CHARA TECHNICAL REPORT

NO. 81 15 SEPTEMBER 1998

CHARA Array User Interface: Progammer's Manual

T.A. TEN BRUMMELAAR (CHARA)

1. INTRODUCTION

Any control system needs a user interface and the CHARA Array is no exception. This document describes the first version of the user interface to be used on all system controllers within the CHARA Array system. While GUI type interfaces are very popular, and normally considered 'a must' for any modern system, due to severe time constraints we will be using a text-based system already fully developed. We will move towards a GUI system in the 'fullness of time'.

The system to be used is based on the user interface used at SUSI which in turn is based on a commercial system developed by the CHIP software company in Australia (now trading under the name CHILLI). It is a text-based system, and while best run within an xterm(1) window[2], can also run on any text terminal. It relies on the ncurses(3) package for screen control and provides mouse, menu, socket and command line control.

While the primary user interface is text only, if it is run within an xterm, programmers can use the simpleX(3) library to perform graphics operations on the X-windows screen. Other general purpose libraries are available including the numerical recipes library for math, the rwfits(3) library for FITS file manipulation and the filter(3) library for implementing digital filters. These libraries will not , however, be documented in this report.

An example application program is listed in Appendix A.

2. WINDOWING

It is assumed that the user is familiar with the ncurses(3) screen manipulation library. Separate documentation is available on this package within the control system software tree. Basically, ncurses provides an abstraction of the screen in memory where a programmer can

[1]Center for High Angular Resolution Astronomy, Georgia State University, Atlanta GA 30303-3083
TEL: (404) 651-2932, FAX: (404) 651-1389, FTP: ftp.chara.gsu.edu, WWW: http://www.chara.gsu.edu.
Funding for the CHARA Array is provided by the National Science Foundation, the W. M. Keck Foundation, the David and Lucile Packard Foundation and by Georgia State University.

[2]Note that sometimes things go wrong in an xterm, you must have the environment variable TERMINFO defined or curses will not know how to work in an xterm. This need not be set to anything, it just has to exist in the environment

create new windows, write to them, get data and so on. The screen itself does not change until the programmer issues a "refresh" command, at which point ncurses works out the optimum actions required to make the screen look the way it should. Operations can be performed on any number of windows on the screen, and the system removes much of the tedium of screen control. Ncurses also provides mouse and input stream functionality, although the mouse data is only available when running within an xterm(1).

2.1. Layout and predefined windows

The user interface breaks the screen up, vertically, into four basic areas: the status area, the main working area, the system area, and the command line. Each of these has at least one predefined global window.

The status area takes up the top 9 lines of the screen and can be used for any purpose, most often displaying status information. Frequently, these data are created and displayed using background processes (see Section 4). Only one window is defined within the status area called (imaginatively enough) `status_window`. This window is automatically refreshed once per second by the background control system and need not be refreshed by the programmer, unless you desire a faster refresh rate.

The main working area is where the menu system is placed and where most application routines put their text, edit screens and so on. The main working area is contained within a box and is 10 lines long. There are three global windows defined within the main working area. First of all there is `main_window` which can be used at any time and fills the entire working area. The other two windows together also fill the entire main working area: `heading_window` takes up the top two lines and is used for titles and headings, and `sub_main_window` occupies the rest of the main working area.

The system area takes up the next two lines and is used to display system messages. The first line is by default highlighted. The system area is covered by the `system_window`.

The last line of the screen is the command line and is not normally used by application programs.

Many functions are available for writing to the various windows, in fact unless you are creating special displays it is rarely necessary to use raw ncurses to display things in any of the global windows. For example, the function call

```
heading(heading_window,"First Line","Second line");
```

places the text `First Line` centered and highlighted in the first line of `heading_window` and the text `Second line` in the second line. Consult the manual pages and the program in Appendix A for more examples of the use of these windows and function calls. Like most libraries of this type, there is probably already a function written to do whatever it is you want to do; try to use them when you can.

Of course, the programmer is at liberty to create new windows and overlay them on the global windows at any time. Care should be taken to remove them again when you are finished with them. It is also possible to get control of the entire screen and use ordinary `printf` statements. You can grab complete control of the screen using the function call

```
plain_screen_on();
```

and release it again with

```
plain_screen_off();
```

This is not often required and is not recommended. One example of the use of this function is the standard serial port function `asccom`, which provides a terminal like connection to a serial port.

2.2. The Active Window

The user interface has many other windows (such as, for example, the error message window), and these may pop up on the screen unexpectedly. This happens most often when an error occurs in a background process. When this pop-up window is removed the user interface needs to know which window to refresh so that the screen can be returned to its original state. This is known as the `active_window`, a global variable that is a pointer to a window. The user interface will, by default, refresh the active window whenever it interferes with the screen. It is the programmer's responsibility to ensure that the active window is set to the correct value. For example

```
active_window = main_window;
werase(main_window);
mvwaddtrs(main_window,0,5,"Here we are in the main window);
.
.
.
wrefresh(main_window);
```

If for some reason the user interface places a new window over the main window it will know that it must refresh the main window when it is done. It should be standard practice to set the active window before writing to any predefined or user defined screen area.

3. ERROR MESSAGES

All programs have error conditions, especially those that control hardware, and these errors need to be reported to the user. Errors are sent to the user via the obviously named function `error()`. This function works very much like a `printf()` function, but places the text in a new window, created by the `error()` function itself. An example of the use of the function is

```
if (there_is_an_error)
{
        return error(ERROR,"Something has gone wrong.");
}
```

and other examples of the use of the `error()` call can be found in Appendix A.

The `error()` function's first argument is an error level, which is returned by the function. The possible error conditions are

- `NOERROR` — No error has occurred, normally considered a good thing.
- `MESSAGE` — No error has occurred, but we want to tell the user something anyway.
- `WARNING` — An error has occurred, although it probably isn't serious.
- `ERROR` — A serious error has occurred.
- `FATAL` — A fatal error has occurred. The program is stopped altogether.

Note that in the example above, and in most cases in practice, the error causes the function to exit, returning the error level. This is not always the case but in many situations it is the appropriate behavior. Any user callable function, or background job, must return an error level, usually `NOERROR`. This is so the user interface can react to any errors. For example, if a background process returns an error level other than `NOERROR`, background processing is turned off to ensure that the error is not repeated.

4. BACKGROUND PROCESSING

The user interface provides a very simple background processing system. Background jobs are normally used to interrogate hardware and place status information on the status screen but, as long as they require no user interaction, can perform any task. Examples of background tasks are given in Appendix A. A declaration of a background task will always take the form

```
int i_run_in_the_background(void);
```

Thus a background job can take no arguments, but always returns an integer, the error level, to the background control system. A background job is added to the list of jobs using the function

```
background_add(i_run_in_the_background);
```

and removed using the function

```
background_del(i_run_in_the_background);
```

The background system is not a multi-stream system and is quite simpleminded. Whenever the system is waiting for the user to type a key, or use the mouse, via the function `get_command()`, it will run each background task in turn. For this reason it is important that any background task be short and fast, otherwise keyboard response time can be affected. So, if you wish to have a long and involved process run in the background (which is not recommended), you need to break it up into several smaller, and faster, background tasks.

A second implication of the way background processes works is that all background processing can be stopped by a user callable function. For example, a function that uploads a large file and does not get any data from the keyboard never calls the function `get_command()` and therefore never allows the background tasks to run. If you write a function such as this it is a good idea to manually call the background tasks every now and then using the call

```
background();
```

which will run the next background job in the queue and return. In this way it is possible to continue displaying status information while performing long tasks.

It is not always appropriate to have the background jobs running. For example, you may require a window so large that it covers the status window and you would not want status information to overwrite it every second. Background processing can be turned off using the call

```
background_off(0,NULL);
```

and started again with the call

```
background_on(0,NULL);
```

where the arguments `(0,NULL)` are included because these are user callable functions, and like the normal C function `main(int argc, char **argv)` accept command line arguments. Since we do not a need a command line here we set the number of arguments, the first parameter, to zero.

These functions are similar to the other user callable functions `background_start(0,NULL)` and `background_stop(0,NULL)`, but not the same. While they may seem redundant they have slightly different uses. Background processing can be started only once, while it can be turned off and back on again many times. Also, once the background processing has been stopped it can not be turned on again, although it can be restarted. In order to avoid confusion, it is best to use the on and off commands within a function while starting and stopping is up to the user.

5. STARTING THE USER INTERFACE

There are only a few things that need to be done in order to get the user interface up and running:

1. Get any arguments from the command line, normally a socket port to use.
2. Put up a title page. For this the functions `ui_clear_screen()`, `center_line()`, `put_line()`, and `wait_for_title()` can be very useful.
3. Initialize the global string pointer `TITLE` to point at a string to be used, as you may have guessed, as a title for the program.
4. Call the user interface initializer function

   `initialise_ui("menu.ini", "help.ini", port);`, where the first argument is the file containing the menu setup (see Section 6.1), the second argument is the help initialization file (see Section 8) and the final argument is a socket port number to use for outside commands (see Section 12).
5. Add the background tasks to the background processing list using the function

   `background_add()`.

6. Start up the user interface with the call `start_ui()`.

It is also common to add an `exit(0)` statement at the end of the `main()` function. While this is redundant (you can never return from `start_ui`), it avoids some compiler warning messages. Assuming that no errors are found in the help or menu initialization files the user interface will now be up and running. Note that the background processing is, by default, not on and needs to be turned on either by the user, or by the autolist system (see Section 6.1.1).

6. ADDING A NEW COMMAND

Each command can be looked upon as a stand alone C program which can display it's output in an ncurses window, or within it's own X window. Other than the fact that it will not be called `main()`, the declaration of a user callable function is the same, that is

```
int new_function(int argc, char **argv);
```

where `argc` is an integer representing the number of command line arguments, including the name of the function itself as argument 0, and `argv` is an array of strings, each containing one of the command line arguments. As discussed in Section 3, the return value is an error level, and hopefully `NOERROR`. The source file should include the header file `charaui.h` and be linked to the rest of the system.

Example user callable functions are given in Appendix A. Normally, the first thing is to either allow, or not allow, socket control of the function (see Section 12), then set the active window, check and analyze the command line arguments and, finally, perform whatever task is required.

In order to connect the new command with the rest of the system, a command line name must be associated with the function. Thus, if the user types this string on the command line the function will be invoked, and the menu initialization file has a useful name for the function as well. This command line name is set up by editing the file `functs.c`, an example of which is given in Appendix B. This is an array of structures, each containing a string defining the command line name, and a function pointer to the actual function. This is all that is required to add a new function to the system. More than one entry for the same function is allowed so you can set up aliases. Furthermore, note that there are a large number of predefined functions available and it is normal to leave these in any user interface.

6.1. The menu initialization file

Having edited the function lookup table in `functs.c`, the new function is now accessible from the command line, but not from the menu system. In order to add it to the menu system you need to edit the menu initialization file `menus.ini` (which is only a default name by the way; you can call this file anything you want really), an example of which is given in Appendix C.

Each menu defined in this file begins with a statement of the menu's name:

```
MENU MenuName
```

The menu name must be a single unique word, and there must be one menu named `MAIN`. The parser code of the menu initialization file is not sensitive to case. The menu name definition is followed by up to ten lines, each representing a menu item. For example,

```
asccom      Communicate with a serial port
```

The first word of each menu item definition is either a function or a menu name, while the remaining text will appear on the screen and should explain the menu item to the user. When invoked by the user, a menu item will either move you into a new menu, if it represents a menu name, or run a function.

With this system, a user is free to re-arrange the menu structure at will without the need to recompile the whole program.

6.1.1. The Autolist

Apart from menu definitions, the menu initialization file also contains the definition, if any, of the so-called `autolist`. The autolist is a list of function calls, including arguments and possible `goto` statements, which can be run as a single command by the user. Autolists are most frequently used in the hardware initialization phase of a control program, that is, at 'boot-up' time.

The autolist definition does not have to be included, but if it is must be the last thing in the menu initialization file and must begin with the keyword `AUTOLIST`. Each line after that is either a goto[3] statement, or a line of text you would normally type at a command line. If it is a goto, the label is the command line name of a function elsewhere in the list and the user will be given the option of moving to that command or continuing.

7. CREATING SCRIPTS

A user will often run the same series of commands many times, which can get dull if they are forced to continually type the same commands over and over. Therefore, a simple scripting language has been made to create new commands at run time. The command to read a new script is

```
script file
```

where `file` is the name of the file containing the script. If no extension is there it assumed to be `.scr`. The new command line name for the script will be the same as the file name. Since scripts are made at run time they can not be part of the menu hierarchy and must be invoked from the command line. Scripts can be created within the autolist but may not be used within the autolist.

Like the autolist, a script consists of a series of commands, one per line. Any text after the character '#' will be considered a comment and ignored. Each command can have any or all arguments exactly as you would type them on the command line. There are some special commands unique to scripts as well:

[3] Yes I know, gotos are not 'elegant' but neither am I.

- `label:` - Mark this position and call it `label`.
- `onmessage label` - If any function called after this line returns `MESSAGE` go to the position in the file marked by `label`.
- `onnoerror label` - If any function called after this line returns `NOERROR` go to the position in the file marked by `label`.
- `onwarning label` - If any function called after this line returns `WARNING` go to the position in the file marked by `label`.
- `onerror label` - If any function called after this line returns `ERROR` go to the position in the file marked by `label`.
- `onfatal label` - If any function called after this line returns `FATAL` go to the position in the file marked by `label`. This is rarely used as a fatal error normally causes the program to crash.
- `onyes label` - If any function called after this line returns `YES` go to the position in the file marked by `label`.
- `goto label` Move to the position in the file marked by `label`.

There are also three user callable functions that come in very handy in scripts with the following standard command line names:

- `message text` - Displays the text using the call `message(system_window,text)`.
- `ask text` - Asks the user a question using the call `ask_yes_no(text,"")` and returns the result.
- `nothing` - Does nothing.
- `endscript` - An alias for `nothing`, often useful as a marker of the end of the script.

An example of a script file is given in appendix A.

8. ONLINE HELP

It is often said that online help systems are normally 90% accurate and only 10% useful. Nevertheless the CHARA user interface includes online help. Like the menuing system, online help is completely user configurable by editing text files. To associate a text file with help you need to edit the initialization file `help.ini` (like the menu initialization file this name is arbitrary). An example help initialization file is given in Appendix B.

Each line of the help initialization file contains a reference to a help file and a description of the help. Note that in the example file in Appendix B there are two lines for each help file, one with a lengthy description of the file, and one with the command line name that the file is about. In this way the user can select from a list of descriptive titles, or simply type

```
help command
```

where `command` is the name of a command. Furthermore, when invoke from the menu window `<?>` entry the menu system will look for a help file with the same name as its command name.

Help files themselves are simple text files with embedded commands, similar to `nroff` or `latex` commands. An example help file is given in Appendix C. The help text file formatting language is very simple, as it only has four commands:

- `.center` (Yes this was written in Australian) which will center the next line of text,
- `.paragraph` which will force a new paragraph,
- `.nl` which forces a new line, and
- `.tab` which forces a tab.

Any other text will be placed into the display structure (see Section 11).

9. USING THE MOUSE

Apart from computer luddites like myself many people like to use a pointing device when working with software. The user interface includes mouse support, directly from the ncurses package, and responds to mouse clicks in (hopefully) predictable ways. You can include mouse support in user callable functions too, as shown in the example code in Appendix A. Note that the mouse functionality will only work when the program is run inside an xterm. Be sure to put any mouse code in between

```
#ifdef __NCURSES_H
```

and

```
#endif
```

statements so that if you should compile the code on another system without mouse support it will still work. Mouse support is not part of the standard curses package and not all systems will have ncurses. All Linux systems do, so this should not be a problem for us.

A mouse click is viewed by the system as the same as a keyboard event. A call to the function `get_command()` will return the next key pressed or the macro `KEY_MOUSE` if the mouse has been used. A call to the function `getmouse(&mouse)` will then return information about the mouse event in the structure

```
        MEVENT  mouse;                  /* A mouse event. */
```

This structure contains a field `mouse.bstate` which will tell what kind of mouse event happened, for example `LEFT_CLICK` or `RIGHT_CLICK` and where the mouse was, in terms of text position on the screen, in the fields `mouse.x` and `mouse.y`. Refer to the ncurses manual for more on using the mouse in an xterm.

10. GETTING INPUT — THE EDITORS

One common requirement of a user callable function is to get data from the user. This is often done using command line arguments,as described in Section 6, or alternatively using some kind of editor. Several methods are provided by the user interface, ranging from getting a simple YES/NO response to a full screen page editor. The full screen editor is beyond the scope of this document, and is in practice rarely required, so only a few of the smaller editors will be discussed here. Examples of the use of these editors can be found in the example program in Appendix A.

The first basic editor is the function `line_edit` which has the declaration

```
int  line_edit(
       WINDOW    *win,     /* Window that string lives in */
       int  y, int x,      /* Position in window to place string */
       char *string,       /* The string to edit */
       int  length,        /* The length of the string including NULL */
       int  value_type,    /* Type of value string will hold ie :
                            * INTEGER, FLOAT or STRING */
       bool (*value_check)(), /* Function to test value of string */
       bool insert_on)        /* TRUE if we are to go
                               * mode, FALSE if we are to go back to whatever
                               * mode we were in before.
                               */
```

This function lets the user edit a string, and emulates both a standard editor and the Unix editor `vi`; that is, if you type the escape key you are in command mode and keys such as `j`, `l` and `w` do as you would expect them to do (i.e., go down a line, go right one character and go forward one word). The `value_type` parameter is to let the editor know what kind of value is required. This can have three different values:

1. `STRING` — any character will be allowed.
2. `INTEGER` — only numeric characters will be allowed.
3. `FLOAT` — only numeric characters, '+', '-', 'E' and '.' will be allowed.

The parameter `value_check` points to a function that can be used to test the final value of the string. It gets the string as a parameter, performs the test and returns either `TRUE` or `FALSE`. If the result is `FALSE` the user is forced to re-edit the string. In cases where you do not need to perform these tests you can use the inbuilt function `return_true()`, which always returns `TRUE`.

When running within an xterm the line editor is aware of the mouse and will move the cursor to the appropriate position when the user left clicks inside the line being edited. The return value of `line_edit()` is the key used to exit the function.

The second basic editor is for enumerated types and has the following declaration:

```
int  pick_choice(
       WINDOW    *win,     /* Window that value lives in */
       int  y, int x,      /* Position in window to place value */
```

```
        int  *value,            /* Variable to be set and default */
        char *strings[],        /* The array of strings to show */
        bool (*value_check)())  /* Function to test value */
```

In this case the array of character pointers `strings` contains a set of strings that describe each of the enumerated type choices. Each of these strings must be the same length and the final entry in the array must be `NULL`. The enumerate type is assumed to have the first value of 0. Hitting the space bar advances the choice while the return key will set the parameter pointed to by `value` to the current choice. Like `line_edit()` if `pick_choice()` is run inside an xterm it does sensible things when the user left clicks the mouse. Also like `line_edit()`, the return value is the key used to exit the function.

The third basic editor is for asking question that require a yes or no response. It has the declaration

```
int  ask_yes_no(char *string1,char *string2)
```

The two strings are displayed on the two lines of the system window and the function waits for either a '`y`' or '`n`' key as a replay. The return value will be either `YES` or `NO`.

There is also a 'compound' edit function called `quick_edit()` with the declaration:

```
int  quick_edit(
        char *item,         /* string describing what is being changed */
        char *status,       /* Default value */
        void *value,        /* pointer to string to be played with  */
        char *strings[],    /* string array for enumerated types */
        int  value_type)    /* STRING, FLOAT, INTEGER or ENUMERATED */
```

This function is probably the most commonly used function for getting single values from the user. It puts up a prompt message in the system window, based on the string `item` and then uses either `line_edit()` or `pick_choice()` on the command line to get the value. If the `value_type` is `ENUMERATED` the function uses `pick_choice()` and you need to supply the array of strings, otherwise `line_edit()` is used and you can set `strings` to `NULL`. As before, the return value is the key used to exit the function. It is up to the user to scan the final value out of the string.

11. SCROLLING TEXT IN A WINDOW

When you have a lot of text to display, more than can fit within a single window, you can use the function `void scroll_text(WINDOW *win)` to scroll this text in a specified window. The text must first be placed into the global display structure,

```
struct sdisplay {
                        int     number_items;
                        char    *string[NUM_TEXT_LINES];

                } *display;
```

where `number_items` is set to the number of lines and the array of character pointers `string[]` points to each line in turn, assumed allocated by the programmer. Before filling this structure it is safest to call the function `clear_display()`.

If you want to place all the text in a file into a scroll window you can use the function `text_format()` which has the declaration

```
void text_format(WINDOW *win, char *filename);
```

In fact, this is the function used by the online help system discussed in Section 8, and it will understand the same formatting commands. The function `printd()`, which works like printf, can be used to place a line of text into the display structure.

12. SOCKETS

When you have multiple control systems, each with its own user interface, it is often necessary to be able to move data from one system to another, or control one system by another. It is also often useful to be able to remotely type commands when you are not physically behind the console of the machine. This functionality is provided in the user interface using standard Unix sockets.

12.1. Command Socket

One of the parameters listed in Section 5 as a requirement for starting the user interface is a port number to use for a command socket. This must be an unused port but can otherwise be any number. This port is opened and the user interface listens to this port looking for commands. This is done as part of the background processing, so no socket commands will be seen if the program is servicing a request from a user on the physical console. In fact, a user at the console will always get priority over a socket user.

The command socket works exactly like the command line, although not all commands will be allowed over a socket. There is not much security on the sockets, but there is a macro defined in the header file `charaui.h` defining the allowed domain from which command sockets will be accepted. Right now this is set to

```
#define ALLOWED_DOMAIN "mtwilson.edu"
```

so only machines on the mountain will be able to use the sockets. It is also possible to define the macro `STANDALONE` within the source file `socket.c` to compile a version for a machine not connected to a network. The maximum number of simultaneous commands sockets is set by the macros `MAX_CONNECTIONS` defined in `charaui.h`.

If you plan to allow a socket user to run one of you functions be sure to add a line like

```
 socket_test_args(3,"arg1 arg2 arg3");
```

to the top of the function. This ensures that the socket user types the correct number of arguments. In this example there are three command line arguments required going by the names `arg1`, `arg2` and `arg3`. If you do not want to allow socket calls to a function put the line

```
no_socket();
```

at the top of the function.

The user on the keyboard is able to log all socket commands, monitor all connections, block all sockets and so on. Of course, these functions must be placed into the command definition structure defined in the file `functs.c`. See Appendix B for an example and a list of these standard functions.

12.2. The active socket

If you are going to allow socket commands in a function, and that function needs to display things, you will need to add extra flexibility to that function. It must know whether it has been called by a keyboard user, in which case it puts the stuff onto the screen, or by a socket user, so it sends the data to the socket. This is done by checking the global variable `active_socket`. If `active_socket` is set to `-1` the function was called by a user typing at console and the function should send its data to the screen. If `active_socket` is not set to `-1` the command was sent by a socket user and `active_socket` will be set to the file descriptor of that socket. All the output should be sent to that socket with a call like

```
socket_print(active_socket,"The result is %f.\n",result);
```

The function `socket_print` behaves just like the standard `printf` except the results are sent to a socket. You can send a message to all socket users with the function

```
all_socket_print("Hello socket users");
```

which also behaves a lot like `printf`.

12.3. Data Sockets

Apart from the command socket, automatically created on startup, it is also possible to create new sockets for moving data around from one machine to another, or indeed to send a command to a different control program. It is entirely up to the programmer to decide on data protocols and so on. The follow functions are available for socket use:

```
int open_data_socket(int port);
int call_open_data_socket(int argc, char **argv);
void close_data_socket(void);
int call_close_data_socket(int argc, char **argv);
int read_data(void *buf, int n);
int read_data_fast(void *buf, int n);
int data_ready(void);
int write_data(void *buf, int n);
int write_data_fast(void *buf, int n);
int data_connected(void);
int data_open(void);
```

Please refer to the various manual pages for more information on these functions.

12.4. Connect Sockets

Both the command and data sockets will 'listen' for incoming data. Sometimes it is necessary to open a socket that will connect to another program which has already established a 'listening' socket. The function `open_connect_socket()` will create such a socket. This function will either return the socket file descriptor or -1 if the action fails. The most common reason for failure is the non-existence of the socket you wish to connect to.

13. COMPILING AND LINKING

Of course, once you've written it, the program must be compiled and linked. An example makefile for doing this is given in Appendix D. There are a few important points for the compile and link stages:

- Each source file must include the header file `charaui.h`.
- I recommend using the flags `-g -O -Wall -pedantic` when compiling using `gcc`.
- The program must be linked to the user interface library, the ncurses library, the math library and the standard C library. The link flags `-lcharaui -lncurses -lc -lm` will achieve this.

One final comment: as with many new programming environments, you will probably find it easier to grab a working controller user interface program and fiddle with it rather than starting from scratch.

A. EXAMPLE APPLICATION PROGRAM

```
/**********************************************************************/
/* testui.c                                                           */
/*                                                                    */
/* Test program for CHARAUI.                                          */
/**********************************************************************/
/*                                                                    */
/*                    CHARA ARRAY USER INTERFACE                      */
/*                 Based on the SUSI User Interface                   */
/*                In turn based on the CHIP User interface            */
/*                                                                    */
/*            Center for High Angular Resolution Astronomy            */
/*              Mount Wilson Observatory, CA 91001, USA               */
/*                                                                    */
/* Telephone: 1-626-796-5405                                          */
/* Fax      : 1-626-796-6717                                          */
/* email    : theo chara.gsu.edu                                      */
/* WWW      : http://www.chara.gsu.edu                                */
/*                                                                    */
/* (C) This source code and its associated executable                 */
/* program(s) are copyright.                                          */
/*                                                                    */
/**********************************************************************/
/*                                                                    */
/* Author : Tony Johnson & Theo ten Brummelaar                        */
/* Date   : Original Version 1990 - ported to Linux 1998              */
/**********************************************************************/

#include "charaui.h"

int time_status(void);                  /* Example background job below */
int data_status(void);                  /* Background job tests data socket */

int main(int argc, char **argv)
{
        int         port;

        /* Check command line */

        if (argc !=4)
        {
                fprintf(stderr,"usage: %s menufile helpfile port\n",argv[0]);
                exit(-1);
        }
        sscanf(argv[3],"%d",&port);

        /* Title page. */

        ui_clear_screen();
```

```
        put_line("");
        center_line("CHARA USER INTERFACE TEST ROUTINE");
        put_line("");
        center_line("The CHARA Array");
        center_line("Center for High Angular Resolution Astronomy");
        center_line("Mount Wilson Observatory, CA 91001, USA");
        put_line("");
        center_line("Telephone: 1-626-796-5405");
        center_line("Fax: 1-626-796-6717");
        center_line("email: theo chara.gsu.edu");
        center_line("WWW: http://www.chara.gsu.edu");
        put_line("");
        center_line("(C) This executable program is copyright.");
        wait_for_title();

        /* Initialize the user interface */

        TITLE = "TESTUI VERSION 0.0";
        initialise_ui(argv[1], argv[2], port);

        /* Setup background job(s) */

        background_add(time_status);
        background_add(data_status);

        /* Let's go! */

        start_ui(); /* Should never return from here. */

        exit(0);
}

/*
 * Here follows and example background job.
 */

#define clean(y,x)        mvwaddstr(status_window,y,x,\
                                "                                       ")

/**********************************************************************/
/* time_status()                                                      */
/*                                                                    */
/* Displays the time in the status window.                            */
/**********************************************************************/

int time_status(void)
{
        long current_time;
        struct tm *now;
```

```
        time(&current_time);
        now = localtime(&current_time);
        clean(0,0);
        wstandout(status_window);
        mvwaddstr(status_window,0,0,"Local Time : ");
        wstandend(status_window);
        wprintw(status_window,"%2d:%02d:%02d",
                now->tm_hour,now->tm_min,now->tm_sec);

        return NOERROR;

} /* time_status() */

/*
 * String for enumerated type
 */

char        *example_types[] = {
                                "Type One  ",
                                "Type Two  ",
                                "Type Three",
                                "Type Four ",
                                "Type Five ",
                                NULL
                          };

/**************************************************************************/
/* example()                                                              */
/*                                                                        */
/* An example function showing various windows and editing functions.   */
/**************************************************************************/

int example(int argc, char **argv)
{
        int         an_int;
        float       a_float;
        char        a_string[81];
        int         an_enumerate;
#ifdef __NCURSES_H
        /*
         * Note: Only ncurses has mouse support, and only for xterms
         */

        MEVENT  mouse;                  /* A mouse event. */
#endif

        no_socket();

        /* Clean things up */
```

```
        werase(command_window);
        wrefresh(command_window);

        /* Have a look at the commands, just like a C programme */

        *a_string = 0;
        for (an_int=0; an_int<argc; an_int++)
        {
                strcat(a_string,argv[an_int]);
                strcat(a_string," ");
        }

        /* Put it up into the warning/error window call */

        error(WARNING,
                "This is the warning/error window. You typed :\n%s\nLast line.",
                a_string);

        /*
         * Introducing the main_window.
         */

        active_window = main_window; /* So system knows what you're up to. */
        werase(main_window);
        mvwaddstr(main_window,0,0,"This is the main_window.");
        mvwaddstr(main_window,2,0,
                "You can do what you like in the main_window.");
        wrefresh(main_window);

        /*
         * Introducing the system_window
         */

        message(system_window,
                "This is the system_window, it's for messages like this:\n\
Type a key to continue.");

        /*
         * Here's how to poll the keyboard.
         */

        while(!kbhit())
        {
                background(); /* Process background jobs while we wait */
        }
        an_int = get_command(); /* Got to clear the character! */
        werase(system_window); /* And clean up the system_window */
        wrefresh(system_window);

        /*
```

```
 * We can do general editing in any window with line_edit();
 */

strcat(a_string,"Like this one");
mvwaddstr(main_window,4,0,"Like editing a line of text : ");
if (line_edit(main_window,4,30,a_string,30,STRING,return_true,TRUE) ==
        KEY_ESC) return NOERROR;
error(MESSAGE,"The string ended up as:\n%s",a_string);

/*
 * Introducing the heading and sub_main_windows
 */

active_window = heading_window;
heading(heading_window,"This is the heading_window.",
        "It's used often for headings obviously.");

active_window = sub_main_window;
werase(sub_main_window);
mvwaddstr(sub_main_window,1,0,"This is the sub_main_window.");
mvwaddstr(sub_main_window,3,0,
        "In the heading window the first line is always centered.");
mvwaddstr(sub_main_window,4,0,
        "The second line isn't.");
mvwaddstr(sub_main_window,5,0,
        "You can do what you like in the sub_main_window.");
mvwaddstr(sub_main_window,6,5,
        "It's for general purposes (this line at (6,5))");
wrefresh(sub_main_window);

message(system_window,"Type a key to continue.");
while(!kbhit()) background();
an_int = get_command();
werase(system_window);
wrefresh(system_window);

/* Try and edit an enumerated type */

an_enumerate = 0; /* Set the default */
if (quick_edit("Example type",example_types[0],
                &an_enumerate,example_types,ENUMERATED) == KEY_ESC)
                        return WARNING;
message(system_window,"You're selection was %s\nType a key to continue.",
        example_types[an_enumerate]);
while(!kbhit()) background();
an_int = get_command();

/*
 * Try and edit an string type
 * Specify STRING type let's the editor to accept
```

```
 * any value character.
 */

sprintf(a_string,"This is the default.    ");
if (quick_edit("Example string",a_string,
                a_string,NULL,STRING) == KEY_ESC) return NOERROR;
message(system_window,"You're string is %s\nType a key to continue.",
        a_string);
while(!kbhit()) background();
an_int = get_command();
werase(system_window);
wrefresh(system_window);

/*
 * Try and edit an integer type
 * Specify INTEGER type forces the editor to only accept
 * numerals as input.
 */

an_int = 42; /* Set the default */
sprintf(a_string,"%d    ",an_int); /* Leave room now */
if (quick_edit("Example int",a_string,
                a_string,NULL,INTEGER) == KEY_ESC) return WARNING;
sscanf(a_string,"%d",&an_int);
message(system_window,"You're integer is %d\nType a key to continue.",
        an_int);
while(!kbhit()) background();
an_int = get_command();
werase(system_window);
wrefresh(system_window);

/*
 * Try and edit an float type
 * Specify FLOAT type forces the editor to only accept
 * numerals decimal places etc as input.
 */

a_float = 3.1415; /* Set the default */
sprintf(a_string,"%f    ",a_float); /* Leave room now */
if (quick_edit("Example int",a_string,
                a_string,NULL,FLOAT) == KEY_ESC) return WARNING;
sscanf(a_string,"%f",&a_float);
message(system_window,"You're integer is %f\nType a key to continue.",
        a_float);
while(!kbhit()) background();
an_int = get_command();
werase(system_window);
wrefresh(system_window);

/* There's even mouse use. */
```

```
#ifdef __NCURSES_H
        error(MESSAGE,"There's even mouse support in an Xterm.\
\nAfter clicking on ESC below, left click the mouse or type <ESC>.\n");

        while(TRUE)
        {
                if ((an_int = get_command()) == KEY_ESC)
                {
                        break;
                }

                if (an_int != KEY_MOUSE)
                {
                        error(WARNING,"No silly the MOUSE!");
                        continue;
                }

                getmouse(&mouse);

                if (mouse.bstate != BUTTON1_CLICKED)
                {
                        error(WARNING,"LEFT click please!");
                        continue;
                }

                error(MESSAGE,"You click it at position (%d,%d).\n\
Note that the mouse works on most things in the interface already.",
                        mouse.x, mouse.y);
                break;
        }
#endif

        /* Unless there's a problem, return NOERROR */

        /* You can ask questions too */

        if (ask_yes_no("You can ask questions too like...",
                      "Do you want to report an error?") == YES)
        {
                return WARNING;
        }

        return NOERROR;

} /* example() */

/************************************************************************/
/* data_status()                                                        */
/*                                                                      */
```

```
/* Tries to read from the data socket. Reports what it gets and sends   */
/* it straight back again.                                              */
/************************************************************************/

int data_status(void)
{
        int        len;
        char        s[256];
        long current_time;
        static long last_time = 0;

        clean(1,0);
        wstandout(status_window);
        mvwaddstr(status_window,1,0,"Data Ready : ");
        wstandend(status_window);
        if (data_ready())
        {
                wprintw(status_window,"YES");
                len = read_data(s,256);
                clean(2,0);
                wstandout(status_window);
                mvwaddstr(status_window,2,0,"Data Read  : ");
                wstandend(status_window);
                wprintw(status_window,"%d",len);
                clean(3,0);
                wstandout(status_window);
                mvwaddstr(status_window,3,0,"Data is    : ");
                wstandend(status_window);
                s[len]=0;
                wprintw(status_window,"%s",s);
        }
        else
        {
                wprintw(status_window,"NO ");
                clean(2,0);
                wstandout(status_window);
                mvwaddstr(status_window,2,0,"Data Read  : ");
                wstandend(status_window);
                wprintw(status_window,"%d",0);
                clean(3,0);
                wstandout(status_window);
                mvwaddstr(status_window,3,0,"Data is    : ");
        }

        time(&current_time);
        if (current_time >= last_time + 5)
        {
                last_time = current_time;
                len = write_data("Time to bug you.\n",
                        sizeof("Time to bug you.\n"));
```

```
                clean(4,0);
                wstandout(status_window);
                mvwaddstr(status_window,4,0,"Data Write : ");
                wstandend(status_window);
                wprintw(status_window,"%d",len);
        }

        return NOERROR;

} /* data_status() */
```

B. EXAMPLE FUNCTION DEFINITION FILE

Here is the function definition file used by the example program in Appendix A. All the functions referred to here come from within the user interface itself except the example function listed last. It is normal to have all of the functions below available in any user interface implementation except this one example function.

```
/*************************************************************************/
/* functs.c                                                              */
/*                                                                       */
/* Description                                                           */
/*       Sets up look up table array of user callable function modules.*/
/*       This is the file to change (along with menus.ini) when         */
/*       installing new user callable functions.                        */
/*************************************************************************/
/*                                                                       */
/*                    CHARA ARRAY USER INTERFACE                         */
/*                 Based on the SUSI User Interface                      */
/*                In turn based on the CHIP User interface               */
/*                                                                       */
/*            Center for High Angular Resolution Astronomy               */
/*              Mount Wilson Observatory, CA 91001, USA                  */
/*                                                                       */
/* Telephone: 1-626-796-5405                                             */
/* Fax      : 1-626-796-6717                                             */
/* email    : theo chara.gsu.edu                                         */
/* WWW      : http://www.chara.gsu.edu                                   */
/*                                                                       */
/* (C) This source code and its associated executable                    */
/* program(s) are copyright.                                             */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/* Author : Tony Johnson & Theo ten Brummelaar                           */
/* Date   : Original Version 1990 - ported to Linux 1998                 */
/*************************************************************************/

/*
 * The following definition is required to ensure there are no
 * compile time re-definition errors.
 */

#define FUNCTS
#include "charaui.h"

/*
 * Declare any local functions here.
 */

int         example(int argc, char **argv);
```

```
/*
 * Definition of user callable function index array.
 * Note that this array starts and ends with NULLs. This is so a
 * index of 0 gives nothing and also to help find the end of the array
 * The strings preceding the function names are the command strings
 * that will be recognized by the command line and the menuing system.
 * All access to these functions are via this table.
 */


struct {
                char    *name;
                int     (*function)(int argc, char **argv);
        } functions[] =

        {       {NULL,          NULL}, /* Must begin with a NULL */

                /*
                 * Make sure all standard user interface functions
                 * are available.
                 */

#include<std_ui_functs.h>

                /*
                 * If this is a real-time control user interface,
                 * make sure all standard clock functions
                 * are available.
                 */

#include<std_rt_functs.h>

                /*
                 * Now add any local functions.
                 */

                {"example",     example},

                {NULL,          NULL}       }; /* Must end with a NULL */
```

C. EXAMPLE MENU DEFINITION FILE

Here is the menu definition file used by the example program in Appendix A. All the functions referred to here come from within the user interface itself except the example function listed last.

```
# File : menus.ini
#
# Purpose : initialization file for the menu system
#
# NOTE : The hash symbol '#' at any point in the file denotes that
#        the rest of the line is a comment.
#
# This file must exist in the home directory. If it is missing, there
# will be a fatal error generation. If there are corrupt sections of this
# file there may or may not be fatal, major, or warning messages.
#
# See the user's manual for information concerning the maintenance of
# this file.
#

MENU                MAIN
        help        Get help
        commands    List available commands
        auto        Select Auto function list
        utils       Utilities Menu
        background  Background control menu
        socket      Socket control menu
        example     Try the example function
        one         Select Menu One
        two         Select Menu Two
        end         Quit system

MENU                BACKGROUND
        help        Get help
        sb          Start Background
        stb         Stop Background
        bon         Background on
        boff        Background off
        bkp         Bypass Keyboard Polling
        sleep       Put Controller to Sleep
        end         Quit system

MENU                SOCKET
        help        Get help
        block       Block socket commands
        unblock     Unblock socket commands
        os          Open command socket
        cs          Close command socket
        soccom      Communicate with a socket
```

```
        bm          Start socket command monitor
        sm          Stop socket command monitor
        bl          Start socket command log
        sl          Stop socket command log

MENU                UTILS
        help        Get help
        auto        Select Auto function list
        bkp         Bypass Keyboard Polling
        sleep       Put Controller to Sleep
        shell       Get system shell
        dir         View directory
        asccom      Communicate with a serial port
        end         Quit system

MENU ONE
        help        Get help
        auto        Select Auto function list
        ni          Try a non-implemented call
        two         Select Menu Two
        three       Select Menu Three
        four        Select Menu Four
        five        Select Menu Five
        six         Select Menu Six
        utils       Utilities Menu
        end         Quit system

MENU TWO
        help        Get help
        auto        Select Auto function list
        one         Select Menu One
        ni          Try a non-implemented call
        three       Select Menu Three
        four        Select Menu Four
        five        Select Menu Five
        six         Select Menu Six
        utils       Utilities Menu
        end         Quit system

MENU THREE
        help        Get help
        auto        Select Auto function list
        one         Select Menu One
        two         Select Menu Two
        ni          Try a non-implimented call
        four        Select Menu Four
        five        Select Menu Five
        six         Select Menu Six
        utils       Utilities Menu
        end         Quit system
```

```
MENU FOUR
        help        Get help
        auto        Select Auto function list
        one         Select Menu One
        two         Select Menu Two
        three       Select Menu Three
        ni          Try a non-implimented call
        five        Select Menu Five
        six         Select Menu Six
        utils       Utilities Menu
        end         Quit system

MENU FIVE
        help        Get help
        auto        Select Auto function list
        one         Select Menu One
        two         Select Menu Two
        three       Select Menu Three
        four        Select Menu Four
        ni          Try a non-implimented call
        six         Select Menu Six
        utils       Utilities Menu
        end         Quit system

MENU SIX
        help        Get help
        auto        Select Auto function list
        one         Select Menu One
        two         Select Menu Two
        three       Select Menu Three
        four        Select Menu Four
        five        Select Menu Five
        ni          Try a non-implimented call
        utils       Utilities Menu
        end         Quit system

AUTOLIST
        sb
        od 1025
        boff
        example one two three
        bkp
        bon
        goto boff
        bkp
        ni These are some arguments
        goto bkp
        stb
        sb
```

A. EXAMPLE SCRIPT FILE

```
# File : test.scr
#
# Purpose : Test of the scripting system.
#
        onerror error
        ls
        onfatal fatal
        help asccom
        onmessage end
        ask Do you want to quit now?
        message Glad you didn't decide to quit.
        goto end
warning:
        message I'm warning you!
        goto end
error:
        message Oooops something terrible has happened.
        goto end
fatal:
        message Oh no... a fatal error!
        goto end
end:
        endscript
```

B. EXAMPLE HELP INITIALIZATION FILE

```
# NOTE :         The hash symbol '#' at any point in the file denotes that
#                the rest of the line is a comment.
#
# File :         help.ini
#
# Purpose :      Information for the help initialization process
#
# This file should reside in the home directory of the UI software.
# If this file is not found by the initialization process, a diagnostic
# message will be printed to the screen and the user will not have any help
# available for the software. This does NOT affect the other processes of
# the software. This file may be maintained and/or modified by the user within
# the following guidelines:
#
# 1. This file contains only the following information:
#        a. Filenames containing help text, and
#        b. A description of what that help text is.
#
# 2. The help files that are described in this file MUST exist in the help
#    directory. If the file is not found by the initialization process, a
#    diagnostic message will be printed to the screen. This does NOT affect
#    the other processes of the help system or the operation of the software.
#
# 3. If a user adds additional help files to or removes files from the help
#    directory, this file should be adapted accordingly or the help text
#    in the new file will not be available.
#    Refer to the header section of existing help files for further
#    information on creating help files.

# Filename                      Description of help available
# ----------------------------- ----------------------------------------------

../help/asccom.hlp      ASCII communication with a serial port
../help/soccom.hlp      ASCII communication with a socket port
../help/bkp.hlp         Blocking the keyboard
../help/block.hlp       Blocking the socket command port
../help/cs.hlp          Closing the command socket port
../help/help.hlp        Getting help
../help/shell.hlp       Getting a command shell
../help/dir.hlp         Listing the current directory
../help/commands.hlp    List all commands
../help/os.hlp          Opening the command socket port
../help/sleep.hlp       Putting the program to sleep
../help/end.hlp         Quiting the program
../help/auto.hlp        Running the automatic command list
../help/sb.hlp          Start background processing
../help/bl.hlp          Start logging socket commands
../help/bm.hlp          Start monitoring socket commands
```

../help/stb.hlp	Stop background processing
../help/sl.hlp	Stop logging socket commands
../help/sm.hlp	Stop monitoring socket commands
../help/boff.hlp	Turning background processing off
../help/bon.hlp	Turning background processing on
../help/unblock.hlp	Unblocking the socket command port
../help/asccom.hlp	asccom
../help/auto.hlp	auto
../help/bkp.hlp	bkp
../help/block.hlp	block
../help/bl.hlp	bl
../help/bm.hlp	bm
../help/boff.hlp	bo
../help/boff.hlp	boff
../help/bon.hlp	bon
../help/commands.hlp	commands
../help/cs.hlp	cs
../help/dir.hlp	dir
../help/end.hlp	end
../help/end.hlp	exit
../help/help.hlp	help
../help/dir.hlp	ls
../help/os.hlp	os
../help/ping.hlp	ping
../help/sb.hlp	sb
../help/shell.hlp	shell
../help/sleep.hlp	sleep
../help/sl.hlp	sl
../help/sm.hlp	sm
../help/soccom.hlp	soccom
../help/stb.hlp	stb
../help/end.hlp	stop
../help/unblock.hlp	unblock
../help/end.hlp	quit

C. EXAMPLE HELP FILE

```
.center
ASCCOM
.center
======
.paragraph
SYNOPSIS:
.paragraph
asccom {port}
.paragraph
DESCRIPTION:
.paragraph
ASCCOM allows the user to communicate with a serial port. Anything typed on the
keyboard is sent to the port and anything sent by the port is put on the
screen. All of this is done as standard ascii characters. Typing the
escape key will return the user to the menu system.

The default port is /dev/modem.
```

When formatted by the user interface this text should look like this:

```
                               ASCCOM
                               ======

 SYNOPSIS:

 asccom {port}

 DESCRIPTION:


 ASCCOM allows the user to communicate with a serial port. Anything
 typed on the keyboard is sent to the port and anything sent by the port
 is put on the screen. All of this is done as standard ascii characters.
 Typing the escape key will return the user to the menu system. The
 default port is /dev/modem.
```

D. EXAMPLE MAKEFILE

```
########################################################################
# Makefile                                                             #
#                                                                      #
# Makefile for the CHARAUI system.                                     #
########################################################################
#     Center for High Angular Resolution Astronomy                    #
# Georgia State University, Atlanta GA 30303-3083, U.S.A.              #
#                                                                      #
# Telephone: 1-626-796-5405        email : theo chara.gsu.edu          #
# Fax      : 1-626-796-6717        WWW   : http:www.chara.gsu.edu      #
#                                                                      #
# (C) This source code and its associated executable                   #
# program(s) are copyright.                                            #
########################################################################
# Author : Theo ten Brummelaar                                         #
# Date   : May 1998                                                    #
########################################################################

# If you are going to change something it'll probably be here:

CC= gcc
CFLAGS= -g -O -Wall -pedantic
LFLAGS= -charaui -lncurses -lc -lm

#
# Master target
#

all: testui

#
# test routines
#

testui: testui.o functs.o
	$(CC) $(CFLAGS) -o testui testui.o functs.o $(LFLAGS)

testui.o: testui.c
	$(CC) $(CFLAGS) -c testui.c

functs.o: functs.c
	$(CC) $(CFLAGS) -c functs.c
#
# Rule for cleaning up the directory
#

clean:
	rm -f testui.o testui core functs.o
```

E. MANUAL PAGES

Here follows a complete list of the functions in the user interface library. Manual pages are available for all of these functions.

```
all_socket_print()
alloc_list_item()
asccom()
ask_yes_no()
background()
background_add()
background_add_name()
background_del()
background_off()
background_on()
background_start()
background_stop()
block_sockets()
break_into_lines()
bypass_key_poll()
bypass_key_poll_off()
call_block_sockets()
call_close_command_socket()
call_close_data_socket()
call_function()
call_open_command_socket()
call_open_data_socket()
call_send_socket_command()
call_unblock_sockets()
center_line()
char_waiting()
check_command()
checkone()
clear_display()
close_command_socket()
close_curses()
close_data_socket()
close_menu_window()
close_serial_port()
command_processor()
commands()
copy_scred_field()
copy_scred_page()
data_connected()
data_open()
data_ready()
delete_script()
delete_scripts()
down_key()
edit_page()
error()
error_messages_off()
error_messages_on()
exception()
fill_text_format()
find_function()
find_list_item()
find_menu()
find_script()
flush_command_socket()
free_list_item()
get_command()
get_menu_help()
get_user_input()
getline()
go_to_sleep()
heading()
help_line()
highlight()
inc_number_text_lines()
init_auto()
init_curses()
init_display()
init_help()
init_menu_structure()
init_script()
initialise_ui()
key_help()
line_edit()
lock_file_name()
machines_connected()
menu()
message()
new_script()
open_command_socket()
open_connect_socket()
open_data_socket()
open_serial_port()
os_utils()
parse_script()
passone()
passthree()
passtwo()
pick_choice()
ping()
```

plain_screen_off()
plain_screen_on()
printd()
process_command_socket()
put_comefrom_into_store()
put_line()
put_store_reversed_into_gotonext()
quick_edit()
read_data()
read_data_fast()
refresh_menu_window()
refresh_screen()
remove_buffer()
return_true()
scred()
scripts()
scroll_text()
send_socket_command()
serial_getchar()
serial_gets()
serial_print()
serial_putchar()
serial_scan()
set_serial_baud_rate()
set_serial_bitlength()
set_serial_hard_handshake()
set_serial_parity()
set_serial_stopbits()
set_serial_xonxoff()
set_up_buffer()
set_up_menu_window()
show_char()
show_string()
soccom()
socket_gets()
socket_print()
socket_scan()
split_string()
start_socket_command_log()
start_socket_command_monitor()
start_ui()
stop_socket_command_log()
stop_socket_command_monitor()
strcmpi()
test_socket_scan()
text_format()
ui_auto()
ui_auto_socket()
ui_end()
ui_help()
ui_list()
ui_shell()
unblock_sockets()
unhighlight()
up_key()
view_dir()
wait_for_title()
wake_up()
write_data()
write_data_fast()
write_ready()