The SCI Programming Language

(an appendix to the SCI32 documentation manual)

Author: Jeff Stephenson
4 April 1988

Revised by: Susan Frischer
1st Revision: 14 September 1994
2nd Revision: 28 September 1994

Table of Contents

Introduction to the SCI Language

The SCI language is an object-oriented language with a Lisp-like syntax. It is compiled by the sc compiler into p-machine code which is used by the interpreter, sci.exe. We will begin our discussion of the language with its basic Lisp-like characteristics, then go on to the object-oriented parts of the language. Like Lisp, SCI is based on parenthesized expressions which return values. An expression is of the form:

```
(procedure [parameter parameter ...])
```

The parameters to a procedure may themselves be expressions to be evaluated, and may be nested until you lose track of the parentheses. Unlike Lisp, the procedure itself may not be the result of an evaluation. An example of an expression is:

a ?

```
(+ (- y 2) (/ x 3))
```

which would be written in infix notation as:

(y - 2) + (x / 3)

All expressions are guaranteed to be evaluated from left to right. Thus,

```
(= x 4)
(= y (/ (+= x 4) (/= x 2)))
```

will result in y = 2 and x = 4.

Comments in SCI begin with a semicolon and continue to the end of the line.

~~integers or floats?~~

Primitive Procedures

Arithmetic primitives

In the following examples, e1, e2, etc. are arbitrary expressions. Brackets [...] indicate optional entries. Procedures evaluate their parameters from left to right.

Addition

```
(+ e1 e2 [e3...])
```

evaluates to: e1 + e2 [+ e3...]

example: `(+ 7 12 4)` evaluates to 23.

Multiplication

```
(* e1 e2 [e3...])
```

evaluates to: e1 * e2 [* e3...]

example: `(* 2 10 3)` evaluates to 60.

Subtraction

```
(- e1 e2)
```

evaluates to: e1 - e2

example: `(- 20 11)` evaluates to 9.

Division

```
(/ e1 e2)
```

evaluates to: e1 / e2

example: `(/ 24 6)` evaluates to 4. does this round when necessary (up, down)?

Remainder

```
(mod e1 e2)
```

evaluates to: the remainder of e1 when divided by e2.

example: `(mod 17 5)` evaluates to 2.

Operation Shift Left

`(<< e1 e2)`

evaluates to: e1 << e2 where the << operation shifts its left hand side left by the number of bits specified by its right hand side.

example: `(<< 7 2)` evaluates to 28.

In binary: 111 << 2 = 11100

Operation Shift Right

`(>> e1 e2)`

evaluates to: e1 >> e2 (as in << except with a right shift)

example: `(>> 7 2)` evaluates to 1.

In binary: 111 >> 2 = 001

Bitwise Exclusive OR Operator

`(^ e1 e2 [e3...])`

evaluates to: e1 ^ e2 [^ e3...]

example: `(^ 11 26)` evaluates to 17.

In binary: 01011 ^ 11010 = 10001

Bitwise AND Operator

`(& e1 e2 [e3...])`

evaluates to: e1 & e2 [& e3...]

example: `(& 11 26)` evaluates to 10.

In binary: 01011 & 11010 = 01010

Bitwise OR Operator

`(| e1 e2 [e3...])`

evaluates to: e1 | e2 [| e3...]

example: `(| 11 26)` evaluates to 27.

In binary: 01011 | 11010 = 11011

Bitwise NOT

`(~ e1)`

evaluates to: the bitwise not of e1 (all 1 bits are changed to 0 and all 0 bits are changed to 1).

example: `(~ 11)` evaluates to -12.

In binary: ~ 01011 = 1111111111110100 (all the leading 0s in the 16 bit number change to 1s).

Boolean primitives

These procedures evaluate their parameters from left to right and terminate the moment the truth value of the expression is determined. If the truth value of the Boolean expression is determined before an expression is reached, that expression is never evaluated. Brackets [...] indicate optional entries. The compiler predefines FALSE to be 0 and TRUE to be 1.

Greater Than

```
(> e1 e2 [e3 ...])
```

evaluates to: TRUE if e1 > e2 [> e3...], else FALSE.

example: `(> 7 4 6)` evaluates to FALSE.

Greater Than or Equals

```
(>= e1 e2 [e3...])
```

evaluates to: TRUE if e1 >= e2 [>= e3...], else FALSE.

example: `(>= 7 4 4)` evaluates to TRUE.

Less Than

```
(< e1 e2 [e3...])
```

evaluates to: TRUE if e1 < e2 [< e3...], else FALSE.

example: `(< 2 4 5)` evaluates to TRUE.

Less Than or Equals

```
(<= e1 e2 [e3...])
```

evaluates to: TRUE if e1 <= e2 [<= e3...], else FALSE.

example: `(<= 7 8 7)` evaluates to FALSE.

Is Equal To

```
(== e1 e2 [e3...])
```

evaluates to: TRUE if e1 == e2 [== e3...], else FALSE.

example: `(== 1 TRUE 1)` evaluates to TRUE.

Is Not Equal To

```
(!= e1 e2 [e3...])
```

evaluates to: TRUE if e1 != e2 [!= e3...], else FALSE.

example: `(!= 7 4 6)` evaluates to TRUE.

AND

```
(and e1 e2 [e3...])
```

evaluates to: TRUE if all the expressions are non-zero, else FALSE.

example: `(and 7 4 6)` evaluates to TRUE.

OR

```
(or e1 e2 [e3...])
```

evaluates to: TRUE if any of the expressions are non-zero, else FALSE.

example: `(or 3 0 2)` evaluates to TRUE.

NOT

```
(not e1)
```

evaluates to: TRUE if the expression is zero, else FALSE.

example: `(not 6)` evaluates to FALSE.

Assignment primitives

All assignment procedures store a value in a variable and return that value as the result of the assignment. In the following, v is a variable and e is an expression.

`(= v e)`	evaluates to	v = e
`(+= v e)`	evaluates to	v = v + e
`(-= v e)`	evaluates to	v = v - e
`(*= v e)`	evaluates to	v = v * e
`(/= v e)`	evaluates to	v = v / e
`(|= v e)`	evaluates to	v = v | e
`(&= v e)`	evaluates to	v = v & e
`(^= v e)`	evaluates to	v = v ^ e
`(>>= v e)`	evaluates to	v = v >> e
`(<<= v e)`	evaluates to	v = v << e
`(++ v)`	evaluates to	v = v + 1
`(-- v)`	evaluates to	v = v - 1

Data Types and Variables

Numbers

All numbers in SCI are 16 bit integers with a range of -32768 to +32767. Numbers may be written as decimal (1024), hex ($400), or binary (%10000000000).

Variables

Variables hold numbers. Variables can be either global, local, or temporary, depending on when they are created and destroyed. The maximum variable name length is 2047 characters. SCI variables are case sensitive (i.e.: MyVariable != myVARIABLE). A variable cannot begin with a number or a special character: # ( ) , . @ [ ] ` " { - nul ^I ^J ^M space.

Local variables are created when a script is loaded and destroyed when it is purged. They are only available while the script is in memory and will not retain a value through a purge-reload cycle. Local variables may also be assigned an optional value at the time of definition (as in the following example: firstVar is assigned the value of 4).

Local variables may be single variables:

```
(local
   firstVar        = 4
   secondVar
   thirdVar
      ...
)
```

or defined as arrays. When defining an array, you have the option of assigning a value to the first element of the array only (the succeeding elements must be assigned their values in additional statements). Note that the brackets surrounding the array definitions do not mean "optional." They are required. Also note that the first element of an array is designated as element 0, the second as element 1, etc. In the following, firstArray is defined to have 10 elements (0 - 9), the first element of which is assigned the value of 2:

```
(local
   [firstArray 10]       = 2
   [secondArray 5]
   [thirdArray 3]
      ...
)
```

Use the statement:

```
[myArray n]
```

to access element n of myArray. To access the fourth element of firstArray (the ten element array from the above example), write:

```
[firstArray 3]
```

Since each file may have only one local statement, that statement must include all the local variables used in that file. Therefore, the statement may contain both single and multiple (array) definitions:

```
(local
   [firstVar 5]
   secondVar
   thirdVar
   [fourthVar 3]
       ...
)
```

Despite the syntactic difference between single variable and local array declarations, SCI really makes no distinction between them. Think of the local declaration statement as a single array containing all the variables (including array elements) listed consecutively. Any variable may be accessed as an element of this "super-array," using any other variable as an index into the array. To clarify this concept, consider the following statement:

```
(local
   var1
   var2
   var3
   var4
)
```

Although these are all single variables, they are considered by SCI to be elements of a four element "super-array." Thus, the value of var1 can be set to that of var4 by any of the following statements:

```
(= var1 var4)
(= var1 [var2 2])
(= var1 [var3 1])
(= [var2 -1] [var1 3])
```

The first method is obviously the preferred method for clarity, but this array property of all variables allows access to variable numbers of parameters in a procedure (see section on procedures).

Global variables live for the duration of the entire game and are accessible to all scripts at all times. Thus they must be defined at the start of the game, either in room 0 or in a header file included by room 0. The definition of a global variable includes its name followed by a unique index number to be used by the table of global variables. An optional value assignment is also permitted. In this example, firstVar has the index number 0 and contains the value 7:

```
(global
   firstVar      0     = 7
   secondVar     1
   thirdVar      2     = 20
      ...
)
```

The syntax for a global array differs slightly from that of a local array, though the philosophy of consecutive elements remains the same. To declare a global variable, leave an array-sized gap in the numbering sequence. In the following, var2 is defined as a global array of 10 elements:

```
(global
   var1       23
   var2       24
   var3       34
)
```

To access the seventh element of var2, write:

```
[var2 6]
```

Temporary variables are created when a procedure or method is entered and destroyed when it is left. Therefore they are only available to the declaring procedure and do not retain a value between calls to that procedure. Temporary variables are defined using the symbol **&tmp**. The discussion on temporary variables will be continued in the section on user-defined procedures.

Text

Text strings are strings of characters enclosed in double quotes, and may be used anywhere you like. Note that the older notation enclosed text within curly braces; this notation may still be seen in some code modules.

```
(Prints "This is immediate text.")
```

prints the text string within the quotes.

```
(= textToPrint "This text is referenced through a
variable.")
```

sets the variable to the pointer to the text string. A text string may also be defined as the name property of an object (this concept will be clarified in the appendix on object-oriented programming).

When SCI goes to squirrel a text string away, it first checks to see if it has seen the string before. If so, it just uses the previous text, rather than duplicating it in another location. If you are using the same lengthy text string in several places, it is possible that you will not type the identical string in each case. Therefore, you can save yourself some trouble (and some memory) by putting the text into a **define** statement:

```
(define lotsOfText  "This is a long text string. I am using a
define statement to avoid having to type it repeatedly.")
```

This introduces another aspect of text strings. If text is too long fit on a single line, you may enter it on several lines. Multiple white-spaces (spaces, tabs, and new lines) get converted to a single space, so the text above ends up with just one space between the words on each line. If you want multiple spaces, enter them as underscores (_). These are converted to spaces in the string and are not compacted.

To include an underscore in the text, type **_** where \ (backslash) is the escape character. Explicit new lines (line feeds without carriage returns) are entered as **\n** (as they are in C). A CR/LF (carriage return/line feed) pair is entered as **\r**. Note that the **\r** should be used in place of **\n** in all strings destined for a file. Characters which are not on the keyboard, but are defined in a font can be included in a text string by preceding the character's two-digit hex value with the \ . For example:

```
(Prints "This is the Sierra symbol: \01")
```

would put the value 1 at the end of the string, and this character in the default font is the Sierra symbol.

The maximum length of a text string is 2047 bytes.

Characters

Characters are single ASCII characters, denoted by preceding the character with a reverse single quote (or tick). For example:

`` `A `` represents uppercase A
`` `? `` represents the question mark

Several character sequences represent special key combinations:

`` `^a `` represents ctrl-A
`` `@b `` represents alt-B
`` `#4 `` represents the F4 key

Literal selectors

Selectors represent methods or properties (and will be further defined in the appendix on object-oriented programming). Placing a pound sign (#) in front of the selector will return the numeric value for use as a parameter. In the following example, the value of the selector showSelf is passed as a parameter to the method eachElementDo:

```
(cast eachElementDo: #showSelf)
```

Literal selectors are commonly used by Collection objects to pass a selector to each of their elements in turn.

Definitions

define

The define statement allows you to define a symbol which will stand for a string of text:

```
(define symbol lots of text)
```

will replace symbol, wherever it is encountered as a token, with lots of text and then continue scanning at the beginning of the replacement text. Thus, if we write:

```
(define symbol some text)
(define some even more)
```

then:

```
(symbol)
```

will become:

```
(some text)
```

which then becomes:

```
(even more text)
```

enum

The enum statement eases the definition of the various states of a state-variable. Say you want to walk an actor from the door of a room across the floor, up the stairs, and through another door. You have a state-variable called actor-pos which will take on a number of values. These could be defined with defines as follows:

```
(local    actorPos
   define AT_FRONT_DOOR        0)
   define IN_ROOM              1)
   define ON_STAIRS            2)
   define TOP_OF_STAIRS        3)
   define UPPER_DOOR           4)
)
```

or you could get the same result with enum:

```
(local   actor-pos
   (enum
      AT_FRONT_DOOR
      IN_ROOM
      ON_STAIRS
      TOP_OF_STAIRS
      UPPER_DOOR
   )
)
```

Enum defaults its first symbol to 0. If you want a different starting value, put it right after the word enum:

```
(enum 7
   AT_FRONT_DOOR
   IN_ROOM
   ON_STAIRS
   TOP_OF_STAIRS
   UPPER_DOOR
)
```

sets `AT_FRONT_DOOR` to 7, `IN_ROOM` to 8, etc.

The value of an enum may also be defined by an expression, as follows:

```
(enum
   AT_FRONT_DOOR = (+ AT_REAR_DOOR 1)
)
```

Note: Define and enum statements may be included within both global and local variable definitions.

Control Flow

The value of a control flow expression is the value of the last expression in the control body which was evaluated. Thus, if we execute the following code:

```
(= x 3)
(= y 2)
(= y
   (if (> x y)
       (- x y)
   else
       (+ x y)
   )
)
```

y will have the value 1.

In the following, code1, code2, ... codeN are sequences of expressions. Brackets [...] indicate optional entries. The term "not FALSE" is used to indicate a non-zero result.

Conditionals

if

```
(if expression code1 [else code2])
```

If expression is not FALSE, execute code1, else execute code2.

example: Set the value of x to the larger of a or b.

```
(if (> a b)            ;expression
   (= x a)             ;code1
else
   (= x b)             ;code2
)
```

cond

```
(cond (exp1 code1) (exp2 code2) ... [(else codeN)])
```

Evaluate e1. If it is not FALSE, execute code1 and exit the cond clause. If it is FALSE, evaluate e2 and continue. If all of the expressions are FALSE and the optional else clause is present, execute codeN.

example: Set x to the larger of a or b and to 0 if a = b.

```
(cond
   ((== a b)                ;expression1
      (= x 0)               ;code1
   )
   ((> a b)                 ;expression2
      (= x a)               ;code2
   )
   (else
      (= x b)               ;codeN
   )
)
```

switch

```
(switch expression (exp1 code1) (exp2 code2) ... [(else
codeN)])
```

Evaluate expression. If it is equal to exp1, execute code1 and exit the switch. If it is equal to exp2, execute code2 and exit. If it doesn't equal any of the expressions and the optional else clause is present, execute codeN.

example: Evaluates a - b.

```
(switch (- a b)                                  ;expression
   (0                                            ;expression1
      (Prints "They are equal")                  ;code1
   )
   (-1                                           ;expression2
      (Prints "B is one unit larger than A")     ;code2
   )
   (1                                            ;expression3
      (Prints "A is one unit larger than B")     ;code3
   )
   (else
      (Prints "A and B differ by more than one");codeN
   )
)
```

switchto

```
(switchto expression (code1) (code2)... [(else codeN)])
```

Evaluate expression. If it is equal to 0, execute code1. If it is equal to 1, execute code2, and so on. **switchto** is a shorthand form of the **switch** statement where the test values are consecutive integers beginning with 0. It is commonly used in the changeState method of an SCI script object (where the expression is the state of the script).

example: Evaluates the variable stateNum.

```
(switchto stateNum                                         ;expression
   (
      (Prints "This is state 0")                           ;code1
   )
   (
      (Prints "This is state 1")                           ;code2
   )
   (else
      (Prints "The variable stateNum is not 0 or 1")   ;codeN
   )
)
```

Iteration

for

```
(for (initialization) condition (re-initialization) code)
```

Evaluate the expressions comprising initialization. Then evaluate condition. If the result is FALSE, exit the loop. Otherwise, execute code, then the expressions comprising re-initialization, and loop back to condition.

example: Set x = a + b while i < n. Note that this value does not change.

```
      (for
         ((= a 0)(= b 1)(= i 3))              ;initialization
         (< i n)                              ;condition
         ((++ i))                             ;re-initialization
            (= x (+ a b))                     ;code
      )
```

Note that the initialization statements may include more than one expression and therefore require a set of surrounding parentheses.

while

```
(while condition code)
```

Evaluate condition. If not FALSE, execute code and loop back to evaluate condition again. Exit the loop when condition is FALSE. (Note that this means that the resultant value of a while condition is always FALSE.) This is equivalent to:

```
      (for (0) condition (0) code).
```

example: Set x = x + the incremented value of i while i < n.

```
      (while
         (< i n)                     ;condition
            (+= x (++ i))            ;code
      )
```

repeat

```
(repeat code)
```

Continually execute the code until some condition in the code (a break) causes the loop to be exited. This is equivalent to:

`(while TRUE code)` or `(for (0) TRUE (0) code)`

example: Set x = x + 2. Note that this will loop forever.

```
(repeat
   (+= x 2)          ;code
)
```

Supporting constructs for iteration

break

```
(break [n])
```

Break out of n levels of loops. If n is not specified break out of the innermost loop.

example: Repeat incrementing i until i > n.

```
(repeat
   (++ i)
   (if (> i n) (break))
)
```

breakif

```
(breakif expression [n])
```

If expression is not FALSE, break out of n levels of loops. If n is not specified, break out of the innermost loop.

example: Repeat incrementing i until i > n.

```
(repeat
   (++ i)
   (breakif (> i n))
)
```

continue

```
(continue [n])
```

Loop back to the beginning of the nth level loop. If n is not specified, loop to the beginning of the innermost loop.

example: Set x = y / i unless i = 0.

```
(for ((i = -5)) (< i 5) ((++ i))
      (if (== i 0) (continue))
      (= x (/ y i))
)
```

contif

```
(contif expression [n])
```

If expression is not FALSE, loop back to the beginning of the nth level loop. If n is not specified, loop to the beginning of the innermost loop.

example: Same as "continue" except using the logical NOT to test i.

```
(for ((= i -5)) (< i 5) ((++ i))
      (contif (not i))
      (= x (/ y i))
)
```

return

```
(return [expression])
```

The return statement returns control to the procedure which called the currently executing procedure. If the optional expression is present, that value is returned as the value of the current procedure. There is an implicit return at the end of all procedures, and the value returned in that case is the value of the last expression evaluated. A return from the main procedure of script 0 returns to the operating system.

example:

```
(return
   (+ x k)          ;optional expression
)
```

Procedures

A procedure is like a user-defined function or subroutine. Procedures are created with the procedure construct. Note that it is allowable for the procedure to take no parameters and have no automatic variables (optional terms are shown in brackets). Procedure names are traditionally initial capitalized. "Code" in these examples represents any list of valid expressions:

```
(procedure (MyProc [p1 p2...] [&tmp t1 t2...])
   code
)
```

This defines the procedure "MyProc" with the parameters p1, p2, etc. and temporary variables t1, t2, etc. (designated by the precedent "&tmp"). These temporary variables disappear on exit from the procedure.

Temporary arrays are defined in the same way as local arrays. In the following example, "myArray" is the name of an array with n elements. Note that brackets are required when designating an array.

```
(procedure (MyProc &tmp [myArray n])
   code
)
```

Even though parameters are optional in a procedure construct, all procedures have one inherent parameter, the compiler-defined variable **argc** (argument count). The argument count contains the total number of parameters passed to the procedure. In the following procedure calls:

```
(MyProc 5 2 4)          ;argc = 3
(MyProc 1 3)            ;argc = 2
(MyProc 7)        ;argc = 1
```

In the following example, argc is used to fail-safe the SetPosition procedure in the event that fewer than two parameters are passed.

```
(procedure (SetPosition x y)
   (if argc                              ;if the argc is not zero, continue.
      (= theX x)
      (if (> argc 1)                     ;if the argc is > 1, continue.
          (= theY y)
      )
   )
)
```

Following are several examples of procedure constructs.

To square a number n:

```
(procedure (MySquare n)
   (return (* n n))
)
```

The following procedure "MyMax" finds the maximum number in a series of arbitrary numbers passed to it. The parameters for the variable p will be accessed as an array, biggest is the variable containing the maximum, and i is the index into the parameter array p.

```
(procedure (MyMax p &tmp biggest i)
   (for
      ((= i 0) (= biggest 0)) ;initialize i, biggest
      (< i argc)              ;compare to total number of parameters passed.
      ((++ i))                ;re-initialize
      (if (> [p i] biggest)
         (= biggest [p i])
      )
   )
   (return biggest)
)
```

Passing the following parameters to MyMax will return the value of 12:

```
(myMax 3 -4 -9 0 -2 7 12 4 3 5)
```

In order to use a procedure before it has been defined in a source file (for example, making a call to MyMax before the actual definition of MyMax), the compiler must be told that the procedure's name corresponds to a procedure, not an object. This is done with another form of the procedure statement:

```
(procedure
   ProcedureName1
   ProcedureName2
      ...
)
```

This tells the compiler to compile code for procedure calls when it encounters the procedure names, rather than code for send messages to an object.

Note that you may not use a parameter as a procedure variable if the caller did not pass it. In the following example, the caller passes the values a and b to the procedure. The value of c is not passed and will therefore be undefined. For example:

```
   ...
   ...
(Times 5 7)                ;passes two parameters to the Times procedure
   ...
   ...
(procedure (Times a b c)         ;procedure expects three parameters
   (= c (* a b))
   (return c)              ;c will be undefined.  The return will not be correct.
)
```

In the previous example, c should be designated as a temporary value (&tmp c) or left out altogether (return (* a b)).

&rest

&rest stands for all of the parameters not specified in the procedure or method. If a procedure or method has a variable number of arguments and wants to pass them on to another procedure or method, **&rest** tells the spin-off procedure to do all of them without knowing just how many parameters there are. The following example uses the procedure MySquare to square the maximum number in a series, as determined by the procedure MyMax:

```
(procedure (MySquare &tmp max)          ;no parameters passed to MySquare
  (= max (MyMax &rest))                 ;&rest passes parameters to MyMax
   (return (* max max))
)
```

&rest can also be used to specify parameters starting at any point in the parameter list of a procedure or method definition. Modifying the previous example, MySquare now defines the first two parameters passed to it as "first" and "second." To include them in the parameter list passed on to MyMax:

```
(procedure (MySquare first second &tmp max)
   (= max (MyMax (&rest first)))                 ;passes all parameters
                                                 ;beginning with "first"
   (return (* max max))
)
```

extern

Calling a procedure in another script is another matter. Since there is no link phase in the development cycle, one procedure cannot know the address of a procedure in a different script. The **extern** statement allows a script to know where the external procedure is:

```
(extern
   ProcedureName scriptNumber entryNumber
      ...
)
```

This says that the procedure referred to by the symbol "ProcedureName" in this script is to be found in script number "scriptNumber" at entry number "entryNumber" in the script's dispatch table. **kernel.sh** and **system.sh** both use the **extern** statement to let all other scripts know where their public procedures are.

The dispatch table for a script is defined by the public statement.

public

All procedures within a script which are to be accessed from outside the script must be entered in the dispatch table for the script with the public statement:

```
(public
   ProcedureName entryNumber
      ...
)
```

puts the procedure "ProcedureName" in the dispatch table at entry number "entryNumber." The entries need not be in numeric order, nor do the numbers need to be continuous (though if they are not continuous, the table will be larger than it needs to be).

Files

Source files for the script compiler traditionally have the extension **.sc**. Header (include) files traditionally have the extension **.sh**. Message editor header files traditionally have the extension **.shm**. Source files may have any filename; two examples are banner.sc and castle.sc. The two output files from the compilation will have the names **number.scr** and **number.hep** where number is the **script#** (defined below) of the module. Following are other files (besides the source file and any user-defined header files) which are involved in a compilation.

classdef

This file contains the information about the structure of the classes which have been defined in the application. It is read automatically by the compiler and is rewritten by the compiler after a successful compilation in order to keep it up to date.

selector

This file contains definitions of selectors which are used in object-oriented programming. It is automatically included in a compile and, like classdef, is rewritten after a successful compile. Any symbol in a properties or methods statement or in the selector position in a send to an object is assumed to be a selector and is assigned a selector number included in selector.

system.sh

This contains the definitions for interfacing with the various system classes and procedures. It also contains the system global variable definitions and defines for keycodes, script numbers, etc. It is automatically included in all compiles.

kernel.sh

This contains the definitions for interfacing with the kernel.

game.sh

This is the game-specific header file. It contains global variables, procedure declarations, and definitions for an individual game. It is automatically included in the compile after system.sh.

classes.txt

This is a text file produced by the compiler that is used by a BRIEF macro to browse the SCI class system.

offsets.txt

This is a text file that contains a list of class-selector pairs for frequently used properties. It is used by the compiler to produce 994.voc.

994.voc

For each class-selector pair in offsets.txt, the compiler writes an entry to this file containing the offset of that property in an SCI object. With this information, the interpreter can access a frequently used property without going through the message-passing mechanism. It is generated by the compiler when the -O command line parameter is used.

996.voc

This is the class table. It specifies the number of the script files that contain each class.

997.voc

This file contains the names of selectors. It is only used by the debugger and in printing error messages.

$$$sc.lck

This read-only file serves as a semaphore, indicating that a compile is occurring in the current directory. This prevents two compiles from trying to do the same task concurrently (updating the same classdef, for instance). $$$sc.lck is created at the beginning of a compile and deleted at the end.

The following two SCI commands deal with source code organization:

script#

The script# command sets the script number of the output file.

```
(script# 4)
```

sets the output file names to 4.scr and 4.hep, regardless of the actual name of the source file.

include

This includes a header file in the current source file at the current position.

`(include "/sc/foo.sh")` or `(include /sc/foo.sh)`

includes the file /sc/foo.sh. Include files may be nested as deeply as desired.

When compiling or including a file, the compiler first looks in the current directory. If it fails to find it there, it next looks for the file in the directories specified in the environment variable **sinclude**. This variable is just like the DOS path variable. The search directories are separated by semicolons. To set the compiler to look for include files in f:/games/sci/system and c:/include if it doesn't find them in the current directory, add the line:

```
set sinclude=f:/games/sci/system;c:/include
```

to your autoexec.bat file.

Compiling SCI Code

The SCI compiler is invoked with the command:

```
sc file1 [file2] [file3] [options]
```

Any number of file specifications may be entered on the command line, and a file specification may include wild-card names.

Options

-a	Abort compile if the file is already locked.
-d	Include debugging information so that the debugger can display source code.
-D<str>	Create a command line define which has the same result as using the define statement in a source file (except that spaces and some other characters are not permitted).
-g<num>	Define maximum number of global or local variables. The default is 750.
-l	Generate an assembly language code listing for the file with the original source interspersed. This is useful when using the built-in debugger of SCI. When compiling filename.sc, the list file is named filename.sl.
-n	Turns off "auto-naming" of objects. As described in the appendix on object-oriented programming, each object has a name property which is used to represent the object textually. Unless the property is explicitly set, the compiler will generate the value for this property automatically, using the object's symbol string for the name. The object names, however, take up space. While they are useful (almost vital) for debugging, if you're running out of memory in a room, it might help to compile with the -n option to leave the names out.
-O	Use offset.txt to generate 994.voc.
-oout-dir	Set the directory for the output files to out-dir.
-s	Display a message when a forward referenced selector is used.
-v	Do not lock the class database.
-w	Output words high-byte first (for the Macintosh).
-z	Turn off optimization. Not a particularly useful option except for those of us who must maintain the compiler.

Index

(scriptID 86 0)