

# Documentation for SimCommand.h and SimCommand.c

Steven Andrews, © 2004-2015

## Header

```
#ifndef __SimCommand_h__
#define __SimCommand_h__

#include "queue.h"
#include "string2.h"

#define SCMDCHECK(A,B) if(!(A)) {if(cmd) strncpy(cmd->erstr,B,STRCHAR);return CMDwarn;}

enum CMDcode
{CMDok,CMDwarn,CMDpause,CMDstop,CMDabort,CMDnone,CMDcontrol,CMDobserve,CMD
manipulate,CMDctrlORobs,CMDall};

typedef struct cmdstruct {
    double on;           // first command run time
    double off;          // last command run time
    double dt;           // time interval between commands
    double xt;           // multiplicative time interval
    Q_LONGLONG oni;      // first command run iteration
    Q_LONGLONG offi;     // last command run iteration
    Q_LONGLONG dti;      // iterations between commands
    Q_LONGLONG invoke;   // number of times command has run
    char *str;           // command string
    char *erstr;         // storage space for error string
    int i1,i2,i3;        // integers for generic use
    double f1,f2,f3;     // doubles for generic use
    void *v1,*v2,*v3;    // pointers for generic use
    void (*freefn)(struct cmdstruct*); // free command memory
    listptrdd data;      // data for observation commands
} *cmdptr;

typedef struct cmdsuperstruct {
    queue cmd;           // queue of normal run-time commands
    queue cmdi;          // queue of integer time commands
    enum CMDcode (*cmdfn)(void*,cmdptr,char*); // function that runs commands
    void *cmdfnarg;      // function argument (e.g. sim)
    int iter;            // number of times integer commands have run
    double flag;         // global command structure flag
    int maxfile;         // number of files allocated
    int nfile;           // number of output files
    char root[STRCHAR];  // file path
    char froot[STRCHAR]; // more file path, used after root
    char **fname;        // file name [fid]
    int *fsuffix;        // file suffix [fid]
    int *fappend;        // 0 for overwrite, 1 for append [fid]
    FILE **fptr;         // file pointers [fid]
    int precision;       // precision for output commands
}
```

```

char outformat;           // output format, 's' or 'c'
int maxdata;              // number of data lists allocated
int ndata;                // number of data lists used
char **dname;             // data list names [did]
listptrdd *data;          // data lists
} *cmdssptr;

```

## History

- 1/10/04 Routines moved to this library from Smoldyn
- 1/20/04 Added invoke member to command structure. Also changed declaration of command executing function.
- 1/22/04 Made more changes to the command superstructure, added some functions, and modified others.
- 6/24/04 Changed scmdstr2cmd so that it now allocates the command queue or expands the queue as needed. Also added integer queue stuff to commands and command superstructure and erstr to commands.
- 11/29/06 Added command storage i1, i2, i3, f1, f2, f3, v1, v2, v3, and freefn to command structure.
- 3/12/07 Added xt member to cmdstruct
- 5/25/07 Modified type 'x' command execution
- 11/20/07 Added and implemented enum CMDcode and added scmdcmdtype
- 11/25/07 Replaced all float data types with doubles
- 11/26/07 Added scmdcmdtime
- 4/15/08 Changed scmdexecute with the simdt input parameter
- 1/12/09 Added command timing options A, B, &, and I
- 1/14/09 Changed integer queue elements oni, offi, and dti from int to long long int
- 1/19/09 Fixed bugs with long long
- 2/8/11 Added dynamic memory allocation to the superstructure for files and rewrote scmdsetfnames, added maxfiles and fappend to superstructure
- 3/27/11 Modified scmdsetfroot
- 4/20/11 Added flag to command superstructure, and functions scmdsetflag and scmdreadflag
- 6/28/11 Added scmdaddcommand
- 4/16/12 Added optional smoldynfuncs.h dependency. This is required for the simLog function. If it isn't present, the library uses printf instead.  
 Edited scmdexecute and scmdoutput to optionally send text to simLog instead of printf.  
 Added type casting to scmdsetfnames for good style and C++ compatibility.
- 7/18/12 Added scmdfprintf function and sigfig element.
- 6/9/15 Added scmdflush.
- 10/6/15 Edited scmdgetfptr to simplify stdout and stderr operation. Fixed some minor bugs in scmdopenfiles, scmdoverwrite, and scmdincfile, all having to do with tests for stdout and stderr.
- 9/10/20 Dilawar had converted scmdfprintf to a C++17 class, but it caused compile errors and didn't add functionality, so I reverted back to the original.

12/4/20 Added data element to command structure, rewrote scmdgetfptr, and more updates. These functions now support NULL file pointers (meaning no output).

When writing simulation programs, it has proven useful to include a runtime command interpreter in the program so that various commands can be executed at specific times during the simulation. These commands are stored as strings and are passed on to a parser and executer at the proper time. This library contains most of the framework necessary for this interpreter. As a primary use of commands is to output simulation results to text files, the command framework also manages a list of output files.

### **Data structures**

The structure `cmdstruct`, pointed to with `cmdptr`, contains the information for one command. `on` is the time that the command turns on, `off` is the time that it turns off, `dt` is the time step between command executions, `xt` is the time multiplier and is only used if  $xt > 1$ , `invoke` is the number of times that the command has been invoked so far (it equals one during the first command call), and `str` is the command string. `oni`, `offi`, and `dti` can be used instead of `on`, `off`, and `dt` to indicate that a command should be run every `dti` iterations. Command execution intervals are never shorter than `dt` but are sometimes longer than `dt` because they can only be executed at the times when `cmdcheck` is called. If  $xt \leq 1$ , the target command times are: `on`, `on+dt`, `on+2dt`, ..., `off`; if  $xt > 1$ , target command times are: `on`, `on+dt`, `on+xt*dt`, `on+xt*dt+xt2*dt`, ..., `off`.

Note that the `cmdstruct` owns the string, meaning that the string is allocated when a `cmdptr` is allocated and freed when the `cmdptr` is freed. `erstr` is storage space for an error message that can be passed from the command back to the calling program. There is additional storage space that can be used by the command, although it does not have to be used, which is `i1`, `i2`, `i3`, `f1`, `f2`, and `f3`; these are all initialized to 0 and keep their values from one command call to the next. Similarly, `v1`, `v2`, and `v3` are general purpose void\* pointers that are initialized to NULL. If memory is allocated for any of these by the command, then the command should also register the address of a function that will free the memory in `*freefn`. The memory will automatically be freed, using this function, when the command will no longer be executed.

`cmdsuperstruct`, which is pointed to by `cmdssptr`, is a structure that contains the list of runtime commands, the address of a function that is supposed to execute them, and information about the output files and data arrays. `cmd` is the regular queue of commands, sorted in order of their next execution times. `cmdi` is the queue of commands that are run every `iter` iterations and for which `dt` is ignored. In the queues, the object key is the `on` value of the command and the object item is a pointer to the command structure. `cmdfn` is a pointer to the function in the main program that is called to take care of commands. It is sent the argument `cmdfnarg`, which is unchanged by the routines here, the command, and the command string; see below. `maxfile` is the number of allocated file spaces, `nfile` is the number of output files, `root` is a root name used before `froot`, which is another root name and is used for all output files, `fname` is a list of file names for the various output

files, `fsuffix` is a list of file name suffixes, and `fptr` is a list of file streams for the output files. Complete file names are a concatenation of `root`, `froot`, the file name, and the suffix if there is one. Usually, `root` is the directory in which the configuration file is located, and `froot` is a subdirectory for output results.

Data files are similar to file output, but store data within the command superstructure rather than in files and are somewhat simpler. There are `ndata` arrays stored here, of a total of `maxdata` that are allocated, each in a list called `data` and given a name that is stored in `dname`. These data are only accessible, at present, through the `libsmoldyn` interface. They are stored as a matrix of doubles, which is appropriate for many data outputs but not ideal for some things, like species numbers and molecule states.

The command superstructure owns all lists and memory pertaining to output files, but `cmdfn` and `cmdfnarg` are merely pointers that are neither allocated nor freed in this library. `flag` is simply a number that individual commands can read and/or set, for communication between commands, and is not used in the command handling at all.

The function in the main program that takes care of commands is called `docommand` in `Smoldyn`. It separates the command string into the first word, which says what the command is, and the rest of the string which contain the parameters for the command, and then it calls the appropriate function to take care of the command. `docommand` is made available to the `SimCommand` library by sending its address to `scmdssalloc` as `&docommand` during initial structure setup. It is called later on, as needed, by `scmdexecute`. In this calling, `docommand` is sent a `void*` type conversion of `sim`, which is a structure for the entire simulation parameters and state, a pointer to the command that is to be executed (`cmd`), and the command string. In this case, the command string is always equal to `cmd->str`, and so is redundant. However, some commands can invoke other commands directly, in which case they call `docommand` with a valid string but either the original command or a `NULL` value for the `cmd` parameter. This means that all commands need to be able to handle `cmd` being `NULL` or the command string in `cmd` being different from the string in `line`. For example, the conditional command in `Smoldyn` called “`ifno`” first checks the condition and then, if appropriate, it calls `docommand` with the remainder of the command string.

### **Internal routine**

```
void scmdcatfname(cmdssptr cmds,int fid,char *str);
```

`scmdcatfname` concatenates all the portions of a file name together, for file number `fid`, into the string `str`, which should already be allocated to size `STRCHAR`. If the total file name is too long, it is truncated at size `STRCHAR`.

### **Externally available functions**

#### **Non-file functions**

```
char *scmdcode2string(enum CMDcode code,char *string);
```

Converts enumerated command code to a string. `string` needs to be pre-allocated. `string` is returned for easy function nesting.

```
cmdptr scmdalloc(void);
```

scmdalloc allocates a command structure, including the string and the error string. The strings are allocated to the fixed size STRCHAR, which is defined in the file string2.h to be 256.

```
void scmdfree(cmdptr cmd);
```

scmdfree frees a command structure.

```
cmdssptr scmdssalloc(int (*cmdfn)(void*,cmdptr,char*),void *cmdfnarg,char *root);
```

scmdssalloc allocates a minimal command superstructure. cmdfn should be sent in pointing to a function that can execute the commands and cmdfnarg is the first argument of that function. For example, in the Smoldyn program, the cmdfn is sent in as &docommand and cmdfnarg is sent in as (void\*)sim, because sim is a structure that contains all information about the current state of the simulation and is required for most commands. root is the file directory root. The only memory that is allocated is for the superstructure itself. In particular, the command queues are not allocated.

```
void scmdssfree(cmdssptr cmds);
```

scmdssfree frees a command superstructure and all internal elements except for cmdfn and cmdfnarg.

```
int scmdqalloc(cmdssptr cmds,int n);
```

scmdqalloc allocates the command queue to size n and sets up the queue indexing parameters. It returns 0 for no error, 1 for insufficient memory, 2 for no cmds, or 3 if a queue was already allocated (and thus should not be written over). This function is called automatically by scmdstr2cmd, so there is no longer any need for it to be called from externally.

```
int scmdqalloci(cmdssptr cmds,int n);
```

scmdqalloci allocates the command queue cmdi to size n and sets up the queue indexing parameters. It returns 0 for no error, 1 for insufficient memory, 2 for no cmds, or 3 if a queue was already allocated. This function is called automatically by scmdstr2cmd, so there is no need to call it from externally.

```
int scmdaddcommand(cmdssptr cmds,char ch,double tmin,double tmax,double dt,double on,double off,double step,double multiplier,const char *commandstring);
```

This is like scmdstr2cmd but works from separate values rather than a string of parameters. cmds is the command superstructure and ch is the command code character (see scmdstr2cmd description). tmin, tmax, and dt are the simulation start, stop, and stepping times. on, off, step, and multiplier are the command timing parameters, for when the command should turn on, when it should turn off, its linear timestep, and its exponential multiplier. Finally, commandstring is the actual string of the command. Not all parameters are used for all command types.

The function can return any of several error codes: 0 is no error, 1 is memory allocation failed, 2 is `cmds` was set to NULL, 5 is command time step (step) was set  $\leq 0$ , 6 is the command timing type character (ch) was not one of those recognized, 7 is a failure to insert the command in the command queue because memory could not be allocated for either a new queue or a larger queue, and 8 is command time multiplier ( $xt$ ) is  $\leq 1$ .

`int scmdstr2cmd(cmdssptr cmds, char *line2, double tmin, double tmax, double dt);`  
 This takes in a string in `line2`, parses it for a command type, timing, and command string, creates a new command for it, and adds it to the proper command queue. The queue is automatically created or expanded if needed. For the command timing of the floating point types (b, a, @, and i), this routine also needs to know the simulation start, stop, and time step parameters given in `tmin`, `tmax`, and `dt`. The format of `line2` needs to have one of the following forms:

<u>code</u>	<u>parameters</u>	<u>execution timing</u>
<u>continuous time queue</u>		
b		runs once, before simulation starts
a		runs once, after simulation ends
@	<i>time</i>	runs once, at $\geq time$
i	<i>on off dt</i>	runs every <i>dt</i> , from $\geq on$ until $\leq off$
x	<i>on off dt xt</i>	geometric progression
<u>integer queue</u>		
B		runs once, before simulation starts
A		runs once, after simulation ends
&	<i>i</i>	runs once, at iteration <i>i</i>
I, j	<i>oni offi dti</i>	runs every <i>dti</i> iteration, from $\geq oni$ to $\leq offi$
E, e		run every time step
N, n	<i>n</i>	runs every <i>n</i> time steps

The function can return any of several error codes: 0 is no error, 1 is memory allocation failed, 2 is `cmds` was set to NULL, 3 is error in `line2` format, 4 is command string is missing from `line2`, 5 is command time step was set  $\leq 0$ , 6 is the command timing type character was not one of those recognized, 7 is a failure to insert the command in the command queue because memory could not be allocated for either a new queue or a larger queue, and 8 is command time multiplier ( $xt$ ) is  $\leq 1$ . A change as of 6/24/04 is that the timing types e and n are now exact because they use integer arithmetic rather than floating point arithmetic; this is most useful for unequal length time steps.

`void scmdpop(cmdssptr cmds, double t);`  
`scmdpop` removes all commands from the regular queue that are for time `t` or before, without executing them. The routine can be used after the simulation to avoid executing simulation time commands after an early exit from the simulation loop. It does not do anything to commands in the integer queue.

`enum CMDcode scmdexecute(cmdssptr cmds, double time, double simdt, Q_LONGLONG iter, int donow);`  
 scmdexecute removes and executes all commands from the command queues that have times that are less than or equal to time for the floating point queue, or iteration counters less than or equal to iter for the integer queue. Integer queue commands are performed first. If iter is set to -1 or less, an internal iteration counter is used; this internal counter starts at 0 and is set to iter any time  $\text{iter} \geq 0$ . simdt is the simulation time step; it is used so that floating point queue commands are executed no more often than once per simulation time step. Commands that should be repeated in the future are put back in the proper queue with the execution time or iteration updated to the previous requested value plus the command time step and with the invoke member incremented. The return value codes are essentially the same as those that are returned from the command executing function given in cmdfn. It is the largest of the errors that are returned from cmdfn: CMDok, CMDpause, CMDstop, CMDwarn, or CMDabort. If the return value is CMDwarn, an error message is sent to stderr that says which command failed as well as what the error string contains if it was used. This function sends all commands to the command function listed in cmdfn. donow is a flag that produces normal operation, described above, when it equals 0; when it is 1, all remaining commands in the queue are executed immediately and are not put back in the queue.

`enum CMDcode scmdcmdtype(cmdssptr cmds, cmdptr cmd);`  
 Assuming command execution function, as well as the individual command functions, are set up for this purpose, this returns the type of command that cmd is. The calls the command execution function with line equal to the command word followed with the word "cmdtype". Commands are supposed to return the appropriate enumerated type. Possible return codes are CMDnone, CMDcontrol, CMDobserve, and CMDmanipulate.

`int scmdnextcmdtime(cmdssptr cmds, double time, Q_LONGLONG iter, enum CMDcode type, int equalok, double *timeptr, Q_LONGLONG *iterptr);`  
 Returns the execution time and iteration number of the first commands that are to be executed at or after time and iter, respectively. Only commands of type type are checked, where this type may be a single type (CMDcontrol, CMDobserve, or CMDmanipulate) or it can be a multiple type (CMDall, CMDcntlORobs). If only times or iterations after time or iter are wanted, set equalok to 0; if equality is ok too, set equalok to 1. If they are not NULL, results are returned in timeptr and iterptr. The function returns 0 if there are no commands to be executed after time or iter, 1 if there are only integer queue commands, 2 if there are only continuous time queue commands, and 3 if there are both commands. This function has not been tested yet.

`void scmdoutput(cmdssptr cmds);`  
 scmdoutput displays the output files, data arrays, the queue of commands, and the command timing parameters to stdout.

`void writecommands(cmdssptr cmds, FILE *fptr, char *filename);`

This prints the contents of the command superstructure and the individual commands to `fptr` using a format that can be read in by Smoldyn. Several items are not written to the file: the `iter` counter and the `root` string in the superstructure, and the `invoke` and internal data of individual commands. `filename` is optional; if it is included, the printed line for `output_files` will not include any file called `filename`. This way, one can run a saved simulation file without immediately overwriting the new configuration file.

`void scmdsetflag(cmdssptr cmds, double flag);`

Sets the command superstructure flag value to `flag`.

`double scmdreadflag(cmdssptr cmds);`

Returns the command superstructure flag value.

`void scmdsetprecision(cmdssptr cmds, int precision);`

Sets the precision. Use a negative number for automatic and a positive number for that precision.

`int scmdsetoutputformat(cmdssptr cmds, char *format);`

Sets the output format. Reads a string in `format`, which should be either “`ssv`” or “`csv`”. In the former case, this sets the `outformat` element to ‘`s`’ and in the latter case, sets `outformat` to ‘`c`’. Returns 0 for success and 1 for unrecognized string.

### Data functions

`int scmdsetdnames(cmdssptr cmds, char *str);`

This inputs a list of data names in `str` as words separated by spaces and creates a new empty data array for each one. This updates the `ndata`, `maxdata`, and `data` elements of the data structure as needed.

`void scmdappenddata(cmdptr cmd, int newrow, int nargs, ...);`

This appends data to the data element that is stored with the command, for observation commands. This also creates the data element if needed. Enter `newrow` as 1 to start a new row in the data table, `nargs` as the number of arguments being entered, and then that many arguments, all of which need to be doubles. This basically just calls `ListAppendItemsDDv`.

### File functions

`int scmdsetfroot(cmdssptr cmds, char *root);`

`scmdsetfroot` sets the file root element of the command superstructure to the string that is sent in. The function returns 0 for success or 1 if either `cmds` or `root` are `NULL`.

`int scmdsetfnames(cmdssptr cmds, char *str, int append);`

`scmdsetfnames` inputs a list of file names in `str` as words separated by spaces (if a file name has a space in it, this routine won’t recognize the name correctly). Set `append` to 0 if any existing file contents should be overwritten and to 1 if they

should be appended. It counts the number of names in the list, updates the `nfile` element of the command superstructure, allocates the `fname` and `fptr` lists as needed, and copies the names to `fname`. The files are not opened. The routine returns 0 for success, 1 for inability to allocate memory, 2 for a file name that could not be read, or 4 if `cmds` is NULL.

```
int scmdsetfsuffix(cmdssptr cmds, char *fname, int i);
```

`scmdsetfsuffix` sets the file suffix number for file name `fname` to equal the integer given in `i`. If the file name is not recognized, a 1 is returned to indicate an error; otherwise a 0 is returned. `scmdsetfnames` has to have been called first.

```
int scmdopenfiles(cmdssptr cmds, int overwrite);
```

`scmdopenfiles` opens any output files that are listed in the `nfile` and `fname` elements of the command superstructure. Any files that are open when this function is called are first closed. The complete file name that is opened for each file is the string in the root element of `cmds`, concatenated with the string in the `froot` element of `cmds`, concatenated with the `fname` string for the file. If the name in `fname` is “stdout” or “stderr” then the file pointer is set to point to `stdout` or `stderr`, respectively. If `overwrite` is non-zero, any prior file is simply overwritten; otherwise, this routine looks for existing files and asks the user if any existing files should be overwritten. The function returns 0 for success and 1 for failure, where failure might arise from the user saying that a file should not be overwritten or from the inability to open a file for writing. If a file could not be opened, an error message is displayed to `stderr`. Upon failure, structures are not freed.

```
int scmdoverwrite(cmdssptr cmds, char *line2);
```

`scmdoverwrite` reads the first word in `line2`, which is supposed to be a file name, looks for that name in the `fname` list of file names, closes the file, and reopens it. That way, the file is made empty for overwriting. The function stores the new file pointer in the `fptr` list of `cmds`. It returns 0 for success or 1 for file not found, or 2 for failure to open for writing.

```
int scmdincfile(cmdssptr cmds, char *line2);
```

`scmdincfile` reads the first word in `line2`, which is supposed to be a file name, looks for that name in the `fname` list of file names, closes the file, increments the file name by one, and opens the new file. The function returns 0 for success, 1 for file not found, and 2 for failure to open new file for writing. The new file pointer is stored in the `fptr` list of `cmds`. This is useful for creating file stacks.

```
int scmdgetfptr(cmdssptr cmds, char *line2, int outstyle, FILE **fptrptr, int *dataidptr);
```

This is a utility routine for use by commands that save data to files and/or tabular data tables. It reads the first one or two words from `line2` and determines if they are file names or data table names by looking them up in the `fname` and `dname` lists in the command superstructure. If they are, then the corresponding file pointer and/or data ID number are returned in `fptrptr` and `dataidptr`. Send in `outstyle` as 1 if only a file name is allowed, 2 if only a data table name is allowed, or 3 if both are allowed;

only one word is read from line2 in the former two cases and two words are read from line2 in the latter case. Returns the number of parsed objects, which can be 0, 1, or 2, or returns -1 if the first word is neither a filename nor a data table name.

**int** scmdfprintf(cmdssptr cmds, FILE \*fptr, **const char** \*format, ...);

Output commands should use this function for printing output rather than the stdio fprintf function. The reason is that this one edits the format string to reflect the number of significant figures that were requested by the user and that are stored in cmdss->sigfig. This also allows for either comma or space delimiters between entries. If fptr is NULL, then this simply returns without doing anything. Returns the number of characters printed, or a negative value for an error.

**void** scmdflush(FILE \*fptr);

This simply flushes the file.

### **Possible improvements**

The command string is fixed at 256 characters, which could be too short for some commands. In particular, a reasonable command might be “multicommand” whose arguments are a list of commands that should be run sequentially. This would allow a block structured command language.