UCRL-MA-108976 Rev. 1

SCORE User's Manual

Stewart A. Brown
Dennis Braddy

Lawrence Livermore National Laboratory

DISCLAIMER

Part of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Table of Contents

1.0 Introduction

PACT is a set of tools to facilitate the development of portable scientific software and to visualize scientific data. As with most constructs, PACT has a foundation. In this case that foundation is SCORE.

SCORE (**System CORE**) has two main functions. The first and perhaps most important is to smooth over the differences between different C implementations and define the parameters which drive most of the conditional compilations in the rest of PACT. Secondly, it contains several groups of functionality that are used extensively throughout PACT.

Although C is highly standardized now, that has not always been the case. Roughly speaking C compilers fall into three categories: ANSI standard; derivative of the Portable C Compiler (Kernighan and Ritchie); and the rest. PACT has been successfully ported to many ANSI and PCC systems. It has never been successfully ported to a system in the last category. The reason is mainly that the "standard" C library supplied with such implementations is so far from true ANSI or PCC standard that PACT would have to include its own version of the standard C library in order to work at all.

Even with standardized compilers life is not dead simple. The ANSI standard leaves several crucial points ambiguous as "implementation defined". Under these conditions one can find significant differences in going from one ANSI standard compiler to another.

SCORE's job is to include the requisite standard headers and ensure that certain key standard library functions exist and function correctly (there are bugs in the standard library functions supplied with some compilers) so that, to applications which include the SCORE header(s) and load with SCORE, all C implementations look the same. This is a tall order, but in practice once SCORE has been successfully compiled only the areas of graphics, IPC, and binary data handling require special handling! This has more of an impact on some programmers than on others. Those who prefer to specify only the exact headers to be included in each source file will find SCORE and PACT unusual. At the expense of a slight increase in compile time, the most commonly used headers are always included. This is crucial to getting the C implementation independence.

Typically, the SCORE header **scstd.h** includes the following:

ANSI	stdlib.h stddef.h stdarg.h float.h
PCC	sys/types.h varargs.h malloc.h
Both	limits.h stdio.h string.h math.h ctype.h signal.h setjmp.h time.h

The single header, **scstd.h**, smooths over most of the generic problems that arise because of implementation defined behavior in the host C implementation. The remainder of the PACT sources ultimately include **scstd.h**. This strategy has been extremely successful for PACT and applications which use PACT.

There are basically three other areas which SCORE functions address: memory management; hash table management; and extended string handling.

2.0 SCORE Constants

Hashing Constants

HSZSMALL = 31
- small table size

HSZSMINT = 67
- small intermediate table size

HSZLRINT = 127
- large intermediate table size

HSZLARGE = 521
- large table size

DOC = 1
- documentation

NODOC = 0
- no documentation

Lexical Scanning Constants

MAX_LEX_BUFFER = 4096
- default size of lexical token buffer

SC_WSPC_TOK = 0
- whitespace token

SC_DELIM_TOK = 1
- delimiter token

SC_IDENT_TOK = 2
- identifier token

SC_DINT_TOK = 3
- decimal integer number token

SC_REAL_TOK = 4
- decimal floating point number token

SC_OINT_TOK = 5
- octal integer number token

SC_HINT_TOK = 6
- hexidecimal integer number token

SC_OPER_TOK = 7
- operator token

SC_STRING_TOK = 8
- string token

SC_KEY_TOK = 9
- keyword token

SC_PRED_TOK = 10
- predicate token

SC_CMMNT_TOK = 11
- comment token

SC_HOLLERTOK = 1000
- hollerith string token

Miscellaneous Constants

REAL = double | float
- THINK C | otherwise

HUGE_REAL = 1.0e30 | 1.0e100
- THINK C | otherwise

ERRDEV = stderr
- device to dump diagnostic messages to

ABORT = 3
- error return flag for longjmps

ERR_FREE = 2
- error free return flag for longjmps

CODE = ““
- name of program using the package

VERSION = ““
- version designation of CODE

2.0.1 scstd.h

BINARY_MODE_R = “rb”

BINARY_MODE_W = “wb”

BINARY_MODE_RPLUS = “r+b”

BINARY_MODE_WPLUS = “w+b”

TRUE = 1

FALSE = 0

ON = 1

OFF = 0

FIXNUM = long
- if not already defined

SMALL = 1.0e-100

HUGE = 1.0e100

HUGE_INT = LONG_MAX | (1 << (sizeof(long)-1)) - 1
- if defined | otherwise

RAND_MAX = 32767.0 | 2147483647.0
- if not already defined

MAXLINE = 200
- if not already defined

SC_BITS_BYTE = 8
- bits per byte

SC_CHAR_I = ‘A’

SC_SHORT_I = ‘B’

SC_INTEGER_I = ‘C’

SC_LONG_I = ‘D’

SC_FLOAT_I = ‘E’

SC_DOUBLE_I = ‘F’

SC_STRING_I = ‘G’

SC_POINTER_I = ‘H’

SEEK_SET = 0
- fseek sets file position relative to beginning

SEEK_CUR = 1
- fseek sets file position relative to current position

SEEK_END = 2
- fseek sets file position relative to end

TICKS_SECOND = 1000000 | CLOCKS_PER_SEC

| 100*ZQHERTZ | CLK_TCK

directory_delim = “/” | “\\” | “:”

directory_delim_c = ‘/’ | ‘\\’ | ‘:’

3.0 The SCORE API

This section of the manual details the SCORE functions intended for use by C application programs. Functions here are presented in alphabetical order and are given with full ANSI C prototypes.

3.1 Hash Table Handling

These routines allow applications to manage multiple heterogeneous open hash tables. They form a generalized version of the hashing routines described in Kernighan and Ritchie's *The C Programming Language*. They are most conveniently used by a call by value language such as C.

C Binding: **void *SC_def_lookup(char *name, HASHTAB *tab)**

F77 Binding: **void *schlkp(integer nc, char *name, integer tab)**

SX Binding:

This function looks up the given name in the specified hash table and returns an object associated with the name if one was previously installed with **SC_install**. In contrast to **SC_lookup** this function strips off the additional information stored in the hashel associated with the *name* and only returns a pointer to the object member of the hashel.

An ASCII string, *name*, and a HASHTAB *, *tab* are the arguments to this function.

In the FORTRAN binding it is assumed that the pointer extension is present.

The return value is a pointer to an object if successful or a null pointer (NULL) if there is no hashel installed under *name*.

C Binding: **void SC_hash_clr(HASHTAB *tab)**

F77 Binding: **integer schclr(integer tab)**

SX Binding:

This routine clears out a hash table for possible reuse. It frees the memory allocated to the hashels in the table, but does not release any other memory. This is because there is no certain knowledge about the objects installed in the table.

The only argument is the HASHTAB *, *tab* to be cleared.

There is no return value from this function.

C Binding: **char **SC_hash_dump(HASHTAB *tab, char *pattern)**

F77 Binding:

SX Binding:

This function returns an array of the names installed in the given hash table. In this way the contents of a hash table may be checked at runtime. The names are alphabetically sorted according the C library function *STRCMP*. If the pattern specified is not null then only names matching the pattern are returned. In the pattern ‘*’ matches any number of characters and ‘?’ matches any single character. Since the number of elements installed in the hash table is a part of the hash table, the length of the array can also be accessed by application programs.

The arguments are the HASHTAB *, *tab* to be listed or dumped and an ASCII string, *pattern*.

The return value is actually a pointer to an array of ASCII strings (which are themselves pointers to character strings).

C Binding: **int SC_hash_rem(char *name, HASHTAB *tab)**

F77 Binding: **integer schrem(integer nc, char *name, integer tab)**

SX Binding:

This function removes the item specified by *name* from the indicated hash table.

An ASCII string, *name*, and a HASHTAB *, *tab* are the arguments to this function.

TRUE is returned if the operation is successfully carried out, and FALSE is returned otherwise.

C Binding: **hashel *SC_install(char *name, void *obj, char *type,**
HASHTAB *tab)

F77 Binding: **integer schins(integer nc, char *name, obj, integer nt, char *type,**
integer cp, integer tab)

SX Binding:

This function installs an object, *obj*, with *type* in the specified *hashtable* under the identifier, *name*. The procedure is an adaptation of the function described in Kernighan and Ritchie in *The C Programming Language*. It allows an application program to have multiple hash tables simultaneously. Furthermore by including a *type* with the installation of an object, applications can install different types of objects in a single table and distinguish between types so that they may be handled appropriately. The *type* is in no way used by the hash package. It is solely for the use of the application.

The given information is encoded in a structure termed a hashel (HASH ELement) which is installed in the specified hash table. This is the information returned by the function **SC_lookup**. If a single type of object is installed in a hash table, the function **SC_def_lookup** returns the object from the hashel directly and passes all responsibility to the application for keeping the types straight. Any type of data object call be installed, however it must be cast to an object pointer first.

In the FORTRAN binding, the argument *cp* is a flag requesting that the installed object is a copy of the argument pointed to by *obj*. In that case, schins will allocate one item of type *type* and copy one such item from *obj* into the new space. The new space will be installed in the hash table.

Input to this function is an ASCII string, *name*, a pointer to the object *obj* to be installed, an ASCII string, *type*, and an HASHTAB *, *tab*.

If successful **SC_install** returns a pointer to the hashel which was installed in the given hash table.

C Binding: **hashel *SC_lookup(char *name, HASHTAB *tab)**

F77 Binding:

SX Binding:

This function looks up the given name in the given hash table and returns an hashel associated with the name if one was previously installed with **SC_install**. In contrast to **SC_def_lookup** this function does not strip off any information stored in the hashel associated with the *name*.

An ASCII string, *name*, and a HASHTAB *, *tab* are the arguments to this function.

The return value is a pointer to an hashel if successful or a null pointer (NULL) if there is no hashel installed under *name*.

C Binding: **HASHTAB *SC_make_hash_table(int size, int docflag)**

F77 Binding: **integer scmkht(integer size, integer docflag)**

SX Binding:

This is the function which creates new hash tables. The size of a hash table should be a prime number for greatest efficiency. For C based applications, there are four #define'd sizes in the header, **schash.h**. They are **HSZSMALL**, **HSZSMINT**, **HSZLRINT**, **HSZLARGE** (31, 67, 127, and 521 respectively).

Applications can have documentation with the installed objects, and for efficiency in searching a hash table for documentation a flag is provided in the hash table structure telling whether or not the objects which are installed have documentation. The predefined flags in C applications are **DOC** and **NODOC** (1 and 0 respectively).

Two integers, *size* and *docflag*, are the input to **SC_make_hash_table**.

The return value is a HASHTAB * if successful and a null pointer (NULL) otherwise.

C Binding: **void SC_rl_hash_table(HASHTAB *tab)**

F77 Binding: **integer scrlht(integer tab)**

SX Binding:

Release the space previously allocated for the specified hash table by **SC_make_hash_table**. Call **SC_hash_clr** first to release the space associated with hash elements.

3.2 Lexical Stream Handling

The following routines assist code developers with integrating automatically generated lexical scanners into their applications and controlling them. The SC_lexical_stream structure encapsulates the lexical scanner, the I/O stream, and various buffers so that applications can apply arbitrarily many lexical scanners to any number of input streams. The SC_lexical_token describes a lexical token for parsers and other applications.

C Binding: **void SC_close_lexical_stream(SC_lexical_stream *str)**

F77 Binding: **integer scclls(integer strid)**

SX Binding:

This function closes the given lexical stream. Closing the lexical stream means closing the I/O stream if it is not stdin and releasing the buffer spaces as well as the SC_lexical_stream itself.

This function takes *str*, a pointer to the SC_lexical_stream, as argument.

This function has no return value.

See Also: SC_open_lexical_stream, SC_get_next_lexical_token, SC_scan.

C Binding: **SC_lexical_token *SC_get_next_lexical_token(SC_lexical_stream *str)**

F77 Binding:

SX Binding:

This function permits applications to view an input stream as a sequence of tokens returned by the lexical scanner bound in an SC_lexical_stream. It returns one token at a time until the end of the stream is encountered when NULL is returned. The function handles all reads of the input stream.

The argument to this function is: *str*, a pointer to a SC_lexical_stream.

A pointer to an SC_lexical_token is returned or NULL if the end of the input stream has been reached.

See Also: SC_open_lexical_stream, SC_close_lexical_stream, SC_scan.

C Binding: **SC_lexical_stream *SC_open_lexical_stream(char *name, int inbfsz, int strbfsz, PFInt scan, PFInt input, PFInt output, PFInt unput, PFInt wrap, PFInt more, PFInt less)**

F77 Binding: **integer scopls(integer nchr, char *name, integer inbs, integer strbs, integer function scan)**

SX Binding:

This function initializes a new lexical stream and returns a pointer to it. A lexical stream consists of: an input buffer; a string buffer for tokens; an output buffer for text not handled by the lexical scanner; an SC_lexical_token buffer; a lexical scanning function produced by a tool such as FLEX or LEX; and additional integer valued functions (some optional) to support the flexibility of the lexical scanner.

The main attempt here is to encapsulate the functions and data structures necessary to support an application in the use of multiple independent lexical scanners. Such capability can be used in conjunction with parsers or other applications where some sort of input language must be broken down into proper sized pieces.

The arguments are: *name*, the name of a file to attach to the lexical stream or NULL for stdin; *inbfsz*, the integer byte size of the input buffer or MAXLINE if 0; *strbfsz*, the integer byte size of string buffer or MAX_LEN_BUFFER if 0; *scan*, the lexical scanning function to be used (for example yylex); *input*, the single character supply function for the scanner (defaults to SC_lex_getc if NULL); *output*, the single character output function for the scanner (defaults to SC_lex_putc if NULL); *unput*, a function which pushes a single character at a time back onto the input stream (defaults to SC_lex_push if NULL); *wrap*, a function which handles end of input conditions for the scanner (defaults to SC_lex_wrap if NULL); *more*, a function to process more input for the scanner if required by the associated lexical rules; and *less*, a function to back up the input stream if required by the lexical scanner.

In the C binding, this function returns a non-NULL pointer to a newly allocated SC_lexical_stream if successful and NULL if not. In the FORTRAN binding, this function returns a non-negative integer identifier for the lexical stream if successful and -1 otherwise.

See Also: SC_close_lexical_stream, SC_scan, SC_get_next_lexical_token.

C Binding:

F77 Binding: **integer scrdls(integer strid, integer nc, char *s)**

SX Binding:

This function reads a line from the specified lexical stream's input stream into its input buffer.

The arguments to this function are: *strid*, an integer identifier specifying the lexical stream as returned by **scopls**; *nc*, an integer size of the character array *s*; *s*, a character array for the contents of the input buffer to be returned to the application.

This function returns the specified number of characters via the supplied character array. The return value is TRUE if successful and FALSE otherwise.

See Also: scopls, scclls, scrdls.

C Binding: **int SC_scan(SC_lexical_stream *str, int rd)**

F77 Binding: **integer scscan(integer strid, integer mxtok, integer width, char *tok, integer ntok, integer nctok, integer ixtok, integer toktyp, REAL tokval)**

SX Binding:

This function scans the current input buffer in the specified lexical stream and returns arrays of tokens.

The lexical rules are defined by the routine attached to the lexical stream when it is opened. However a default set of rules implements a FORTRANish syntax.

The string to be scanned is contained in the lexical stream. A call to **scrdls** can be made to read a new line of text from the input stream into the input buffer, otherwise the current contents of the input buffer will be scanned.

The default scanner, **f77lxr**, defines the following token types:

TYPE	NAME	EXAMPLE
1	DELIMITER	& ( ) , : < = > _ |
2	ALPHANUM	abc
3	INTEGER	10
4	REAL	1.2 6.0e10
5	OCTAL	17b
6	HEX	#17
7	OPERAND	.and.
8	STRING	“foo”
1000	HOLLERITH	3hFOO

Invoke the lexical scanner to traverse the current input buffer and identify the lexical tokens as defined by the lexical rules which generated the scanner. These are placed in a buffer of SC_lexical_tokens.

The other arguments to this function are: *str*, a pointer to a SC_lexical_stream; *rd*, an integer flag; *mxtok*, and integer maximum number of tokens to be returned; *width*, an integer character field width (e.g. char*8 => 8); *tok*, character array char**width*(*mxtok*) for returned tokens; *ntok*, an integer count of tokens available; *nctok*, an integer character length of each token; *ixtok*, an integer index in *tok* for each token; *toktyp*, type for each token; and *tokval*, an array of REAL numerical values for numerical tokens.

If the *rd* argument it TRUE an new line is read from the input stream into the input buffer before it is scanned.

In the C binding this function returns the number of lexical tokens found in the current input buffer. In the FORTRAN binding this function returns TRUE if successful and FALSE otherwise.

See Also: SC_open_lexical_stream, SC_close_lexical_stream, SC_get_next_lexical_token.

3.3 Association List Handling

Association lists are similar to hash tables in that they associate a key (in this case an ASCII string) with a “value”. They differ from hash tables in that the elements are chained together in a linked list instead of being organized in a hashed table. The principle advantage of association lists is lower memory overhead. The disadvantage is the relative inefficiency of checking each key until a match is found. The application developer must decide.

C Binding: **pcons *SC_change_alist(pcons *al, char *name, char *type, void *val)**

F77 Binding: **integer scchal(integer al, integer nn, char *pname, integer nt, char *ptype, integer nv, val)**

SX Binding:

This routine changes the value associated with *name* on the specified association list. If there is no value associated with the *name* key then the value is added to the list under the *name*. Because of this property is is permissible to pass a NULL association list into this function. In this way association list can be built up without explicitly creating them. It also means that the return list may not have the same address as the input list and consequently the return list must be used in all future calls to these routines.

The other arguments are: *type*, an ASCII string specifying the data type of the values(s) being added to the list (arrays may be used as values!); *val*, a pointer to the values to be added to the list; and, in FORTRAN, *nv*, the number of values to be added. In the C binding val must have been dynamically allocated with **MAKE**, **MAKE_N**, or **SC_alloc**.

C Binding: **pcons *SC_rem_alist(pcons *al, char *name)**

F77 Binding: **integer scrmal(integer al, integer nn, char *name)**

SX Binding:

Removes the named element from the association list, *al*. Because the specified element may be the first on the list, the modified list is returned and must be used in all future calls to these functions!

C Binding: **void SC_free_alist(pcons *al, int level)**

F77 Binding: **integer scrlal(integer al, integer level)**

SX Binding:

Free the specified association list, *al*. The *level* argument refers to how much information is to be released. If *level* is 1 the only the name key is released; if *level* is 2 only the value is released; and if *level* is 3 both are released.

C Binding: **pcons *SC_copy_alist(pcons *al)**

F77 Binding: **integer sccpal(integer al)**

SX Binding:

This routine returns a copy of the given association list if successful.

C Binding: **pcons *SC_append_alist(pcons *al1, pcons *al2)**

F77 Binding:

SX Binding:

This routine copies the contents of association list *al2* to association list *al1*. As always a NULL list for *al1* or *al2* is acceptable. The modified assocation list *al1* is returned if successful.

3.4 String Handling

The string handling routines in SCORE provide functionality which extends or supplements that available from the standard C library.

3.4.1 Predicates

The functions test strings for specific properties and return TRUE or FALSE depending on the outcome.

C Binding: **int SC_numstrp(char *s)**

Return TRUE iff the string is a valid representation of a a number.

C Binding: **int SC_intstrp(char *s, int base)**

Return TRUE iff the string is a valid representation of an integer in the specified base.

C Binding: **int SC_fltstrp(char *s)**

Return TRUE iff the string is a valid representation of a floating point number.

C Binding: **int SC_chrstrp(char *s)**

Return TRUE iff the string contain printable characters only.

C Binding: **int SC_blankp(char *string1, char *string2)**

Return TRUE iff *string1* is blank or a comment. A comment begins with a character in *string2* followed by a blank, tab, or end of line.

3.4.2 Pattern Matching

These functions do certain patterm matching operations.

C Binding: **char *SC_strstr(char *string1, char *string2)**

Find the first occurrence of *string2* in *string1*.

C Binding: **char *SC_strstri(char *string1, char *string2)**

Find the first case insensitive occurrence of *string2* in *string1*.

C Binding: **char *SC_str_replace(char *s, char * patto, char * pattn)**

Replace all non-overlapping occurrences of *patto* in string *s* with *pattn*.

C Binding: **int SC_regx_match(char *s, char *patt)**

Match string *s* against regular expression *patt*. '*' matches any number of characters. '?' matches any single character.

3.4.3 String Sorting

These functions are involved in sorting operations on strings.

C Binding: **void SC_string_sort(char **s, int number)**

Sort an array of character pointers by what they point to. The arguments are: *s*, an array of pointers to ASCII strings, and *number*, the number of strings.

3.4.4 String Operations Involving Case

These functions are used in connection with the case of the characters in strings.

C Binding: **int SC_str_icmp(char *s, char *t)**

Compare two strings ignoring case.

C Binding: **char *SC_str_upper(char *s)**

Convert a string to all uppercase.

C Binding: **char *SC_str_lower(char *s)**

Convert a string to all lowercase.

3.4.5 Tokenizers

These functions supplement the string tokenizing capability of the standard C library.

C Binding: **char *SC_firsttok(char *s, char *delim)**

Find the first token in a string.

C Binding: **char *SC_firsttokq(char *s, char *delim, char *quote)**

Find the first token or quoted token string in a string.

C Binding: **char *SC_lasttok(char *s, char *delim)**

Find the last token in a string.

3.4.6 Other String Operations

These additional string operations do not fitt into any of the above categories.

C Binding: **char *SC_strrev(char *s)**

Reverse a string in place.

C Binding: **int SC_char_count(char *s, int c)**

Count the occurrences of a specified character in a string.

C Binding: **char *SC_squeeze_blanks(char *s)**

Replace contiguous blanks in a stirng with a single blank and remove leading and trailing blanks.

3.5 Time and Date Handling

The SCORE routines dealing with time and date are provided mainly for true portability and standardization.

C Binding: **double SC_cpu_time(void)**

F77 Binding:

SX Binding:

Return the combined user and system processor time in seconds and microseconds consumed since the first call to ***SC_cpu_time***. Most systems limit actual time resolution to 0.01 seconds to reduce overhead.

C Binding: **double SC_wall_clock_time(void)**

F77 Binding:

SX Binding:

Return the wall clock time in seconds and microseconds since the first call to ***SC_wall_clock_time***. Most systems limit actual time resolution to 0.01 seconds to reduce overhead.

C Binding: **char *SC_date(void)**

F77 Binding:

SX Binding:

Get the time and date. This is a guaranteed to work version of the standard C library function **ctime**.

3.6 Memory Management

The following is a summary of routines which constitute a simple, fast, and small layer of memory management on top of the standard C library routines. It was originally implemented to allow PDBLib be able to trace down layers of indirections and write out arbitrary data structures, but it can also be used to do such things as array bounds checking on dynamically allocated arrays when routines have no other knowledge of a space than the pointer to it.

C Binding: **int SC_arrlen(void *ptr)**

F77 Binding: **integer scalen(pointer ptr)**

SX Binding:

Given a pointer to space allocated with **SC_alloc** or **SC_realloc** extract the number of bytes to which the pointer points and return it.WARNING: this function can fail to recognize the presence of a pointer allocated statically or with ***malloc***, ***calloc***, or ***realloc***! The FORTRAN binding is intended for use in FORTRAN implementations which support the pointer extension.

The number of bytes pointed to by *ptr* is returned if successful, and -1 if not.

C Binding: **void *SC_alloc(long nitems, long bytepitem)**

F77 Binding: **use scmake**

SX Binding: **memory management is automatic**

Allocate a new space in memory *nitems* x *bytepitem* long and return a pointer to it. The arguments are: *nitems*, the number of items (e.g. floats); and *bytepitem*, the number of bytes per item.

Returns a non-NULL pointer to a newly allocated space if successful and NULL if not. The pointer should be cast to the appropriate type in C.

C Binding: **int SC_free(void *ptr)**

F77 Binding: **use scfree**

SX Binding: **memory management is automatic**

Release the space pointed to by *ptr.* Returns TRUE if successful, FALSE otherwise.

C Binding: **int SC_mem_trace(void)**

F77 Binding: **integer scmemt(void)**

SX Binding:

Return the number of chunks of memory being managed or an error condition.

Return -1 if the forward and backward counts differ, -2 if a NULL pointer occurs in the chain, -3 if the link count exceeds the number of blocks; otherwise return the number of chunks of memory being managed.

C Binding: **void *SC_realloc(void *ptr, long nitems, long bytepitem)**

F77 Binding: **use screma**

SX Binding: **memory management is automatic**

Reallocate the space in memory associated with *ptr* so that it is *nitems* x *bytepitem* long and return a pointer to it. Copy the contents of the old space into the new space if necessary, but preserve the original contents pointed to. *Ptr* must be a pointer to a space previously allocated by **SC_alloc**. The other argument are: *nitems*, the number of items (e.g. floats); and *bytepitem*, the number of bytes per item.

Returns a non-NULL pointer to a newly allocated space if successful and NULL if not. The pointer should be cast to the appropriate type in C.

C Binding: **char *SC_strsave(char *s)**

F77 Binding:

SX Binding:

Allocate a new space in memory large enough to contain the char array *s*, copy its contents to the new space, and return a pointer to it. *S* must be a null terminated array of characters.

Return a non-NULL pointer to a newly allocated space if successful and NULL if not.

C Binding: **type *MAKE(type)**

F77 Binding: **use scmake**

SX Binding: **memory management is automatic**

Allocate a new space in memory the size of *type* and return a pointer to it which has been cast to *type **. In the C binding this is a macro and *type* is a primitive or derived type specifier.

Returns a non-NULL pointer to a newly allocated space if successful and NULL if not.

C Binding: **type *MAKE_N(type, long ni)**

F77 Binding: **integer scmake(pointer ptr, integer ni, integer bpi)**

SX Binding: **memory management is automatic**

Allocate a new space in memory for *ni* items the size of *type* (C) or *bpi* bytes each (FORTRAN) and return a pointer to it. In the C binding, which is a macro, *type* is a primitive or derived type specifier, and the return value is a pointer cast to *type* *. In the FORTRAN binding, which is intended for use in FORTRAN implementations which support the pointer extension, *ptr* is a pointer (e.g. ipa of pointer (ipa, a)).

The C binding returns a non-NULL pointer to a newly allocated space if successful and NULL if not. The FORTRAN binding returns 1 if successful and 0 if not.

C Binding: **type *REMAKE(void *ptr, type)**

F77 Binding: **use screma**

SX Binding: **memory management is automatic**

Reallocate the space in memory associated with *ptr* to the size of *type* and return a pointer to it, which has been cast to *type* *. Copy the contents of the old space into the new space if necessary. In the C binding this is a macro and *type* is a primitive or derived type specifier.

Returns a non-NULL pointer to a reallocated space if successful and NULL if not.

C Binding: **type *REMAKE_N(void *ptr, type, long ni)**

F77 Binding: **integer screma(pointer ptr, integer ni, integer bpi)**

SX Binding: **memory management is automatic**

Reallocate the space in memory associated with *ptr* to be *ni* items the size of *type* (C) or *bpi* bytes each (FORTRAN) and return a pointer to it. Copy the contents of the old space into the new space if necessary. In the C binding, which is a macro, *type* is a primitive or derived type specifier, and the returned pointer is cast to *type* *. The FORTRAN binding is intended for use in FORTRAN implementations which support the pointer extension.

The C binding returns a non-NULL pointer to a reallocated space if successful and NULL if not. The FORTRAN binding returns 1 if successful and 0 if not.

C Binding: **void SFREE(void *ptr)**

F77 Binding: **integer scfree(pointer ptr)**

SX Binding: **memory management is automatic**

Release the space pointed to by *ptr*. The FORTRAN binding is intended for use in FORTRAN implementations which support the pointer extension and always returns 1.

C Binding: **int SC_zero_space(int flag)**

F77 Binding: **integer sczrsp(integer flag)**

SX Binding: **not applicable**

If *flag* is zero (do not zero out memory), one (zero memory when allocated and when released), two (zero memory only when allocated), or three (zero memory only when released), return the new value; otherwise, return the old value. The default is for space to be zeroed out when allocated and when released. This is useful (in spite of the overhead) in order to spot the situation where space is freed when more than one pointer points to it.

3.7 Miscellaneous Routines

3.7.1 Bit Level Manipulations

C Binding: **int SC_bit_count(long c, int n)**

F77 Binding:

SX Binding:

Count the number of set bits in the specified number of bytes of a given long.

C Binding: **unsigned int SC_bit_reverse(unsigned int i, int n)**

F77 Binding:

SX Binding:

Reverse the specified number of bits of a given unsigned int.

3.7.2 Numeric Conversion

C Binding: **int SC_stoi(char *s)**

F77 Binding:

SX Binding:

Convert a string to an int. Return 0 if the string is null.

C Binding: **STRTOL(char *str, char **ptr, int base)**

F77 Binding:

SX Binding:

Convert a string to a long and return a pointer to any unconverted suffix. This macro invokes either the standard C library function **strtol** or the guaranteed to work SCORE equivalent.

C Binding: **double ATOF(char *s)**

F77 Binding:

SX Binding:

Convert a string to a double. This macro invokes either the standard C library function **atof** or the guaranteed to work SCORE equivalent.

C Binding: **double SC_stof(char *s)**

F77 Binding:

SX Binding:

Convert a string to a double. Return 0.0 if the string is null.

C Binding: **double STRTOD(char *nptr, char **endptr)**

F77 Binding:

SX Binding:

Convert a string to a double and return a pointer to any unconverted suffix. This macro invokes either the standard C library function **strtod** or the guaranteed to work SCORE equivalent.

3.7.3 Other Numeric

C Binding: **double ABS(double x)**

F77 Binding:

SX Binding:

Return the absolute value of a double.

C Binding: **max(a, b)**

F77 Binding:

SX Binding:

Return the greater of the two arguments.

C Binding: **min(a, b)**

F77 Binding:

SX Binding:

Return the lesser of the two arguments.

3.7.4 File Search

C Binding: **int SC_isfile(char *s)**

F77 Binding:

SX Binding:

Is the string the name of an existing file?

C Binding: **int SC_isfile_ascii(char *s)**

F77 Binding:

SX Binding:

Is the string the name of an existing ascii file?

C Binding: **char *SC_search_file(char **directory, char *s)**

F77 Binding:

SX Binding:

Search a list of directories for a file and return the full path name if the file exists.

3.7.5 I/O

C Binding: **PRINT**

C Binding: **GETLN**

3.7.6 Interrupts

C Binding: **void SC_init(char *msg, PFByte fnc, int sighand, PFByte sigfnc, int bfhand, char *bf, int bfsize)**

F77 Binding:

SX Binding:

Setup the interrupt handler, top level longjump, and output buffering for an application.

C Binding: **void SC_interrupt_handler(int sig)**

F77 Binding:

SX Binding:

Handle interrupts in a default sort of way.

3.7.7 Other

C Binding: **void SC_banner(char *s)**

F77 Binding:

SX Binding:

Display the input string as a banner on **stdout.**

C Binding: **void SC_pause(void)**

F77 Binding: **integer scpaus(void)**

SX Binding:

Pause until a character arrives on **stdin.**

4.0 Examples

This section will illustrate the use of the SCORE functions.

5.0 Related Documentation

SCORE is one part of a collection of libraries called PACT. It provides the foundation for PACT. Its functionality includes memory management, hash table functions, and string handling support.

The list of PACT Documents is:

PACT User's Guide, UCRL-MA-112087

SCORE User's Manual, UCRL-MA-108976 Rev.1 (this document)

PPC User's Manual UCRL-MA-108964 Rev.1

PML User's Manual, UCRL-MA-108965 Rev.1

PDBLib User's Manual, M-270 Rev.2

PGS User's Manual, UCRL-MA-108966 Rev.1

PANACEA User's Manual, M-276 Rev.2

ULTRA II User's Manual, UCRL-MA-108967 Rev.1

PDBDiff User's Manual, UCRL-MA-108975 Rev.1

PDBView User's Manual, UCRL-MA-108968 Rev.1

SX User's Manual, UCRL-MA-112315

T

U

V

W

Z