NAME

pinfo_ti – Description of the terminal interface to the portable Infocom datafile interpreter.

SYNOPSIS

```
#include <stdio.h>
#include "infocom.h"

extern gflags_t gflags;
extern char     *ti_location;
extern char     *ti_status;


const char *scr_usage;
const char *scr_long_usage;
const char *scr_opt_list;

int scr_cmdarg( int argc, char ***argv_p );

int scr_getopt( int c, const char *arg );

void scr_setup( int margin,
           int indent,
           int lines,
           int context );

void scr_begin();

void scr_putline( const char *buffer );

void scr_putscore();

void scr_putsound( int number,
             int action,
             int volume,
             int argc );

void scr_putmesg( const char *buffer, Bool is_error );

int scr_getline( const char *prompt,
            int length,
            char *buffer );

void scr_window( int size );

void scr_set_win( int win );

FILE *scr_open_sf( int length, char *buffer, int type );

void scr_close_sf( const char *buffer, FILE *fp, int is_save );

void scr_end();

void scr_shutdown();
```

DESCRIPTION

These routines form the interface between the portable Infocom datafile interpreter (pinfocom) and a particular type of terminal. The pinfocom package comes with interfaces to the following terminal types:

termcap
: This terminal type supports all(?) UNIX systems via an interface to either termcap or terminfo terminal capability and either termio, termios, or sgtty line discipline control. Optional support for the GNU readline line editing and history library is also available.

amiga
: This terminal type supports an Amiga interface, including sound and command line editing. See amiga.c for full details.

msdos
: This terminal type supports MS-DOS based systems. See msdos.c for full details.

stream
: This terminal type should support any system with a C compiler; it uses simple C stdio routines that all C libraries will have. Nothing fancy, but it works.

Also in the works is an X11-based interface.

This man page describes the interface between the pinfo interpreter and the terminal support packages so that programmers can create their own interfaces to new systems. *Terminal interface* will be abbreviated to **TI** below.

Environment

The interpreter operates in two modes: game-playing mode (default) and information mode (if any of the command-line options **−h**, **−o**, **−O**, **−v** and/or **−V** are given), where the game is not played but information about it is instead displayed on the screen.

All input and output in game-playing mode is done via the **scr_*()** functions described below. Output during information mode is via simple C stdio library functions.

The only interface between the interpreter and the terminal handling code is through the functions and global variables described below; no other symbols in the TI code should be externally visible and any other external symbols in the interpreter code are subject to change.

GENERAL

Some general issues to be considered:

Asynchronous Events

The interpreter registers a signal handler for the SIGINT signal and sets to ignore the SIGQUIT signal (if defined).

Any other signals or other asynchronous events are free to be handled by the TI as it sees fit. Note, however, that none of these handler routines may modify or access any interpreter variables: interpreter variables listed below are only guaranteed to be valid when one of the scr_*() functions below has been properly called from within the interpreter code.

Proportional Fonts

If the TI supports multiple fonts it is perfectly reasonable to use proportional-width fonts for printing general game text if you wish.

However, at certain points in the game fixed-width fonts are required, such as when printing maps, etc. To accomodate this, before printing to the screen **scr_putline()** (see below) should invoke the macro **F2_IS_SET(B_FIXED_FONT)**. If this macro returns non-0 then the current buffer should be printed in a fixed-width font; if it returns 0 then a proportional-width font may be used if desired.

Scripting

All Infocom games support the **script** command, which is supposed to begin a transcript of the user's adventure. Normally this initialized the printer and all subsequent text and commands were printed to the printer as well as to the screen.

The TI may support this feature as well. There's no Z-Code primitive for turning on and off scripting, however: instead a flag is set or reset. In **scr_putline()** and **scr_getline()** the macro call **F2_IS_SET(B_SCRIPTING)** should be used; if it returns non-0 then the output line or prompt and command should be printed to the script file as well as the display.

The interface will call **scr_open_sf()** with a file type of **SF_SCRIPT** when a script file is to be opened and **scr_close_sf()** when a script file is to be closed.

Fixed Window

Only *Seastalker* makes use of the fixed window feature in Standard Series games, but it's nice to have nevertheless.

If the TI can support a fixed window in addition to the normal scrolling text window then **scr_begin()** should notify the interpreter by calling **F1_SETB(B_STATUS_WIN)**. In this case the **scr_window()** and **scr_set_win()** functions described below will be used to create/manage/delete the fixed window.

If the TI cannot support a fixed window then the above macro should not be called and **scr_window()** and **scr_set_win()** may be dummy functions.

GLOBAL VARIABLES

These global variables are defined in the interpreter code and may be used by the TI:

gflags

Contains global flags and other general variables of interest. The only field which may be modified is the **pr_status** field, if the TI doesn't support status line printing.

ti_location

This nul-terminated string is a short description of the current location of the player, for printing on the status line. It is usually displayed on the left-hand side of the status line.

ti_status

This nul-terminated string contains the status of the player. Depending on the game this may be score, score and turn number, or the current time in a 12-hour clock. This string is usually printed on the right-hand side of the status line.

These global variables should be defined in the TI .c file; they will be declared external where used in the interpreter code.

scr_usage

The **scr_usage** variable should be initialized to a string containing info about any extra command-line options that the TI makes available. Here's an example from the *termcap* TI:

```
const char *scr_usage = "[-H file] [-C file]";
```

If there are no extra command-line options the variable should be set to the empty string (""), **not** NULL.

scr_long_usage

The **scr_long_usage** variable should be initialized to be a string containing a one-line per option description of any extra command-line options that the TI makes available. Here's an example from the *termcap* TI:

```
const char *scr_long_usage = "\
\t-H file\tread/store command history in file\n\
\t-C file\tread/store user command completions in file\n";
```

If there are no extra command-line options the variable should be set to NULL.

scr_opt_list

The **scr_opt_list** variable should be initialized to be a string containing getopt(3)-style descriptions of any extra command-line options that the TI makes available. Briefly, each option character is placed in the string; if the option has an argument (no optional arguments allowed) then it should be immediately followed by a colon (:). Here's an example from the *termcap* TI:

```
const char *scr_opt_list = "C:H:";
```

If there are no extra command-line options the variable should be set to the nul string ("").

Interpreter Macros

The following macros are defined in *infocom.h* and available to the TI writer for access to certain data structures.

The flags available with **F1_*()** macros are:

B_USE_TIME
If set them time is displayed on the status line, otherwise score/moves is displayed. This is a read-only value; the TI cannot change it.

B_TANDY
The Tandy license flag; if set then Tandy licensing mode is turned on, otherwise it's turned off.

B_ALT_PROMPT
The alternate prompt flag; if set then alternate prompting is enabled, otherwise it's disabled.

B_STATUS_WIN
If set the TI supports fixed windows (for *Seastalker*), if not set it doesn't.

The flags available with **F2_*()** macros are:

B_SCRIPTING
If set then scripting mode is currently enabled, otherwise it's disabled.

B_FIXED_FONT
If set then the current buffer must be printed in a fixed-width font if possible; otherwise it may be printed in a proportional font.

B_SOUND
If set the interpreter may issue **scr_putsound()** requests; otherwise it won't. Note this flag is read-only and may not be set by the TI.

F?_IS_SET()
This macro returns a 0 if the specified game header flag is not set, and a non-0 (note **not** necessarily 1!) if it is not set.

F?_SETB()
This macro sets the specified game header flag.

F?_RESETB()
This macro resets (un-sets) the specified game header flag. The following functions are defined in the interpreter and available to the TI writer for use (no other functions in the interpreter should be called).

xmalloc()
This function calls **malloc()** and exits with an error if there is no heap left.

xrealloc()
This function calls **realloc()** and exits with an error if there is no heap left.

chop_buf()

buf Buffer to be chopped.

max Max number of chars wanted

This function locates the longest section of **buf** less than or equal to **max** characters containing at least one complete word. It does not modify the buffer. It returns a pointer to the space where the line should be broken, or to the nul character ('\0') if **buf** is less than or equal to **max** chars long.

This function is useful with fixed-width font TIs for chopping up the buffer passed to **scr_putline()** into sections which fit on the screen. See examples of its use in *stream.c*.

TI Functions

The following functions need to be provided by the TI code.

scr_cmdarg()

argc Number of command-line args.

argv_p Pointer to an array of strings; each one is a command-line argument.

This function will be called before any other scr_*() function, and before the command-line arguments are examined by the interpreter. It allows the TI to add (and possibly remove) arguments to the command line.

If the interface has the ability to determine command-line options from an alternate source, such as a configuration file, an environment variable, windowing resources, or saved game resources, then it should add these options to **argv_p** so they will be noticed by the interpreter.

Note that when adding options they should be inserted between **(*argv_p)[0]** and **(*argv_p)[1]**, shifting the existing options over, so that they will be properly overridden by any options given on the command line.

This function should return the number of elements in **argv_p** after modification (the new argument count). It is expected that after this function is called, **(*argv_p)[0]** remains the name of the interpreter program, **(*argv_p)[1]** to **(*argv_p)[argc-1]** are command-line arguments, and **(*argv_p)[argc]** is 0 (NULL).

Although it is perfectly legal to remove arguments from **argv_p** as well, this is trickier because you must follow all UNIX getopt(1) conventions. It is suggested that **scr_getopt()** is used to process arguments on the command line, and **scr_cmdarg()** used only to add extra arguments if needed.

scr_getopt()

c The command-line option character.

arg The command-line option argument (if used).

This function will be called each time an option specified in **scr_opt_list** is found on the command line. It is guaranteed that only options contained in **scr_opt_list** will be passed, and each option which requires a command-line argument will have one. Error messages for poorly formed options will be printed by the interpreter and this function will not be called.

scr_setup()

margin
The number of spaces in the right margin.

indent The number of spaces in the left margin. Note that normally only the output text is indented, not the command prompts.

lines The number of lines per screenful. If this value is non-0 then the TI must make every effort to use this value as the number of lines per screenful. If it is 0 then the interface must infer the screen size as best it can.

context
The number of lines that should be kept at the top of the screen when scrolling; this many lines should be left at the top of each page of output if a paged output mode is in effect.

This function is called after the game has initialized and examined its command-line arguments but before anything else is done. It is called for both game-playing and information modes, so it should perform any setup necessary for general stdio printing.

This function should return the number of characters that may be printed on one line in informational mode. If game-playing mode is to be used then this value is ignored.

scr_shutdown()

This function should perform any general end processing appropriate for both game-playing and informational modes.

scr_begin()

This function will be called just before the interpreter goes into game-playing mode. If the interpreter is running in information mode this function will never be called.

Any game-playing mode specific processing should be done here, such as opening a new window, clearing the screen, setting terminal characteristics, etc. After this function returns the screen should be set up and the cursor should be positioned at the lower-left hand corner of the screen.

Additionally, if the TI supports the creation of a fixed status window (in addition to the normal status line) it should call the macro **F1_SETB(B_STATUS_WIN);** here to notify the game Z-Code so that it will properly call the **scr_window()** and **scr_set_win()** functions (see below).

scr_end()

This function should perform any end processing appropriate for game-playing mode only, such as deleting windows, resetting terminal characteristics, etc.

scr_putline()

buffer A nul-terminated string containing any number of characters. The string may be empty, but the pointer will not be NULL. There will be no newline character ('\n') at the end of the string.

This function should print **buffer** to the screen, performing whatever line wrapping, paging, etc. is necessary. Processing should proceed as follows:

For each lineful of characters in **buffer**:

* print whatever indent was specified.
* print a lineful along with a newline and whatever scrolling is necessary.
* if the interface supports paging, and it's enabled, and it's been (screenful - context) lines since the last prompt was printed, then perform whatever paging is necessary.

If the TI supports proportional- and fixed-width fonts then the macro call **F2_IS_SET(B_FIXED_FONT)** should be used before printing **each** line; if the macro returns non-0 then that line must be printed in a fixed-width font if possible (maps, etc.). If the macro returns 0 then a proportional font may be used if desired.

If the TI supports scripting then the macro call **F2_IS_SET(B_SCRIPTING)** should be checked; if the macro returns non-0 then **buffer** should be printed to the script file as well.

scr_putscore()

This function is called whenever the values on the status line are to be displayed. If the interface supports status-line printing and it is enabled, then this function should use the values of **ti_location** and **ti_status** to display the status line on the screen in whatever manner it chooses (these variables always contain the proper values; they may be used by any TI function).

This function should be also be called whenever the TI code might have disturbed the status line.

scr_putsound()

number
The sound number to be played.

action The action to be taken.

volume
The volume at which to play the sound (between 1 and 8).

argc The number of arguments (1 to 3) passed to the function.

This function will be called by the interpreter if the game being played contains sound support. If the interface doesn't support sound then a simple output line may be printed by this function stating that fact.

Unfortunately different versions of Infocom games require different methods of sound support, but generally a separate sounds file is supplied with the datafile and the sounds must be retrieved and played from there.

scr_putmesg()

buffer A nul-terminated string containing a message from the interpreter.

is_error
Set to 1 if the message is an error message, or 0 if it's an informational message only.

This function is called whenever the interpreter needs to print a message for some reason. Messages from the game will be printed via scr_putline(). The TI should **not** attempt to exit the program or anything else; the interpreter will decide what to do about the message.

scr_getline()

prompt
A nul-terminated string which is the prompt to be printed when requesting a command.

length The maximum number of characters (including the nul character) which can be placed in **buffer**.

buffer A string of **length** characters, in which the command should be returned.

This function is called when the interpreter needs command input from the user. The function should accept up to **length**-1 characters of input and place them, along with a terminating nul character, into **buffer**. All command history, editing, shell escaping, etc. etc. must be done internal to this function. Also if the user types more than **length**-1 characters the function should flush the rest of the input and notify the user that it is doing so.

If this function causes the status line to be messed up (i.e., a shell escape causes extra lines to be printed, etc.) care should be taken to rewrite the status line correctly.

The interpreter supports an "interpreter escape mode", where interpreter flags and options may be modified. This mode is invoked by the user typing the escape character (by default "@") as the first character on the command line. If this function sees that as the first character, it should call **ti_escape()** with the remainder of the command line (**not** the escape character). Once that function returns the prompt should be reprinted and more input should be obtained without returning. If the TI wishes to provide its own escape commands it should process them itself and not call **ti_escape()** for those commands.

The escape char by itself will list the available options. If the TI wishes to supply options in addition to the interpreter options, it should call **ti_escape()** first, then print its own options.

Note that the TI may provide alternative ways of manipulating these options, such as pull-down or pop-up menus on windowing systems; **ti_escape()** can be called whenever control is properly passed to the TI; not from a signal handler, etc.

If the TI supports scripting, then it should check the command to see if it was "script". If it was and scripting is not already enabled then it should be enabled here. If the enabling fails then an error should be printed and the TI should ask for another command without returning to the interpreter.

The function should return the number of characters placed into **buffer**, not counting the terminating nul character.

scr_window()

size Create a fixed status window **size** lines high. If **size** is 0, then delete the current fixed status window (if any).

This function is called if the game supports a fixed status window (currently *Seastalker* is the only such Standard Series game), and the TI has registered that it supports such functionality by calling **F1_SETB(B_STATUS_WIN)** in scr_begin() (see above).

The function will be passed the number of vertical lines needed for the fixed status window. When the window is to be deleted it will be passed an argument of 0. Note it may be passed a 0 even if the window currently doesn't exist; in this case the function should just silently return.

If the TI doesn't support a fixed status window then just provide a dummy function which does nothing.

scr_set_win()

win Determines which window to start printing in.

This function is called if the game supports a fixed status window (currently *Seastalker* is the only such Standard Series game), and the TI has registered that it supports such functionality by calling **F1_SETB(B_STATUS_WIN)** in scr_begin() (see above).

After the **scr_window()** function has been called (see above), this function will be used to switch back and forth between the two windows. If called with **win** set to 0, then the TI should set up to start printing in the regular text window. If called with **win** set to 1, then set up to start printing in the fixed status window. All actual printing is done with the normal **scr_putline()** function.

scr_open_sf()

length The maximum number of characters (including the nul character) which can be placed in **buffer**.

buffer A string of **length** characters, in which the filename should be returned.

type The type of save file to be opened: SF_SAVE, SF_RESTORE, or SF_SCRIPT.

This function is called when the interpreter needs to open a save game file for either saving or restoring, or to open a script file.

When **scr_open_sf()** is called, **buffer** will contain a default filename (initially it will be a static name, after the first call it will contain the last filename entered). If **length** is 0, then the file should just be opened and the file pointer returned. If **length** is greater than 0, the function should print an appropriate prompt, then accept up to **length**-1 characters of input and place them, along with a terminating nul character, into **buffer** then open that file and return the file pointer.

If **type** is SF_SAVE or SF_SCRIPT, then the file is being opened to save the game or write script to and so should be opened for writing (binary mode for SF_SAVE and plain text mode for SF_SCRIPT). The function should test if the file already exists and if so, ask the user to confirm overwriting it.

If **type** is SF_RESTORE, then the file is being opened to restore a game and so should be opened for reading in binary mode.

Any command history, editing, filename completion, shell escaping, etc. etc. must be done internal to this function. Also if the user types more than **length**-1 characters the function should flush the rest of the input and notify the user that it is doing so.

If this function causes the status line to be messed up (i.e., a shell escape causes extra lines to be printed, etc.) care should be taken to rewrite the status line correctly.

If the TI wishes to use this function for other purposes, the **type** argument can be used: the interpreter will only call this function with values >=0 for **type** so the TI is free to call it with special values <0 if it wishes.

The function should return a FILE pointer to the open file. If the user cancels the operation somehow or the open fails, then the function should return 0 (NULL). If the return value is NULL then the interpreter will examine **errno**: if **errno** is 0 then the interpreter will assume the operation was canceled; if it's non-0 then the interpreter will assume there was an error during opening of the file and print an appropriate error message.

If the function returns non-NULL, the interpreter will attempt to read/write data from the current position in the file without rewinding or other manipulation. This means that you may store TI-specific information at the beginning of saved game files and read them back in when restoring if you like.

scr_close_sf()

filenm The name of the game file just saved/restored

fp FILE pointer of the game just saved/restored

type The type of save file to be opened: SF_SAVE, SF_RESTORE, or SF_SCRIPT.

This function is called immediately after a game save file is saved or restored or a script file is to be closed. At the very least this function should perform an **fclose(fp)** to close the file. Additionally it may add more terminal-specific data, create special configuration files, change file privileges, etc.

If the TI wishes to use this function for other purposes, the **type** argument can be used: the interpreter will only call this function with values >=0 for **type** so the TI is free to call it with special values <0 if it wishes.

Writing a New Interface

If you wish to create a new TI, I suggest that you start with stream.c as the simplest current interface. Be sure you examine all of this code so you understand in detail what each function does. You should then create your own .c file and add functionality as you like.

MAKEFILE

After you create your new TI you should name the .c file something relevant, such as *pcdos.c* for an interface to an IBM PC-compatible in DOS, or *pcwindows.c* for an interface to an IBM PC-compatible running MS-Windows, or something. Then you should create a new TERMTYPE section in the Makefile. Decide what, if any, flags you need to give to the compiler and fill in TERMFLAGS; likewise for any extra libraries you need and TERMLIB.

If you'd like your interface distributed with the pinfocom distribution, and an interface doesn't already exist, then send it to the pinfocom maintainer and he/she'll incorporate it in. Note you must place your interface code under the GNU Public License for it to be distributed (see the copyright notices in the other files), and provide an address for people to contact you about bugs, enhancements, etc. in a comment in the .c file.

If an interface already exists then you'll have to contact the author and the two of you can hash it out between you :-).