A Subsystem Utilities Package for UNIX

Ken Raeburn

Printed: January 16, 1990

Copyright ©1987 MIT Student Information Processing Board

Contents

1 Why? 3

2 Command Table Source 3

2.1 Standard Requests 4

3 Command Table Compiler 5

4 Runtime Usage 5

5 Compilation-time Usage 8

6 User Interface 9

7 Intended Changes 9

1 Why?

put some explanation of purpose here

2 Command Table Source

The start of the source file for the command table compiler must indicate the command table name, with the command:

`command_table` *table_name;*

This is the name of the table that will be used for external references. After the table name declaration, a list of subsystem requests should be provided, of the form:

`request` *subr_name, description of command, request_name* {*, request_name*}*;*

That is, the keyword "request," the name of the subroutine to be invoked, a comma, a short (one-line) description of the command (which will be displayed with the "list_requests" request), another comma, and a list of comma-separated names by which the requst can be invoked. Any information in an entry which consists of non-alphanumeric characters (particularly the description because it contains spaces, or special characters such as "?" in a request_name) must be quoted. The entry is terminated by a semicolon.

One special form of entry can also be used for routines which are to be listed in the command table, but have not yet been written. (This should not be used in the production version of programs, but may be useful during the writing of them.) The "request" keyword is replaced by the keywork "unimplemented," and the subroutine name (and the following comma) is omitted, thus:

```
unimplemented   delete_all_files,
                ``Delete all my files.'',
                delete_all_files, daf;
```

When this request is called, a routine in the subsystem library will inform the user that the request is not currently implemented.

There is also an optional "options" field, which appears as the last request_name and should be a parenthesized list of (comma-separated) options. At present, the only options defined are "dont_list," which indicates that the command should not be listed by the "list_requests" request, and "dont_summarize," which has a similar effect on the "summarize_requests" routine.

After all entries in the table have been specified, the end of the table must be specified with the "end" keyword.

2.1 Standard Requests

The following table is supplied with the subsystem library, and uses some of the routines supplied in the library. (Many of these have not yet been implemented.)

```
command_table  ss_std_requests;

request ss_self_identify, ``Identify the subsystem.'',
        ``.'',
        (dont_list, dont_summarize);

request ss_help, ``Display info on command or topic.'',
        help;

unimplemented
        ss_list_help,
        ``List topics for which help is available.'',
        list_help, lh;

request ss_list_requests, ``List available commands.'',
        list_requests, lr, ``?'';

request ss_quit, ``Leave the subsystem.'',
        quit, q;

unimplemented
        ss_abbrev,
  ``Enable/disable abbreviation processing of request lines.'',
        abbrev, ab;

unimplemented
        ss_execute,
        ``Execute a UNIX command line.'',
        execute, e;

unimplemented
        ss_summarize_requests,
     ``Produce a list of the most commonly used requests.'',
        ``?'';

request ss_subsystem_name,
        ``Return the name of this subsystem.'',
        subsystem_name,
        (dont_list);
```

```
request ss_subsystem_version,
        ''Return the version of this subsystem.'',
        subsystem_version,
        (dont_list);

end;
```

The "." request, `ss_self_identify`, prints the name and version of the currently running subsystem. The `ss_help` routine searches for help files in a series of directories provided by the program. The abbreviations processor has not yet been implemented, but it will allow the entering of abbreviations bythe user, similar to the "alias" facility of `csh`. Since `ss_summarize_requests` has also not been implemented, the "?" command currently calls the same routine as "list_requests".

3 Command Table Compiler

The command table is converted to a usable form by the program `make_commands`. It takes one mandatory argument, the name of the file which contains the command table. This filename should end with the suffix *.ct*; the compiler will not work if it does not. (`Make_commands` also takes one optional argument after the filename, `-n`, which will cause it to retain its temporary files rather than deleting them.)

One output file is produced, with a suffix of *.o*; it contains one object, the command table, and references to all the procedures listed in the table.[1] This table is of type `ss_request_table`, which is defined in the include file *ss.h*.

4 Runtime Usage

The only handle the programmer needs on the data structure manipulated by the subsystem library is the index number of the subsystem control info structure (sci_idx) manipulated. (It is conceivable that, in some cases, a programmer might wish to have more than one subsystem available at once, with the ability to switch from one to another temporarily.) This index is used by most of the routines in the subsystem library, and is supplied to the application program at the "creation" of an invocation of the subsystem, which is done with the following routine:

[1] This file is compiled from an intermediate C source file with options `-O -R` to put the command table into the shared (text) portion of the resulting executable, and run through `ld` to eliminate extra symbol table entries.

```
int ss_create_invocation(subsystem_name, version, info_ptr,
                                request_table_ptr, code_ptr)
        char *subsystem_name, *version;
        char *info_ptr;
        ss_request_table *request_table_ptr;
        int *code_ptr;
```

If no errors occurred, the returned value is the index of the new subsystem, and `*code_ptr` is set to zero. If `*code_ptr` is nonzero, it is an error code which may be examined with the `com_err` library.

When the subsystem is initially created, only the specified request table is in its search list. Others can be added or deleted using the routine:

```
ss_add_request_table(sci_idx, request_table_ptr, position, code_ptr)
ss_delete_request_table(sci_idx, request_table_ptr, code_ptr)
        int sci_idx;
        ss_request_table *request_table_ptr;
        int position;
        int *code_ptr;
```

Position can be used to specify the order of searching the tables. A large number (e.g., 9999) will insure that the table being added is at the end of the search sequence.

Directories containing help files can be added to or deleted from the internal list searched by using `ss_add_info_dir` and `ss_delete_info_dir`:

```
void ss_add_info_dir(sci_idx, info_dir, code_ptr)
void ss_delete_info_dir(sci_idx, info_dir, code_ptr)
        int sci_idx;
        char *info_dir;
        int *code_ptr;
```

Initially, the input prompt for the subsystem will be the subsystem name followed by a colon and two spaces, thus:

```
        subsystem:  _
```

The prompt can be examined with `ss_get_prompt` and changed with `ss_set_prompt`:

```
ss_set_prompt(sci_idx, new_prompt)
char *ss_get_prompt(sci_idx)
        char *new_prompt;
```

The input/parse/dispatch loop is managed by the subroutine `ss_listen`:

```
void ss_listen(sci_idx, code_ptr)
        int sci_idx;
        int *code_ptr;
```

The subroutines called (listed in the command tables) should be written to the following declaration:

```
int subroutine(argc, argv, sci_idx)
        int argc;
        char **argv;
        int sci_idx;
```

Until better argument parsing and interpretation is planned, the subsystem library will do no more than parse the command line into a list of arguments (as would be supplied to `main()`). The argument `sci_idx` is supplied last so that it may be omitted when only one subsystem is involved in a program.

Command lines can be executed not only from the input loop, but also from the program itself. The subroutines `ss_execute_command` and `ss_execute_line` will execute the commands supplied as if they had been typed to the input loop.

```
int ss_execute_command(sci_idx, argv)
void ss_execute_line(sci_idx, line_ptr, code_ptr)
        int sci_idx;
        char **argv;
        char *line_ptr;
        int *code_ptr;
```

The error code from `ss_execute_command` is given as the return value. In a future version of this library, probably many, if not most, of the routines will return error codes in this fashion.

To terminate the input loop, the user can enter a request which runs the `ss_quit` procedure, or the program can call `ss_abort_subsystem`:

```
void ss_abort_subsystem(sci_idx, code)
        int sci_idx;
        int code;
```

When control returns to `ss_listen`, it will return to its caller with the error code supplied to `ss_abort_subsystem`. (The subroutine `ss_quit` simply executes a call to `ss_abort_subsystem`.)

While running within a subsystem request, the routines `ss_current_request` and `ss_name` will return information about the request. The `ss_current_request` function returns the primary name of the request being executed (the first one listed in the request table entry), or NULL if none is being executed. This string is internal information, and should not be modified. The routine `ss_name` returns a string naming the subsystem and the request, if any. The returned string is of the form "subsystem" or "subsystem (request)". The string returned is allocated and should be freed when it is no longer needed.

```
char *ss_current_request(sci_idx, code_ptr)
char *ss_name(sci_idx)
```

If output is being generated that may be more than one screenful for the user, it may be output through a file descriptor generated by the routine `ss_pager_create()`. This routine takes no arguments, and returns an integer which is either -1 (some error occurred) or a file descriptor number. Currently, it generates output through "more", and a call to `wait()` should be made after the file descriptor is closed, but this should be changed at some point to make the output routines internal; much nore control can be gained this way.

If errors are encountered during the execution of the subsystem, they should be reported through the routine `ss_perror`. It is written to work with the `com_err` library, and can therefore be used to report errors returned from the UNIX operating system, from the subsystem library, and from any other package written to use the `com_err` library. [2]

```
ss_perror(sci_idx, code, message)
        int code;
        char *message;
```

5 Compilation-time Usage

To generate the object file, run the program `make_commands`:

```
% make_commands commands
```

This will generate *commands.o* from *commands.ct*. This file and `libss.a` should be linked with your object files.

If you are using a makefile, you can add the following lines to it to have command-table object files automatically rebuilt:

[2] At the moment,it isn't written to use the `com_err()` routine; it uses `error_message()` and prints a message in a format similar to `com_err()`. It will be rewritten to use `com_err()`, so that the hooks for catching the output with other procedures will work.

```
# rule for rebuilding .o files from command table sources
.SUFFIXES:      ct
.ct.o:
        make_commands $*.ct
```

6 User Interface

The user interface of this library is currently fairly weak, but reliable. Providing alternatives to the (only) existing one is a major intended upcoming change to this library. Currently, the user sees a prompt string, must type a command and its arguments and hit return, and the command will be executed if it is valid. While this is sufficient for some, others would prefer single-key commands, or completion of commands and arguments; many would like to see an interface based on the X window system.

7 Intended Changes

This section is sort of fuzzy, but some of the main inteded changes are:

Revise calling sequence for some routines (at least the ones returning "void") so that error codes are handled as return values; this is generally more efficient.

Build alternative interfaces: Something based on curses. Something based on X, once version 11 and its toolkits become stabilized.

Get abbreviations working.

Do some argument parsing/interpretation/validation. Some of the problems with programs written today (and subroutines written for this library today) is that it is very easy to make minor errors which cause the argument parsing to work incorrectly. This should probably be done as a separate but integrable module, so that commands can use their (UNIX-standard) command-line arguments (via argc/argv) this way also.

Write info files for the subroutines supplied. Maybe even a man page for the library.

Completion of commands and maybe even arguments.