SCORE DOCUMENT

Stewart A. Brown
Dennis Braddy

Table of Contents

Examples

Related Documentation

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.

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.

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

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 = □ /□ |□ \\□ |□ :□

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.

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.

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.

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.

String Handling

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

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.

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.

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.

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.

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.

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.

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.

Memory Management

Overview of the SCORE Memory Manager

SCORE implements a layer of memory management over the system memory manager to provide several services that most system memory managers don't. These services provide information about managed memory which enable advanced usage of dynamic memory or they assist applications in debugging situations involving overwritten memory, leaked memory, or recycled memory.

System memory managers hand out pointers to dynamically allocated spaces. Sometimes in a generic routine of an application it would be useful to be able to lookup the number of bytes and the type of the data to which a pointer refers. The SCORE memory manager provides this service. The routines SC_arrlen and SC_arrtype will return the number of bytes and an integer type index for any pointer allocated by the SCORE memory manager. The PACT PDB library uses this capability extensively.

Some applications use dynamic memory in very complex ways which make it difficult to know when it is safe to release a piece of dynamically allocated memory. Reference counting is a good technique in many cases. The SCORE memory manager provides functions to let applications tag pointers with the number of references and SC_free (and SFREE) act properly with respect to the number of references.

Application developers frequently need to monitor their memory usage for one reason or another. The SCORE memory manager provides functions to keep track of how much memory has been allocated and freed. It also can compare snapshots of its own heaps of memory in order to detect and report memory leaks. Some versions of the SCORE memory manager allocation routines allow you to supply a "name" with the allocation call. In such a case the memory leak report tells you the "name" of the leaked memory. This makes it easier to identify the leaked memory than a simple memory address does.

The SCORE memory manager provides diagnostics to monitor the integrity of the heap. This can be extremely useful in detecting when some dynamic space is over or under indexed and memory consequently corrupted. This scheme is not 100% effective because it isn’t doing index checking on writes, however it very often detects that memory has been corrupted by an over or under indexed write.

Another useful feature is that the application can specify whether dynamic memory bytes are to be initialized to zero on allocation and/or reset to zero after being freed. This patterning of memory can be very effective in finding problems where pointers are being incorrectly used after being freed or when multiple pointers refer to the same chunk of dynamic memory.

The SCORE memory manager manages its own heap(s) in such a way as to be more efficient than many system memory managers. Consequently in applications where the dynamic memory is churned heavily, it can yield marked improvement in performance. In thread parallel applications it provides an even bigger benefit by managing heaps on a per thread basis and avoiding the overhead of locking and unlocking memory management operations.

Description of the SCORE Memory Manager

In the model used by the SCORE memory manager, memory is thought of as being dispensed from a bin. Each bin contains blocks of memory of a certain size. A request for a certain number of bytes of memory finds a block from the bin containing the smallest sized blocks big enough to hold the requested number of bytes. Requests for more space than the largest bin holds are turned over to the system memory manager.

The SCORE memory manager maintains a data structure called a heap for each active thread in an application. Each heap contains an array of lists of free memory blocks or free lists. The array of free lists is indexed by byte size. Each free list represents the bin from which a certain size of memory will be allocated.

In this strategy it is recognized that a fraction of the managed memory will be wasted in that it will be unused some of the time. The application can control this by determining the byte sizes of the bins. If you have a large number of bins you will have a smaller amount of wasted space. On the other hand, a large number of bins means that the memory manager heaps take up more space. Fortunately you can control this with SC_configure_mm.

A common kind of memory usage involves allocating and freeing small things repeatedly. These may be single instances of structures, strings, or small arrays of numbers. You would like to be fairly space efficient for such operations. It is also common enough to have a range of large memory spaces which are allocated and freed in cycles. You want to be able to cover a wide range of sizes but perhaps don't mind wasting a modest fraction of the space. These considerations led to a binning strategy in which there is a linear region which increases in size by 8 bytes at a time (i.e 8, 16, 24, 32, ...) and some point the binning cuts over to exponential to permit a wide range of sizes to be covered.

When memory is requested, the number of bytes is hashed to a bin. The first element of the free list for that bin is popped off the free list and returned. It may or may not be zeroed out depending on the flag set by SC_zero_space. It is also marked as having a reference count of 0. Each block is also inserted into a doubly linked list of active blocks. This permits the SCORE memory manager to check the integrity of managed memory as the application runs (via calls to SC_mem_chk).

Each memory block has a header that contains, in addition to the reference count, the size in bytes of allocation request. This can be accessed by SC_arrlen at any time until the block is freed.

When a pointer to allocated memory is handed to SC_free, several things are done. The reference count is decremented. If the new count is less than 1, the block is added to the free list of the bin whose size it hashes to. The block may or may not be zeroed out depending on the value of the flag set by SC_zero_space.

When the SCORE memory manager finds an empty free list while trying to honor an allocation request, it goes out to the system memory manager. It gets a "big" block of space from the system memory manager and then breaks it into the as many blocks as possible. Each block is the size appropriate to the bin it is intended for. It will always get at least one blocks worth of space. Each of these blocks is added to the free list for the bin. The size of the "big" block can be set with SC_configure_mm. This is the maximum block size parameter discussed below.

As an application is running, the SCORE memory manager has two kinds of memory blocks: blocks on the free lists; and blocks in the linked list of active blocks. Using the function SC_mem_chk, each kind of memory can be checked for integrity. This is often useful to detect when memory overwrites in the application have corrupted memory.

A new feature of the SCORE memory manager is that the memory allocated in the "big" blocks and the memory allocated to honor requests above the maximum bin size can be protected via the mprotect system call. This will be referred to as guarded or protected memory. The functions which manage these protected blocks of memory form a subsystem of the SCORE memory manager referred to as the Protected SCORE Memory Manager (PSMM). You can control this feature with the new call, SC_use_guarded_mem.

All this means that attempts to access memory before or after the allocated space results in a SIGSEGV being issued. This is very handy for tracking down certain application bugs involving dynamic memory usage. Because the mprotect call deals in units of memory pages, usually only access beyond the memory block cause a SIGSEGV to be raised. By using SC_configure_mm to set the maximum bin size to 0 all memory allocated by the SCORE memory manager can be protected.

Protected memory doesn't solve all memory related problems. You can still have bad memory accesses that don't hit protected memory. The use of SC_mem_chk is good for finding some other kinds of bad memory accesses. When trying to track down bad memory acccess bugs different strategies will need to be employed in different situations. Sometimes the bad memory accesses come from other libraries which are loaded into your application. A new library has been added to PACT, libpsmm.a. This provides a malloc, calloc, free, and realloc interface to the PSMM. If you link with libpsmm.a before the system version (often in libc.a) you will have all memory allocations, even from code you don't control, protected.

It is a very sad fact of life that when an application corrupts memory, it doesn't always immediately crash. Instead, it wanders off somewhere and dies. Where it eventually dies may have nothing to do with where the damage occurred. In particular, it sometimes dies on the next attempt to allocate memory. If you see your application crash in the SCORE memory manager, you should immediately suspect that your application has corrupted memory. The SCORE memory manager has been used heavily in many different kinds of environments for many years. There may still be bugs in it, but you are unlikely to have found one. You should check your application first as hard as that may be to do.

Configuring the Memory Manager

You may configure the SCORE memory manager to optimize its performance for your application's pattern of memory usage. This is done by calling SC_configure_mm. It should be called before any other operations by the SCORE memory manager. The SCORE memory manager will call SC_configure_mm itself to setup the defining parameters with default values chosen to give good performance for an "average" memory usage scenario. SC_configure_mm works only once. Any subsequent calls are reduced to no-ops.

The controls that you exert are to set: the cut over point between the linear and exponential region referred to as the maximum block size for the linear region; the target maximum managed block size; the bin size ratio in the exponential region; and the maximum block size for requests from the system memory manager.

SC_configure_mm computes the actual number of bins to set up for each heap. You supply numbers that characterize or reflect the sizes of blocks of memory you use and how much space you are willing to be wasted.

If the size of an allocation request exceeds the maximum managed block size, this request is passed on to the system memory manager. Such a block will also be passed back to the system memory manager when freed. In a parallel application, each such transaction would have a lock around the allocation and free. On the other hand, when requests are less than the maximum managed block size, any transaction would only lock when memory had to be obtained from the system memory manager to replenish an empty free list. This is why the SCORE memory manager performs well in parallel applications. It is also why spaces allocated in a thread must be freed in the same thread. Because there are no locks the application must avoid the potential conflict inherent in having different threads operate on the same heap at the same time.

The default values are: 128 bytes for the maximum block size for the linear region; 4096 bytes for the target maximum managed block size; 4096 bytes for the maximum block size for requests from the system memory manager; and 1.1 for the bin size ratio in the exponential region. These values result in 53 bins with upper boundaries:

```
8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 96, 112, 128
```

for the linear region; and

```
144, 168, 192, 216, 248, 280, 312, 352, 392, 440, 488,
544, 608, 672, 744, 824, 912, 1008, 1112, 1232, 1360,
1504, 1664, 1840, 2032, 2240, 2472, 2728, 3008, 3312,
3648, 4016, 4424, 4872, 5368, 5912, 6512, 7168, 7888, 8680
```

for the exponential region. Notice that the target maximum managed block size was exceeded. It is only a target. We must guarantee that a chunk of memory is aligned to an 8 byte boundary. As the bin boundaries are being computed, each boundary is rounded up to the nearest 8 bytes and the ratio is applied get the next boundary. The more bins the more accumulated error.

Now observe that the worst case of absolute memory waste occurs when you request a 7889 byte chunk of memory. You will actually be getting an 8680 byte chunk. You will have wasted 791 bytes or 9.1% of this memory. On the other hand, you have sped up the allocation considerably by making it possible to hash right to a pool of available memory. This is the tradeoff. If you select a smaller ratio, you get less waste. You also get more memory tied up in the memory manager. These are all choices that you get to make.

Tracking Down Memory Leaks

A nagging problem with dynamically allocated memory is improper accounting of memory blocks. The worst case of this is the memory leak. That is when an application inadvertently loses any reference to a block of memory and is therefore unable to free it. The SCORE memory manager supplies two functions, SC_mem_map and SC_mem_monitor, to help identify memory accounting problems. SC_mem_map writes an ASCII text file listing all of the memory blocks (not marked as permanent) currently being managed by the SCORE memory manager. The address, byte length, and, in some cases, the name associated with each memory block is listed. SC_mem_monitor is used to monitor applications for memory leaks. It takes a differential look at the memory being managed by comparing the results of an SC_mem_map before and after a region of code. To use it you make a call at the beginning of a section of code you want to monitor. This first call returns the number of bytes of memory which the SCORE memory manager currently thinks the applications has actively allocated. The second call at the end of the section of code being monitored takes that number of bytes as the first argument. If the number of bytes differs SC_mem_monitor concludes there is a leak. This may not be precisely true in that the change in the number of bytes could be positive or negative. In the first case memory was allocated between the two calls which was not freed. In the second case memory was freed between the two calls which was not reallocated.

When a leak is detected there are 3 alternative treatments which SC_mem_monitor does: 0, do nothing; 1, report the number of leaked bytes; and 2, report the number of leaked bytes and list the blocks of memory involved. This last is most detailed and is very useful for tracking down which space(s) leaked. The report of the number of leaked bytes is made into a character array supplied by the application.

Here is an example of the use of SC_mem_monitor:

```
int rv, ret;
char s[MAXLINE];

rv = SC_mem_monitor(-1, 2, "A", s);

    ... your code here ...

ret = SC_mem_monitor(rv, 2, "A", s);
```

In the example you want to determine whether or not a section of code (i.e. ... your code here ...) is leaking memory. You make a call to SC_mem_monitor before the code of interest saving the return value which will be used as input to the second call after the section you are checking. When you are doing the first call to get the current state of the memory and reporting the number of bytes active, the first argument must have a value of -1. Now when you use SC_mem_monitor, you must decide what or how much information you want from it. In this case we are asking for level 2 information which is the most detailed. After the first call there will be a temporary file, mem-u-A.before in the directory where the program is being run. This is the raw output of SC_mem_map. This will be sorted alphabetically into the file, mem-s-A.before. When the second call is made a file mem-u-A.after will be written by SC_mem_map. It will be sorted into mem-s-A.after. Next the two sorted files are diff'ed into a file mem-u-A.diff. If there is a difference, this file is printed to the terminal. All the files except, mem-u-A.diff are deleted as soon as possible.

If the second argument to SC_mem_monitor had been 1, the SC_mem_map calls would not have been made and there would have been no files created. Only the number of bytes difference would have been reported.

NOTE: All arguments but the first must be the same in the pairs of SC_mem_monitor calls.

Memory Manager Functions

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: void SC_configure_mm(long mxl, long mxm, long bsz, double r)
F77 Binding: use scmake
SX Binding: memory management is automatic

Configure the SCORE memory manager. This is used to set parameters which control the function of the SCORE memory manager. When matched to the pattern of memory usage in an application the result is improved performance with respect to the allocation and freeing of dynamic memory. See the discussion of the SCORE memory manager above. The arguments are: mxl, the maximum block size for the linear region; mxm, the target maximum managed block size; bsz, the maximum block size requested from the system; and r, the bin size ratio in the exponential region.

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_chk(int n)
F77 Binding: integer scmemc(integer n)
SX Binding:

Return the number of chunks of memory being managed or an error condition. The 1 bit causes the active allocated memory to be checked. The 2 bit causes all of the free lists to be checked.

Return -1 if the active memory has been corrupted, -2 if free lists have been corrupted, -3 if both have been corrupted; otherwise return the total number of chunks of memory being managed. NOTE: memory may have been sufficiently corrupted that this check will result in a segmentation violation. This fact does NOT invalidate this function as a diagnostic.

C Binding: int SC_mem_map(FILE *fp, int flag)
F77 Binding: integer scmemp(integer old)
SX Binding: (memory-map fp)

This function is used to dump a map of the memory blocks which the SCORE memory manager currently tracks. The map is printed to the supplied file (could be stdout) in C and to the terminal in FORTRAN. If the flag has the 1 bit set, memory being used by PACT is not listed. This helps applications see only their own memory.

C Binding: int SC_mem_monitor(int old, int lev, char *id, char *msg)
F77 Binding: integer scmemm(integer old, integer lev, integer ni, character*(*) id, integer ncm, character*(*) msg)
SX Binding: (memory-monitor old lev id)

This function is used to monitor applications for memory leaks. See the section Tracking Down Memory Leaks above for a detailed discussion of how to use it. When a leak is detected there are 3 alternative treatments which SC_mem_monitor does: 0, do nothing; 1, report the number of leaked bytes; and 2, report the number of leaked bytes and list the blocks of memory involved. This last is most detailed and is very useful for tracking down which space(s) leaked. The report of the number of leaked bytes is made into a character array supplied by the application.

The arguments are: old, the number of bytes expected (-1 for the first call and the return value from the first call for the second call); lev, the treatment of a leak condition; id, a base identifier for temporary files; and msg, a character array from the application to hold an error message.

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: void SC_use_guarded_mem(int on)
F77 Binding: integer scgmem(integer on)
SX Binding:

Turn on memory protection if the argument is not 0 and turn it off otherwise.

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.

Fortran binding returns 1 iff successful.

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 input flag is set to 1 memory will be zeroed out when allocated and when released. If flag is 2 memory will be zeroed out on allocation only. If flag is 3 memory will be zeroed out on release only. If flag is 0 memory will not be zeroed out. The zeroing of memory on release can be useful (in spite of the overhead) in order to spot the situation where space is freed when more than one pointer points to it. The default is for space not to be zeroed out.

Returns the old value of the flag.

Miscellaneous Routines

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.

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.

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.

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.

I/O

C Binding: PRINT

C Binding: GETLN

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.

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.

Parallel Programming with SCORE

PACT supports parallel programming in both SMP and distributed models. In SCORE there is a portable interface over various standards for threads. The purpose of this is to allow applications to deal with threads at a very low level while remaining portable. There is also a higher level of thread related routines which help applications manage threads (e.g. thread pools) and easily parallelize sections of code.

For distributed parallel support see PPC.

Low Level Thread Interface

High Level Thread Interface

C Binding: void SC_init_threads(int np, PFVoid tid)
F77 Binding:
SX Binding:

Initialize np threads. If the function tid is supplied it will be used to set its argument to a value between 0 and np-1. If tid is NULL SCORE will use its own built in function.

C Binding: void SC_init_tpool(int np, PFVoid tid)
F77 Binding:
SX Binding:

Initialize a pool of np threads. If the function tid is supplied it will be used to set its argument to a value between 0 and np-1. If tid is NULL SCORE will use its own built in function. The threads are created and wait to be given work by SC_do_threads.

C Binding: void SC_chunk_loop(PFPByte fnc, char *msg, int mn, int mx, int serial, byte *argl)
F77 Binding:
SX Binding:

This routine is used to break a loop into chunks each of which is done by a different thread. fnc is a function which does the body of the loop, mn and mx are the loop limits (i.e. for (i = mn; i < mx; i++)), serial is a flag which will force the loop to be done sequentially (this is useful for debugging), and argl contains a pointer that will be passed into fnc. The intention is that the application will define a struct which contains essentially the information that would be passed as arguments to fnc if the underlying thread mechanism were sufficiently general. An instance of the struct will be loaded up and a pointer to it passed to fnc.

C Binding: void SC_chunk_split(int *pmn, int *pmx, void **prv)
F77 Binding:
SX Binding:

This routine is used by the function fnc passed to SC_chunk_loop. It returns the index limits to be covered by the current thread. It also sets up the return value for the function fnc. This return value, prv, is used to check that the entire loop range is completed.

C Binding: void SC_queue_work(PFPByte fnc, int serial, byte *argl)
F77 Binding:
SX Binding:

This routine is used to break a loop into chunks each of which is done by a different thread. This function is used when the body of the loop does different amounts of work depending on the data involved. Each thread asks for the next index using SC_queue_next_item or a similar application supplied function which simply locks and dispenses the next item of work (or an index to it). In such a case, it is a good idea to effect a sort with the greatest amount of work coming first and the least coming last. This gives good load balancing when the number of threads is less than the number of work items to be processed. fnc is a function which does the body of the loop, serial is a flag which will force the loop to be done sequentially (this is useful for debugging), and argl contains a pointer that will be passed into fnc. The intention is that the application will define a struct which contains essentially the information what would be passed as arguments to fnc if the underlying thread mechanism were sufficiently general. An instance of the struct will be loaded up and a pointer to it passed to fnc.

C Binding: int SC_queue_next_item(int ng)
F77 Binding:

SX Binding:

This routine is used by the function fnc passed to SC_queue_work. It returns the integer index of the next item of work to be done. There is nothing special about this function. The application can dispense work anyway it sees fit. This is simply a very useful special case.

Examples

This section will illustrate the use of the SCORE functions.

Example of parallelizing a loop with SC_chunk_loop:

```
           .
           .
           .

/* a simple loop */
   for (i = 0; i < n; i++)
       a[i] = sqrt(b[i]);
           .
           .
           .
```

after adding a struct definition and a function embodying the loop becomes:

```
typedef struct sqd
   {double *a;
    double *b;};

void *body(arg)
   void *arg;
   {int i, mn, mx;
    double *a, *b;
    void *rv;
    sqd *par;

    SC_chunk_split(&mn, &mx, &rv);

    par = (sqd *) arg;
    a = par->a;
    b = par->b;

    for (i = mn; i < mx; i++)
        a[i] = sqrt(b[i]);

    return(rv);}

           .
           .
           .

/* the same loop parallelized */
     {sqd par;
      par.a = a;
      par.b = b;
      SC_chunk_loop(body, 0, n, TRUE, &par);}
           .
           .
           .
```

Example of parallelizing a loop with SC_queue_work:

```
           .
           .
           .
```

```
/* a simple loop where work takes different times for different
 * input data values
 */
   for (i = 0; i < n; i++)
       a[i] = work(b[i]);
             .
             .
             .
```

after adding a struct definition and a function embodying the loop becomes:

```
typedef struct sqd
   {int n;
    double *a;
    double *b;};

void *body(arg)
   void *arg;
   {int i, n;
    double *a, *b;
    sqd *par;

    par = (sqd *) arg;
    a = par->a;
    b = par->b;
    n = par->n;

    for (i = SC_queue_next_item(n); i != -1; i = SC_queue_next_item(n))
        a[i] = work(b[i]);

    return(NULL);}

             .
             .
             .

/* the same loop parallelized */
     {sqd par;
      par.a = a;
      par.b = b;
      par.n = n;
      SC_queue_work(body, TRUE, &par);}
             .
             .
             .
```

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	Current 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

Last modified: Tue Feb 20 04:49:46 PDT 2001

Modified by: rodrigues2@llnl.gov