name *iconName* and sets the location of the object it represents to *x, y, z*. This is only currently used for icons that represent lights.

`set recorder` `"`*name*`"`

Sets the current video recorder to the recorder named by *name*. The quotation marks are required. This command may not occur within a recording block.

This command is obsolete and has been superseded by the `activate` command.

`set rotation` *speed* [*x y z*]

Sets the observer controlling the current window to rotate at *speed* around an axis given by *x, y, z*. The best way to get the axis is to log from SciAn. Select Rotation Inertia in the Preferences window and SciAn will automatically emit a `set rotation` command when you spin a space.

`set screen` *width height*

Sets the size of the recording screen to *width* by *height* pixels. This command has been superseded by the recorder controls.

`set size` *textBoxName fontSize*

Finds a text box in the current window with name *textBoxName* and sets its text size to *fontSize*, which should be an integral number of points. This is usually used for annotations and clock readouts in space panels.

`set value` *controlName value*

Sets the value of a control named *controlName* to *value*. It is important to make sure that *value* is a value that the specified control can accept. The most common types of values are real number and integer followed by string and vector. It is safest to use the logging facility of SciAn to produce scripts specifying values to set.

`set variable` *varName value*

Within a `snapshot` block, sets variable *varName* of the snapshot object to *value*. This command only works within a `snapshot` block. Be very careful with snapshots.

`shell` *command*

Executes *command* as if it had been entered within a shell. This may be useful in movie scripts to do machine-specific operations. *Command* may be any command that the system can execute.

Security *SciAn uses the* `system` *function to execute a shell command. There is a potential security problem posed by the* `system` *function if SciAn has been incorrectly installed to run under root no matter who started it. To avoid this problem, SciAn first tests to see if the user's ID is the same as the user ID of the process. If it isn't, SciAn will not run the command and will give an error message.*

`show back panel controls`

Shows the controls of the back space panel.

`show controls` *name1, name2 ...*

Shows the control windows for the named objects. This has the same effect as the Show Controls button or the Show Controls menu item.

`show datasets`

Shows the datasets window. This has the same effect as choosing Show Datasets from the Datasets menu.

`show filereaders`

Shows the window containing the file readers in SciAn. This has the same effect as choosing Show File Readers from the File menu.

`show fileswindow`

Brings up a dialog asking for a directory name which results in a new file system window. This has the same effect as choosing New File Browser from the File menu.

`show frame`

Shows the frame around the current window. This is useful only if the windowing system has the capability of hiding the frame in the first place.

`show front panel controls`

Shows the controls of the front space panel.

`show help`

Shows the help window. This has the same effect as choosing Help from the Windows menu.

`show info` *name1, name2 ...*

Shows the file information window for the named files. This is the same effect as the Show Info button or the Show Info menu item.

`show panel`

Shows the control panels of a visualization window. This command expects the current window to be a visualization window. See `hide panel`.

`show preferences`

Shows the Preferences window. This has the same effect as choosing Preferences from the Main menu.

`show recorders`

Shows the recorder drivers. This has the same effect as choosing Show Recorder Drivers from the Animation menu.

`show space controls`

Shows the controls for the space.

`snap`

Snaps a single frame on the video recorder. This command only works within a `recording` block.

`tile` *width height*

Tiles all of the visualization windows to a rectangle *width* by *height* at the lower left corner of the screen. This is handy to set up a group of visualization windows for a simple side-by-side comparison.

`time` *time string*

Specifies the time that this script (log) was written by SciAn. This line is ignored by the script reader and is only put in logs for your information.

`timereadout` *name left right bottom top*

Creates a time readout with the given name and bounds in the front space panel of the current window. This has the same effect as dragging out a time readout using the time readout drawing tool.

`turn on` *name1, name2 ...*
`turn off` *name1, name2 ...*

Turns on or off all of the named objects. These are usually used when the icons represent visualization objects or lights to turn on and off. These commands have the same effect as the Turn On and Turn Off items in the Object menu.

`turn show controls on`
`turn show controls off`

Turns on and off the “show controls” flag in SciAn, which determines whether control windows are shown on the screen. In normal operation, show controls is turned on. Control panels which are shown appear on the screen. When show controls is turned off, control panels are not shown on the screen but instead appear as hidden windows. All the script commands on objects within hidden windows work normally. This is useful in scripts for movies where you do not want to see the control panels.

`version` *version string*

Specifies the version of SciAn which wrote the script.

`videoscreen`

Resizes the current window so that it fills the video screen at the lower left hand corner of the screen. This location is just right for the NTSC mode of the Silicon Graphics IRIS workstations. The `videoscreen` command will never be logged from SciAn; the `locate` command will be used instead. The `videoscreen` command is there for convenience when writing a script by hand.

`visualize` *name1, name2 ...*

Visualizes the named datasets. This command has the same effect as the Visualize button in the Datasets window.

`visobjectas` *name1, name2 ...*

Creates templates for all possible visualizations of the named datasets. This command has the same effect as the Visualize As button in the Datasets window.

`window` *windowName*

Finds the window named *windowName* and selects it as the current window. The current window affects subsequent commands.

6 Recorder Drivers

<under construction>

7 Programming Concepts

This chapter gives an overview of the basic concepts needed to write additional code for SciAn. It is a prerequisite for any of the later chapters that describe how to write additional pieces of code.

Although we have tried very hard to keep SciAn from getting excessively cluttered, it is very much a product of evolution and has its share of history. For every rule stated in this section, you will no doubt be able to find exceptions.

Policy

SciAn is not in the public domain. Florida State University holds its copyright, because we are interested in having some control over what is released with our names on it.

If you are thinking of modifying the code, we'd appreciate it if you could send us a message (`scian-info@scri.fsu.edu`) describing what you plan to do. We're really quite easy to talk to. If you can't do that for some reason (such as if your project is shrouded in secrecy), you are welcome to modify the code as much as you like for internal use only, at your own risk.

Please don't distribute code you have modified. When we do releases, we have no way of knowing whatever assumptions you have made that might no longer be valid. We don't want to get into the position of giving an upgrade to a third party and having them get mad at us because the upgrade causes some of your code to break. In a code as complex as SciAn, it is often very difficult for a user to tell where in the code the error is.

Instead, please talk to us about having your code included in the next release of SciAn. You will get full credit, of course, and we will be the first layer of support. You can continue to keep the copyright on your code, as long as the conditions are the same as SciAn—anybody who wants to can download it for free. Please also let us know any disclaimers or legal mumbo jumbo that needs to be added.

Language requirements

SciAn is written entirely in ANSI-compliant C. It was originally written in pre-ANSI C and has been modified to compile without errors on the default settings of most ANSI compilers. There are still quite a few pre-ANSI remnants which remain in the code, but we are slowly eliminating them, and all new development should be fully ANSI, with the exception of method functions, as described later.

Only a few uniquely ANSI features are used. Function prototypes are used, as are `void` and `void *`. Structure passing and `enum` types are avoided to reduce the number of problems when porting. The `const` operator is only used when required by prototypes of library functions, such as `qsort`. Also as described later, SciAn uses object-oriented programming techniques using standard C functions and constructs. No object-oriented system such as C++ or Objective C is required.

The `#define` construct is used quite heavily to define macros to speed up critical pieces of the code. It is necessary that any preprocessor or compiler behave well with multi-line defines, continued using backslashes (\) at the end of each line. Include files are occasionally used as hyper macros, especially in `ScianPictures.c`. When these are used, a set of `#define` constructs before the file is included sets up macros to be expanded within the include file.

There are a few pieces of FORTRAN code which can be compiled and linked with SciAn. These are all written in FORTRAN 77, and need not be compiled into SciAn if a FORTRAN compiler is not available.

The SciAn source code

All the source code to SciAn is in a set of `.c, .h`, and `.f` files. All possible variations of SciAn are contained in the same set of source files. Which variation is being compiled depends on `#define` statements, most of which are in the `machine.h` include file.

When you download SciAn, most of the time you get the exact source code that we are currently working on. All differences are controlled by `#define` statements.

machine.h

The `machine.h` include file contains most of the compilation options. At the top is a group of `#define` statements which specify all the machine types on which SciAn can be compiled. If you want to try to convert SciAn to a different machine type, you will need to add another constant here.

Following is a group of `#if` and `#ifdef` statements which are designed to determine the proper machine type at compile time and set the `#define` constant `MACHINE` to the correct one. Unfortunately, this portion of the code is sensitive to the whims of compiler and operating system designers and may need to be changed for unusual systems.

Following is a group of individual `#define` statements which define various constants based on the selected machine. A list in the file explains each constant.

Following are some `#include` statements to include helper files produced by `ScianPreInstall.c`, described in the next section. These select options which can vary from machine to machine of the same type.

Finally is a `#define RELEASE` statement. This protects you from many of the ugly, half-finished bits in progress. We have left all these suppressed bits of code in SciAn on the off chance that you might need one of them. Please talk to us before messing with this flag, though. Once you have removed this you are alone in the swamp, and there is no shortage of alligators.

If you do a `#define COMPONLY` in `machine.h`, SciAn will be compiled as a computation-only node. When SciAn is compiled computation-only, it will run scripts just fine, but nothing will appear on the screen. Currently, this is only useful for remote file servers and remote computations. However, we hope to be able to produce a software renderer some day which will allow animations to be produced completely off line. The `COMPONLY` option is still experimental and should not be used by the faint of heart.

Making SciAn

Compilation is controlled by a single `Makefile`, which uses several auxiliary files to control compilation and linking options. `ScianPreInstall.c`, which is compiled and run when `make INSTALL` is done, analyzes the system on which it is running and produces a number of helper files included by `machine.h` and `Makefile`. The fact that some files must be included by `Makefile` is the reason that `make INSTALL` and `make scian` must be done as separate steps.

There are two files which `Makefile` uses to determine compilation and linking options. The file `flags.make` contains the compilation options, and the file `lfiles.make` contains the link flags and libraries. Both files are created by `ScianPreInstall.c`. This program identifies the machine type from the `#define` constants given in `machine.h`. Based on the machine type, it then selects two template files of the form `flags.`*machineType*`.make` and `lfiles.`*machineType*`.make`, where *machineType* is a short code giving the type of machine. To these files it adds additional information based on what it can guess about the machine and the questions it asks the user.

Because `lfiles.make` and `flags.make` are created automatically, they should not be edited by hand. An exception to this rule is temporary changes made for debugging purposes. Remember that the changes will go away the next time `make INSTALL` is done.

Source files

At last count, there were somewhere between 50 and 100 source files, most of them `.c` or `.h` files. In general, each source file contains a group of related functions and variables, although there is some overlap. For example, `ScianWindows.c` contains routines for generic windows, and `ScianObjWindows.c` contains routines for windows that hold objects.

In general, each `.c` file has a corresponding `.h` file, giving at least external prototypes and declarations for the functions, methods, and variables in the file. The file `Scian.h` needs to be included by every SciAn `.c` source file. `Scian.h` contains the most common declarations as well as all system include files needed by any source file. If you need to include additional system files, put them in `Scian.h`.

Note *You may have noticed a peculiarity in capitalization by now. The program is called SciAn, but the executable is scian, and each source file begins with Scian. The executable name is all in lower case because it's easier to type. This is for a reason which seemed a good idea at the time but now may safely be considered a historical accident.*

The easiest way to find out which `.h` files are required for a new `.c` file is to find an existing `.c` file which performs a similar set of functions and use its list of files, modifying it as necessary.

To add a new source file to SciAn, edit `Makefile`. Near the beginning, you will find a definition for a macro `COBJ` that gives a list of all the `.o` object files required by SciAn. Edit this definition so that an object file corresponding to your new source files are present. We have been trying to keep the names listed in alphabetical order, because it makes it easier to esimate the progress of a long make by looking at the names compiled so far.

Once `Makefile` has been editied, you need to produce the dependencies on your source code and put them in a file called `cdepend.make`. This file is included by `Makefile` and is interpreted as a group of dependency lines for all the source files. There is a utility called `mkmk.c` in the `util` directory to help you do this. Compile `mkmk.c` into an executable named `mkmk` and then enter

```
mkmk S*.c > cdepend.make
```

This will analyze the `#include` files in all the C source code and produce a group of dependencies, which will be used the next time any make is done.

Note *The `mkmk` program is not very intelligent. It only looks at the files included by each source file, and not at any files that are included by the include files. The dependencies also include the value of the macro `HFILES`, which is found at the beginning of `Makefile`.*

The design of SciAn

SciAn is run as a single executable process, not a group of modules communicating through shared memory. This has advantages and disadvantages. Advantages are compactness and speed as well as an independence from shared memory limitations. Disadvantages are the fact that the process can become blocked when waiting for a long operation to complete, and the fact that the entire source needs to be recompiled for each new addition. We are looking for ways around these problems, perhaps using a limited multiprocess approach which does not bring a smaller machine to its knees.

Internally, SciAn is designed around a single event loop, which handles window updates, user events, script reads, and deferred messages (explained later). Normally, a garbage collection of the complete object system is done every time through the event loop. Garbage collection is very fast and does not have a noticeable impact on the speed of SciAn. However, it does result in the fact that SciAn constantly uses CPU cycles, even when it appears to be idle.

Data types

A number of data types are defined in `SciAn.h` and `ScianTypes.h` which should be used when writing code for SciAn.

The `real` data type is the type used to hold floating point real numbers in SciAn. It is the same data type used by real objects and arrays. It is currently the same as `float`, but this may change. In the standard math library on many machines, floating point functions such as `sin` are designed to take `double` arguments and return `double` results. There are also functions, either beginning or ending with an `f`, which do the corresponding function for `float` arguments. The macros `rsin`, `rcos`, `rtan`, `ratan2`, `rsqrt`, and `rpow` map to the correct math library routine for the `real` data type.

The `Bool` data type is used for Boolean values within C code. A `Bool` can be one of the two constants `true`, which happens to be 1, or `false`, which happens to be 0. Within variables, a different system is used for true and false. False is represented by `NULLOBJ` or an integer 0, and true is represented by any nonzero integer. When setting variables of objects, use `ObjTrue` and `ObjFalse`. `IsTrue(`*object*`)` determines whether an object is considered true or false. It returns a `Bool`.

Memory allocation

Do not use the normal UNIX memory allocation mechanisms. Most of the memory allocation you will need to do will be done by the object system (explained later). If you really need to allocate a block of memory for your own purposes, use the SciAn functions `Alloc`, `Realloc`, and `Free`, which are replacements for `malloc`, `realloc`, and `free`, respectively. You can also use the `SAFEFREE` macro on a variable. It frees the memory only if the variable is non-zero and then sets the variable to zero.

The object system

SciAn uses an object-oriented programming system built around a functional interface in standard C. Actually, SciAn is a hybrid, and many sections of the program use conventional programming approaches. The object system is a combination of many ideas and was designed for utility and power within SciAn.

Many ideas within the object system, such as the use of pointers to typed data and garbage collection, should be familiar to users of any variety of the `LISP` programming language.

If you are not familiar with object-oriented programming or LISP, or you have had traumatic experiences with them, relax, don’t worry. We are not fetishists or purists for either LISP or object systems. We have simply used the concepts that gave us the power to build SciAn and left the rest. The interface is pure C.

An object is a structure which combines data and functions. Most of the data within objects is stored in variables. The functions within objects are stored as methods. They are activated by sending messages to the objects.

References to all objects are made through a pointer of type `ObjPtr` (defined in `ScianTypes.h`). For all practical purposes, no distinction need be made between an object pointer and the object itself.

An object may have one other object as its class. An object inherits variables and methods from its class, which may in turn inherit them from its class, and so on. In most cases, only one copy of a variable or method is inherited. In a few special cases (such as when sending the `DRAW` message) an object inherits all of the variables or methods for a certain ID from all of the classes of objects. This feature is exploited to allow visualization objects, for example, to inherit a variety of complex behavior for their classes.

Unlike many object-oriented systems, SciAn makes no structural distinction between the terms “object” and “class.” Any object can be the class of another object, and any class can be treated as an object in its own right. The distinction reflects nothing more than the role of the object in a particular operation.

Variables and methods are identified using ID’s. An ID is a unique integer of type `NameTyp` (defined in ScianTypes). The file `ScianIDs.h` contains `#define` statements for all the ID’s used by SciAn. By convention, the name of an ID is made up of all upper-case letters and numerals without underscores or other special characters. An object may have zero or one variables and zero or one methods with a single ID. When an object has both a variable and a method with the same ID, the method is usually a make method, as described later.

For those familiar with other object-oriented programming systems, the SciAn system is a single-inheritance, fully dynamic, variable-typed system with batch garbage collection.

SciAn has many types of objects. The type of an object, as distinct from the class, describes the extra information within an object beyond its variables and class.

Plain objects

The most basic type of object is the plain object. Although any type of object can have variables, methods, and a class, the vast majority of those which actually do are plain objects. Plain objects are created using a `NewObject(`*class*`, 0)` function call, where *class* is the superclass of the object. If this is a brand new class of object, use `NULLOBJ`, the null object, as the class. The 0 is the number of extra bytes at the end of the object to keep for extra information. For a plain object, this is always 0. You can test whether an `ObjPtr` is a plain object using the `IsObject(`*object*`)` macro. Like all of the other `Is-` macros, it returns data of type `Bool`, either `true` or `false`. (Type `Bool` is defined in `Scian.h`.)

Numeric objects

There are two scalar numeric objects: real objects and integer objects. Real objects are created using NewReal(value), where value is the real value to use and is either a real number (of type `real`, defined in `Scian.h`) or one of the three variables `plusInf`, `minusInf`, or `missingData`. The real value of an object can be extracted by using `GetReal(`*object*`)`, and an object can be tested to see if it is a real number using `IsReal(`*object*`)`. `NewInt`, `GetInt`, and `IsInt` perform the same functions for integer objects. There are no arithmetic functions for real and integer objects. The way to deal with them is to get their values and use C operators.

Character strings

String objects are used to store character strings. Create a string using `NewString(`*value*`)`, where *value* is a C string. Use `GetString(`*object*`)` to get the pointer to the string and `IsString(`*object*`)` to test it. The `ConcatStrings(`*object1*`,` object2`)` function is useful to concatenate two strings.

Arrays

Array objects are used to hold arrays of data. Create an array using `NewArray(`*type*`,` *rank*`,` *dims*`)`, where type is the type of the array, rank is the rank, or the number of dimensions of the array, and dims is a pointer to an array of longs giving the dimensions of the array. There are five types of arrays: real, object, byte, short, and pointer. Their types are `AT_REAL`, `AT_OBJECT`, `AT_BYTE`, `AT_SHORT`, and `AT_POINTER` respectively, as described in `ScianArrays.h`. Real arrays store real numbers and are used for most datasets. Byte arrays store single bytes and are used for compressed datasets. Object arrays store object pointers and are used for vector components and other purposes. Short and pointer arrays store short integers and pointers and are only used for very esoteric purposes.

Working with arrays is a bit more complex than working with integers or real numbers. There are `IsRealArray`, `IsObjArray`, `IsByteArray`, `IsShortArray`, and `IsPointerArray` macros to determine which kind of array a given object is. The `RANK(`*object*`)` macro returns the rank of the array, and the `DIMS(`*object*`)` macro returns a pointer to the dimensions. For example, `DIMS(array)[0]` is the first dimension of `array`. The elements of an array must be modified within C using the pointer to the elements returned by the `ELEMENTS(`*object*`)` macro. Array elements are stored tightly packed in row-major order, so the pointer returned by an `ELEMENTS` macro can be passed directly to a function which expects an array of the correct dimensions.

There are two functions which copy entire arrays. `CArray2Array(`*object*`,` *array*`)` copies all the elements within the C array *array* to array object *object*. `Array2CArray(`*array*`,` *object*`)` does the converse. These two functions are only useful for small real arrays. Use these two routines with care. In both cases, the size of the array is calculated from the dimensions of the array object. You must ensure that there are enough elements in the C array to hold all the elements. Also, because these routines can take a wide variety of arguments, it is not possible to write ANSI C prototypes for them. This removes compile-time protection against accidentally swapping the arguments. Just remember that the destination is the first argument, in the style of standard C functions such as `strcpy`.

The only correct time to change the elements within an array is immediately after creating it. If an array-valued variable needs to be changed later, it should be set to an entirely new array.

Geometry

Geometry objects are used to store geometry read from files. A geometry object only makes sense in reference to an unstructured dataset which gives locations of vertices. You should never need to deal with geometry objects directly, as all the functionality is provided by functions which act on datasets.

Pictures

Picture objects are used to store 3-D pictures to display within a space. Typically, a visualization object will produce a picture object for its `SURFACE` variable, which is automatically displayed by the `DRAW` method at the correct time.

Palettes

Palettes, which are created with `NewPalette()`, are mappings from numerical values onto colors. A palette is usually the `CPALETTE` variable of a dataset or a visualization object.

Symbols

Symbols are used to store the IDs of variables and methods in objects. For example, a control which controls the variable of an object may have a `REPOBJ` variable containing a pointer to the object it represents and a `WHICHVAR` variable containing a symbol that tells which variable to control. Create a symbol using `NewSymbol(`*variable*`)`, and get the variable ID from a symbol with `GetSymbolID(`*symbol*`)`.

Variables and methods

Variables and methods are objects and code which are associated with an object. Variables and methods are accessed through an ID, which is an integer of type `NameTyp`, defined in `ScianIDs.h`. An object may have zero or one variables and zero or one methods for every ID.

In the normal development of SciAn, whenever a new variable or message is needed, its ID is added to `ScianIDs.h`. There are, however, ten variables (`USER1` through `USER10`) which you can use for your own purposes. This is generally much easier than adding IDs to `ScianIDs.h`, which requires that every source file which includes it be recompiled. When you decide to make your code additions available in the general release, we will add the variables to `ScianIDs.h`.

Objects in SciAn are value typed rather than variable typed. That is, the type of an object is determined exclusively from its pointer and the data where it points, not from the variable where it was found. A variable can conceivably hold any type of data, although it may be an error to use the wrong kind of data for some variables. This is in contrast to languages such as C, which are variable typed.

Both variables and methods are set dynamically and participate in class inheritance.

Setting variables

There is one routine to set a variable of an object: `SetVar(`*object*`,` *variable*`,` *value*`)`, where *object* is the object that will hold the variable, *varible* is the ID of the variable, and *value* is the value. For example, to set the `LINEWIDTH` variable of an object called `contourObj` to integer 3, you would use

```
SetVar(contourObj, LINEWIDTH, NewInt(3));
```

`SetVar` will create a new variable of the given ID for the object and set its value to the value. If there was an old variable of that ID, its value will be lost. `SetVar` does not affect the values of any class to which the object belongs. `SetVar` returns the value to which the variable was set.

Getting variables

The first thing to do when getting a variable is to make sure it exists. Do this using the `MakeVar(`*object*`,` *variable*`)` function, where *object* is the object containing the variable, and *variable* is the ID of the variable to get. `MakeVar` is necessary for the dependencies mechanism to work (see “Variable dependencies” below). Some variables don’t participate in the dependencies system, and there are some cases (such as in make methods) where the step can be avoided. However, it is easiest and safest to do a `MakeVar` for all variables before getting them.

Once the variable has been made, the simplest way to get a variable is to use the `GetVar(`*object*`,` *variable*`)` call, where *object* is the object containing the variable and *variable* is the ID of the variable to get. If there is a variable of the given ID within the object, `GetVar` returns its value. If such a variable cannot be found, `GetVar` goes up the classes until it finds a value and then returns it. If it runs out of classes, it returns `NULLOBJ`, which is simply an `ObjPtr` of 0. This inheritance makes it easy to set the default value of a variable in a class, which can be overridden by a `SetVar` to an object of that class.

`GetVar` is easy to use, but it has one drawback: it will get the value of the variable, whatever type it happens to be. In order to write bulletproof code, it is a good idea to make sure the object is the right type before proceeding. One way to do this is to use protected GetVars, which are declared in `ScianErrors.h`. Protected GetVars are like `GetVar`, except that they have an expected type in the name of the function. They also take an additional argument, the first, which is a string giving the name of the function in which the protected GetVar is called. This string is used as the routine name for an error message which is emitted using `ReportError` if the variable is missing or of the wrong type. If any error occurs, a protected GetVar will return `NULLOBJ`, making error treatment very easy. For example, here is a code fragment to get the `LINEWIDTH` variable as set in the previous example:

```
ObjPtr var;
int lineWidth;

/*First make the var*/
MakeVar(contourObj, LINEWIDTH);

/*Now get it, protected as an integer*/
var = GetIntVar("ExampleFragment", contourObj, LINEWIDTH);
if (var)
```

```
{
        lineWidth = GetInt(var);

        /*Now do stuff that relies on lineWidth having a value*/
}
else
{
        /*Couldn't get the var, but the error message already went out, so don't
          do anything*/
}
```

`GetIntVar`, `GetRealVar`, `GetStringVar`, `GetPictureVar`, `GetPaletteVar`, and `GetSymbolVar` all do what you might expect from the names. `GetArrayVar` gets a variable which is guaranteed to be a real array or `NULLOBJ`. There are no protected GetVars for the other types of arrays. `GetFixedArrayVar(`*name*`,` *object*`,` *variable*`,` *rank*`,` *dim0*`,` *dim1*`,` *...*`)` gets a real array with the given rank and dimensions.

`GetPredicate(`*object*`,` *variable*`)` returns a `Bool` which is `true` if the given variable exists and is true and `false` otherwise.

Setting methods

Objects communicate by sending messages to each other. A message is named by an ID and is implemented within an object using a method. A method is actually a pointer to a piece of C code which implements the method.

Methods must be defined as C functions prior to any setting of the method, so that the function pointer is in the scope of the call that sets the method. The first argument of all methods is an `ObjPtr` argument which will contain the object to which the message is being sent. The remaining variables vary from method to method.

Because methods take a variable number of arguments, they may not be prototyped. Instead, they must be defined as traditional C functions.

All methods must return an `ObjPtr`. If there is nothing meaningful to return, the method should return `ObjTrue` if it succeeded and `ObjFalse` if it did not.

It is good practice to declare methods as static, as they should only be called through the message mechanism and never directly from C.

Once a method has been defined within C, it can be set as a method of an object using the `SetMethod(`*object*`,` *methodID*`,` *methodPtr*`)` function call, where *object* is the object, *methodID* is the ID of the method to set, and *methodPtr* is a pointer to the method.

As a general rule, it is best to set methods close to the root in the class hierarchy. Sometimes setting the methods of individual objects cannot be avoided, however.

The following example sets a method to get the name of an object to method ID `GETNAME`:

```
static ObjPtr GetObjectName(object)
```

```
ObjPtr object;
/*Gets the name of an object*/
{
	ObjPtr retVal;		/*The value to return*/

	MakeVar(object, NAME);
	retVal = Getvar(object, NAME);
	if (retVal)
	{
		if (IsString(retVal))
		{
			return retVal;
		}
		else
		{
			ReportError("GetObjectName", "NAME is not a string");
			return NULLOBJ;
		}
	}
	else
	{
		return NULLOBJ;
	}
}
...
	SetMethod(objectClass, GETNAME, GetObjectName);
```

Sending messages

Sending messages is a process with two steps. First you have to get the method of an object. Then you have to call this method on the object.

To make things a little easier, there is a type called `FuncTyp` which is equivalent to a method pointer. Use the function `GetMethod(`*object*`,` *methodID*`)` to get the method *methodID* of a object *object*. If there is a method in *object* or any of its classes, `GetMethod` will return a pointer to the method, or `(FuncTyp) 0` if one could not be found. Then use the returned method to call the method using C function pointer calling. Remember to make the first argument the object itself. Additional arguments can be included, but the caller and the method must agree on what they are.

The following example gets the name using the `GETNAME` method as described above:

```
ObjPtr name;
FuncTyp method;

	method = GetMethod(object, GETNAME);
	if (method)
	{
		name = (*method)(object);
		/*Do something with name*/
		...
	}
	else
	{
		/*No method, report an error or something*/
		...
	}
```

`ScianMethods.c` contains C wrappers for some commonly used methods.

`GetMethodSurely(`*name*`,` *object*`,` *methodID*`)` is a protected `GetMethod`, which generates an error if the method cannot be found. Most of the time, it's OK for an object to be missing a method, so `GetMethod` is the correct function to use.

Deferred messages

Often it is necessary to send a message to an object, but the object need not respond immediately. In some cases, such as showing the controls of the object in response to a button click, executing the message immediately has undesirable side effects. For cases like this, use `DeferMessage(`*object*`,` *methodID*`)`. The execution of the method will be put off until the next time through the event loop. Messages sent using `DeferMessage` cannot contain arguments.

Variable dependencies

SciAn contains a simple but powerful mechanism for linking variables with each other. It is based on the notion of dependency. Basically, a variable *v* depends on another variable *w* if an up-to-date value of *v* uses the value of *w*. When *w* changes, *v* must at some time be updated, and the SciAn dependencies system handles this.

Most of the updates within SciAn occur as a result of variable dependencies. For example, an isosurface visualization object knows that it needs to recalculate its surface when the isovalue changes as a result of dependencies. Dependencies can be any number of layers deep. For example, the surface of a mesh can depend on the dataset given by an ortho slicer filter, which can in turn depend on the value of the variables that determine which slice to use.

Dependencies link variables and methods together. Each variable which depends on another variable must have a make method, which makes the current variable of the variable based on all the variables it depends upon. Whenever an object has both a variable and a method with the same ID, the method is the make method for the variable.

To do dependencies, you must do two things. First you must declare all the dependencies of a particular variable. This is done with the `DeclareDependency(`*object*`,` *destVarID*`,` *sourceVarID*`)` function. Then, you must provide the variable with a make method. The make method is an ordinary method which takes one argument: the object. (Each variable has a different make method, so the variable ID is not passed.) The method has the job of setting the variable and returning `ObjTrue` if the variable changed or `ObjFalse` if it did not.

The following example declares an isosurface's `SURFACE` variable to depend on its `ISOVAL` variable and provides a make method for the `SURFACE` variable.

```
static ObjPtr MakeIsoSurface(visObject)
ObjPtr visObject;
/*Makes an isosurface's SURFACE variable*/
{
	ObjPtr picture;

	picture = NewPicture();

	/*Well, there's some very complicated stuff here to make the isosurface,
	  but that's not important.  At least, though, there has to be something
```

```
          like the following:*/
        MakeVar(visObject, ISOVAL);
        var = GetIntVar("MakeIsoSurface", visObject, ISOVAL);
        if (!var)
        {
                /*Can't do anything without the iso value*/
                return ObjFalse;
        }
        /*Now do something with ISOVAL and all the other stuff to put polygons
          into picture*/
        ...

        /*Set the picture*/
        SetVar(visObject, SURFACE, picture);
        return ObjTrue;
}

        /*Somewhere in the code, probably when isoClass is created*/
        DeclareDependency(isoClass, SURFACE, ISOVAL);
        SetMethod(isoClass, SURFACE, MakeIsoSurface);
```

The `DeclareIndirectDependency(`*object*`,` *destVar*`,` *indVar*`,` *sourceVar*`)` function is used to declare indirect dependencies. Assume *object* has a variable *indVar* which contains a pointer to another object, say *object2*. This declares *destVar* of *object* to depend on *sourceVar* of *object2*. Of course, for this to work then *destVar* must depend on *indVar* as well, and this dependency is declared automatically.

Dependencies only work when a variable is made using `MakeVar(`*object*`,` *variable*`)`. That is why it is important to make all variables before using them. When a `MakeVar` occurs on a variable, the entire tree of dependencies ending at the variable is traversed, and each variable which needs to be made is made.

Like variables and methods, dependency information is inherited. However, instead of just inheriting one dependency, all dependencies for a given variable are inherited (although only a single make method is inherited). It is possible to declare a variable dependent on as many other variables as you like.

Garbage collection

Garbage collection of all objects in the system is done frequently only during the event loop. This guarantees that garbage collection will not be done while one of your methods or functions is being executed.

When garbage collection occurs, all objects which are not anchored are deleted. Anchoring of objects is defined recursively. An object is anchored if one of the following conditions holds:

- The object is on the global reference list,
- The object is in the database,
- The object is a variable of an anchored object, or
- The object is marked by the `MARK` method of an anchored object.

Garbage collection proceeds by marking every object on the reference list, which recursively marks all the variables of each object and sends `MARK` messages to each object. Only a few objects actually have `MARK` methods, and the purpose is only to mark extra objects not in the plain object structure, such as the elements of object arrays. Then the garbage collector deletes all marked objects, first sending a `CLEANUP` message to the object. The `CLEANUP` method can dispose of extra memory not within the normal object structure.

Add objects to the global reference list with `AddToReferenceList(`*object*`)`, remove them with `RemoveFromReferenceList(`*object*`)`. The only time this is ever necessary is during `Init` and `Kill` functions at the beginning and end of the program to deal with object classes.

If you have set a variable of an existing anchored object to your new object, it will be anchored. Also, if you declare a new object to SciAn (such as with `RegisterDataset`), it will automatically be anchored.

You can create temporary SciAn objects within your routines. When you are done with them, just leave the pointer dangling. The storage will be collected the next time through the event loop.