NAME

tack − *terminfo* action checker

SYNOPSIS

tack [−itV] [term]

DESCRIPTION

The **tack** program has three purposes: (1) to help you build a new terminfo entry describing an unknown terminal, (2) to test the correctness of an existing entry, and (3) to develop the correct pad timings needed to ensure that screen updates do not fall behind the incoming data stream.

Tack presents a series of screen-painting and interactive tests in ways which are intended to make any mismatches between the terminfo entry and reality visually obvious. **Tack** also provides tools that can help in understanding how the terminal operates.

OPTIONS

−i Usually **tack** will send the reset and init strings to the terminal when the program starts up. The *−i* option will inhibit the terminal initialization.

−t Tell **tack** to override the terminfo settings for basic terminal functions. When this option is set **tack** will translate (cr) to \r, (cud1) to \n, (ind) to \n, (nel) to \r\n, (cub1) to \b, (bel) to \007, (ff) to \f and (ht) to \t.

−V Display the version information and exit.

term Terminfo terminal name to be tested. If not present then the $TERM environment variable will be used.

OVERVIEW

Since **tack** is designed to test terminfo entries it is not possible to rely on the correctness of the terminfo data base. Because of this the menuing system used with **tack** is vary primitive. When a menu is printed it will scroll the entire screen. To compensate for this verbose menu system **tack** permits menu selection type ahead. If you already know what action you would like **tack** to perform then you can enter that value immediately and avoid the menu display. When in doubt the question mark (?) is a good character to type. A carriage return will execute the default action. These default actions are designed to run all the standard tests.

When **tack** first comes up it will display some basic information about the terminal. Take some time to verify this information. If it is wrong many of the subsequent tests will fail. The most important item is the screen size. If the screen size is wrong there is no point in proceeding. (home) and (clear) are also critical to the success of subsequent tests. The values of (cr) (ind) (cub1) and (ht) may effect the tests if they are defined incorrectly. If they are undefined **tack** will set them to reasonable defaults. The last two entries on the display are the enquire and acknowledge strings. These strings are taken from the user strings (u9) and (u8).

By now you must be wondering why the terminfo names are enclosed in parenthesis. This has no profound meaning other than it makes them stand out. The **tack** program uses this convention any time it displays a terminfo name. Remember **tack** is designed to rely on as little of the terminfo entry as possible.

CREATING NEW ENTRIES

Tack has a number of tools that are designed to help gather information about the terminal. Although these functions are not dependent on terminal type, you may wish to execute **tack** with options *−it*. This will turn off initialization and default the standard entries.

These tools may be reached from the main menu by selecting the “tools” entry.

Echo tool: All data typed from the keyboard will be echoed back to the terminal. Control characters are not translated to the up arrow format but are sent as control characters. This allows you to test an escape sequence and see what it actually does. You may also elect to **enable hex output on echo tool** this will echo the characters in hexadecimal. Once the test is running you may enter the “lines” or “columns” keywords which will display a pattern that will help you determine your screen size. A complete list of keywords will be displayed when the test starts. Type “help” to redisplay the list of available commands.

Reply tool: This tool acts much like the echo tool, but control characters that are sent from the terminal more than one character after a carriage return will be expanded to the up arrow format. For example on a standard ANSI terminal you may type:

CR ESC [c

and the response will be echoed as something like:

^[[? 6 c

ANSI sgr display: This test assumes you have an ANSI terminal. It goes through attribute numbers 0 to 79, displaying each in turn and using that SGR number to write the text. This shows you which of the SGR modes are actually implemented by the terminal. Note: some terminals (such as Tektronix color) use the private use characters to augment the functionality of the SGR command. These private use characters may be interjected into the escape sequence by typing the character (<, =, >, ?) after the original display has been shown.

ANSI status reports: This test queries the terminal in standard ANSI/VT-100 fashion. The results of this test may help determine what options are supported by your terminal.

ANSI character sets: This test displays the character sets available on a ANSI/VT-100 style terminal. Character sets on a real VT-100 terminal are usually defined with smacs=\E(0 and rmacs=\E(B. The first character after the escape defines the font bank. The second character defines the character set. This test allows you to view any of the possible combinations. Private use character sets are defined by the digits. Standard character sets are located in the alphabetic range.

VERIFYING AN EXISTING ENTRY

You can verify the correctness of an entry with the "begin testing" function. This entry is the default action and will be chosen if you hit carriage return (or enter). This will bring up a secondary menu that allows you to select more specific tests.

The general philosophy of the program is, for each capability, to send an appropriate test pattern to the terminal then send a description of what the user should expect. Occasionally (as when checking function-key capabilities) the program will ask you to enter input for it to check.

If the test fails then you have the option of dynamically changing the terminfo entry and re-running the test. This is done with the "edit terminfo" menu item. The edit submenu allows you to change the offending terminfo entry and immediately retest the capability. The edit menu lets you do other things with the terminfo, such as; display the entire terminfo entry, display which caps have been tested and display which caps cannot be tested. This menu also allows you to write the newly modified terminfo to disc. If you have made any modifications to the terminfo **tack** will ask you if you want to save the file to disc before it exits. The filename will be the same as the terminal name. After the program exits you can run the tic(1M) compiler on the new terminfo to install it in the terminfo data base.

CORRECTING PAD TIMINGS

Theory of Overruns and Padding

Some terminals require significant amounts of time (that is, more than one transmitted-character interval) to do screen updates that change large portions of the screen, such as screen clears, line insertions, line deletions, and scrolls (including scrolls triggered by line feeds or a write to the lowest, right-hand-most cell of the screen).

If the computer continues to send characters to the terminal while one of these time-consuming operations is going on, the screen may be garbled. Since the length of a character transmission time varies inversely with transmission speed in cps, entries which function at lower speeds may break at higher speeds.

Similar problems result if the host machine is simply sending characters at a sustained rate faster than the terminal can buffer and process them. In either case, when the terminal cannot process them and cannot tell the host to stop soon enough, it will just drop them. The dropped characters could be text, escape sequences or the escape character itself, causing some really strange-looking displays. This kind of glitch is called an *overrun*.

In terminfo entries, you can attach a **pad time** to each string capability that is a number of milliseconds to delay after sending it. This will give the terminal time to catch up and avoid overruns.

If you are running a software terminal emulator, or you are on an X pseudo-tty, or your terminal is on an RS-232C line which correctly handles RTS/CTS hardware flow control, then pads are not strictly necessary. However, some display packages (such as ncurses(3X)) use the pad counts to calculate the fastest way to implement certain functions. For example: scrolling the screen may be faster than deleting the top line.

One common way to avoid overruns is with XON/XOFF handshaking. But even this handshake may have problems at high baud rates. This is a result of the way XON/XOFF works. The terminal tells the host to stop with an XOFF. When the host gets this character, it stops sending. However, there is a small amount of time between the stop request and the actual stop. During this window, the terminal must continue to accept characters even though it has told the host to stop. If the terminal sends the stop request too late, then its internal buffer will overflow. If it sends the stop character too early, then the terminal is not getting the most efficient use out of its internal buffers. In a real application at high baud rates, a terminal could get a dozen or more characters before the host gets around to suspending transmission. Connecting the terminal over a network will make the problem much worse.

(RTS/CTS handshaking does not have this problem because the UARTs are signal-connected and the "stop flow" is done at the lowest level, without software intervention).

Timing your terminal

In order to get accurate timings from your terminal **tack** needs to know when the terminal has finished processing all the characters that were sent. This requires a different type of handshaking than the XON/XOFF that is supported by most terminals. **Tack** needs to send a request to the terminal and wait for its reply. Many terminals will respond with an ACK when they receive an ENQ. This is the preferred method since the sequence is short. ANSI/VT-100 style terminals can mimic this handshake with the escape sequence that requests “primary device attributes”.

ESC [c

The terminal will respond with a sequence like:

ESC [? 1 ; 0 c

Tack assumes that (u9) is the enquire sequence and that (u8) is the acknowledge string. A VT-100 style terminal could set u9=\E[c and u8=\E[?1;0c. Acknowledge strings fall into two categories. 1) Strings with a unique terminating character and, 2) strings of fixed length. The acknowledge string for the VT-100 is of the first type since it always ends with the letter “c”. Some Tektronics terminals have fixed length acknowledge strings. **Tack** supports both types of strings by scanning for the terminating character until the length of the expected acknowledge string has arrived. (u8) should be set to some typical acknowledge that will be returned when (u9) is sent.

Tack will test this sequence before running any of the pad tests or the function key tests. **Tack** will ask you the following:

Hit lower case g to start testing...

After it sends this message it will send the enquire string. It will then read characters from the terminal until it sees the letter g.

Testing and Repairing Pad Timings

The pad timings in distributed terminfo entries are often incorrect. One major motivation for this program is to make it relatively easy to tune these timings.

You can verify and edit the pad timings for a terminal with the “test string capabilities” function (this is also part of the “normal test sequence” function).

The key to determining pad times is to find out the effective baud rate of the terminal. The effective baud

rate determines the number of characters per second that the terminal can accept without either handshaking or losing data. This rate is frequently less than the nominal cps rate on the RS-232 line.

Tack uses the effective baud rate to judge the duration of the test and how much a particular escape sequence will perturb the terminal.

Each pad test has two associated variables that can be tweaked to help verify the correctness of the pad timings. One is the pad test length. The other is the pad multiplier, which is used if the pad prefix includes "*". In curses use, it is often the first parameter of the capability (if there is one). For a capability like (dch) or (il) this will be the number of character positions or lines affected, respectively.

Tack will run the pad tests and display the results to the terminal. On capabilities that have multipliers **tack** will not tell you if the pad needs the multiplier or not. You must make this decision yourself by rerunning the test with a different multiplier. If the padding changes in proportion to the multiplier than the multiplier is required. If the multiplier has little or no effect on the suggested padding then the multiplier is not needed. Some capabilities will take several runs to get a good feel for the correct values. You may wish to make the test longer to get more accurate results. System load will also effect the results (a heavily loaded system will not stress the terminal as much, possibly leading to pad timings that are too short).

NOTE

The tests done at the beginning of the program are assumed to be correct later in the code. In particular, **tack** displays the number of lines and columns indicated in the terminfo entry as part of its initial output. If these values are wrong a large number of tests will fail or give incorrect results.

FILES

tack.log — If logging is enabled then all characters written to the terminal will also be written to the log file. This gives you the ability to see how the tests were performed. This feature is disabled by default.

term — If you make changes to the terminfo entry **tack** will save the new terminfo to a file. The file will have the same name as the terminal name.

SEE ALSO

terminfo(5), **ncurses**(3X), **tic**(1M), **infocmp**(1M). You should also have the documentation supplied by the terminal manufacturer.

BUGS

If the screen size is incorrect, many of the tests will fail.

AUTHOR

Concept, design, and original implementation by Daniel Weaver <dan.weaver@znyx.com>. Portions of the code and documentation are by Eric S. Raymond <esr@snark.thyrsus.com>.