Documentation for SimCommand.h and SimCommand.c

Steven Andrews, © 2004-2021

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.

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 {
  struct cmdsuperstruct *cmds; // owning command superstructure
  char timing;          // timing character (e.g. B, A, i, @, x)
  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 {
   int condition;          // 0=new, 1=update time, 2=new command, 3=ready
   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 *simvd;              // void pointer to simulation structure
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;
```

Data structures

The structure `cmdstruct`, pointed to with `cmdptr`, contains the information for one command. `cmds` is a pointer to the superstructure pointer that owns this command. `twin` is a pointer either from the command template to the command in the queue, or vice versa from the command in the queue to the template. `timing` is a single character that represents the command timing rules, of which the options are listed below. `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. `oni`, `offi`, and `dti` are used instead of `on`, `off`, and `dt` for commands in the integer queue; here, for example, a command is run every `dti` iterations. `invoke` depends on whether the command is a template or in a queue; for the tempate, it is 0 if the command has not been put in a queue and for a command in a queue, it is the number of times that the command has been invoked so far (it equals one during the first command call).

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` ≤ 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+xt`^{2}`dt`, ..., `off`.

`str` is the command string. 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.

The following elements only apply for commands in a queue and are not used by templates at all. `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 lists of runtime commands, the address of a function that is supposed to execute them, and information about the output files and data arrays.

condition is an integer that tells the status of the superstructure. 0 indicates that the data structure is new and/or that it should be fully reinitialized before starting a new simulation. 1 indicates that a simulation is ongoing, but that the time step changed, so all commands in the queue need time updates; also, there may be new commands in the template list that should be added to the queues. 2 indicates that there are new commands in the template list that need to get added to the queues, but that other times don't need updating. 3 indicates that everything is ready to go.

cmdlist is the list of templates for all commands. It has allocated size maxcmdlist and used size ncmdlist. These commands are unsorted and retained even if the simulation is over. These commands are not run from here but are only archived here for copying later into one of the two queues.

cmd is the floating-point (regular) queue of commands, sorted in order of their next exection times. cmdi is the integer 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 (set to docommand for Smoldyn, which is in smolcmd.c). It is sent the argument simvd (void* pointer for the simulation structure), 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. fappend is set to 0 for each file that should be overwritten and 1 for files that should be appended. Output is printed to the file with precision significant figures. outformat can be either 's' or 'c' to indicate space or comma delimited columns.

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 simvd 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.

Control flow

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.

Command timing

Command timing is encoded in a single character. The list is as follows. The parameters are the values that the user inputs.

code	parameters	execution timing
?		timing hasn't been set yet
continuous time queue		
b		runs once, before simulation starts
a		runs once, after simulation ends
@	*time*	runs once, at ≥ *time*
i	*on off dt*	runs every *dt*, from ≥ *on* until ≤ *off*
x	*on off dt xt*	geometric progression
integer queue		
B		runs once, before simulation starts
A		runs once, after simulation ends
&	*i*	runs once, at iteration *i*
I,j	*oni offi dti*	runs every *dti* iteration, from ≥ *oni* to ≤ *offi*
E,e		run every time step
N,n	*n*	runs every *n* time steps

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).

4/13/21 Added `cmdlist` to the command superstructure and `timing` to commands. Also added twin and timing to commands and condition to the superstructure. Substantial code overhaul.

5/17/21 Renamed `cmdfnarg` element of `cmdsuperstruct` to `simvd`. Also, included it in the `SCMDPRINTF` macro, so that simulation flags are accounted for during output. Also, exposed `scmdalloc` and `scmdfree` to header file.

Utility functions

Internal: `void scmdcatfname(cmdssptr cmds,int fid,char *str);`
This 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`.

Internal: `void scmdcopycommand(cmdptr cmdfrom,cmdptr cmdto);`
Copies most elements of `cmdfrom` to `cmdto`. This does not copy `i1`, `i2`, `i3`, `f1`, `f2`, `f3`, `v1`, `v2`, `v3` values or `freefn`, all of which it sets to either 0 or `NULL`. It sets `invoke` to 0 and puts an empty string into `erstr`. This function sets the `twin` element of `cmdto` to point to `cmdfrom` but it does not change the twin element of `cmdfrom`. This function is used by `scmdupdatecommands` for copying commands from their templates to the command queues.

Internal: `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.

Memory management

Internal: `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.

Internal: `void scmdfree(cmdptr cmd);`
`scmdfree` frees a command structure.

`cmdssptr scmdssalloc(int (*cmdfn)(void*,cmdptr,char*),void *simvd,char *root);`
`scmdssalloc` allocates a minimal command superstructure. `cmdfn` should be sent in pointing to a function that can execute the commands and `simvd` is the first argument of that function. For example, in the Smoldyn program, the `cmdfn` is sent in as `&docommand` and `simvd` 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);`
`scmdfree` frees a command superstructure and all internal elements except for `cmdfn` and `simvd`.

Internal: `int scmdcmdlistalloc(cmdssptr cmds,int newspaces);`
Expands the `cmdlist` element of the command superstructure, where the expansion amount is `newspaces` (which needs to be >0). This creates the initial list if needed. Returns 0 for success or 1 for out of memory.

Internal: `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 by `scmdupdatecomamnds`.

Internal: `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 by `scmdupdatecommands`.

`void scmdsetcondition(cmdssptr cmds,int cond,int upgrade);`

Sets the command superstructure `condition` parameter, based on `cond` and `upgrade`. If `upgrade` is 0, then this makes the condition no greater than `cond`, downgrading it if needed but not upgrading it. If `upgrade` is 1, this makes the condition no less than `cond`, upgrading it if needed but not downgrading it. If `upgrade` is 2, then this simply sets the condition to `cond`.

`int scmdaddcommand(cmdssptr cmds,char timing,double on,double off,double step,double multiplier,const char *commandstring);`

This adds a command into the `cmdlist` element (for templates) of the command superstructure with minimal error checking. `cmds` is the command superstructure and `timing` is the command code character (see `scmdstr2cmd` description). `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 returns: 0 for no error, 1 if memory allocation failed, 2 is `cmds` was set to `NULL`, 3 if the `commandstring` is missing, or 6 if the timing type wasn't recognized. This downgrades the superstructure condition to 2 (or leaves it lower) to indicate that this command needs to be added to a queue.

`int scmdstr2cmd(cmdssptr cmds,char *line2,char **varnames,double *varvalues,int nvar);`

This takes in a string in `line2`, parses it for a command type, timing, and command string, and then sends those parameters off to `scmdaddcommand` to actually add the command. The format of `line2` needs to have one of the following forms:

code	parameters	execution timing
continuous time queue		
`b`		runs once, before simulation starts
`a`		runs once, after simulation ends
`@`	*time*	runs once, at ≥ *time*
`i`	*on off dt*	runs every *dt*, from ≥ *on* until ≤ *off*
`x`	*on off dt xt*	geometric progression

integer queue		
B		runs once, before simulation starts
A		runs once, after simulation ends
&	*i*	runs once, at iteration *i*
I,j	*oni offi dti*	runs every *dti* iteration, from ≥ *oni* to ≤ *offi*
E,e		run every time step
N,n	*n*	runs every *n* time steps

Returns 0 for success, 1 for failed memory allocation, 2 is `cmds` was set to `NULL`, 3 is `line2` missing or error in `line2` format or command string is missing from `line2`, 6 if the command timing type character was not one of those recognized.

Internal: `void scmddocommandtiming(cmdptr cmd,double tmin,double tmax,double dt)`
This inputs a pointer to single command that is already in a command queue, in `cmd`, and the simulation timing parameters `tmin`, `tmax`, and `dt`. This either computes or recomputes the command timing parameters. It only works with timing parameters for the relevant queue, leaving the others untouched. It also only modifies the timing parameters that weren't set up by `scmdaddcommand`.

`int scmdupdatecommands(cmdssptr cmds,double tmin,double tmax,double dt);`
As necessary, this copies commands from the template list to the queues, and updates the command timing information. Run this as often as desired, including right before starting a simulation. Returns 0 for success, 1 for failure to allocate memory, or 6 for unknown timing character (a bug). If `tmax` < `tmin` or `dt` ≤ 0, this doesn't do anything but just returns 0, meaning that it's done all it can with the available information, which is nothing.

`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 `iter` ≥ 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.

```
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, taking the commands from the command list, not from the queues. Items not written to the file are: 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.

```
double scmdsetcondition(cmdssptr cmds,int condition);
```
Sets the superstructure condition value to the number given here. The system is that it's 0 if the commands are not in the queues yet, or if the ones in there are out of date (e.g. the time step was changed but commands have not been re-enqueued yet), and 1 if the commands are in the queue and are ready to go.

```
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 narg, ...);
```

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, narg 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 `dataitptr`. 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.

Also, the setflag function could be made a lot more powerful using a broadcast message function.