Automated testing of the Curses library

Brett Lymn

Introduction

The curses library is a library that provides a terminal independent method of updating a terminal screen. It also provides optimisation of the screen update in an attempt to minimise the number of characters sent to the screen. It is a complex and subtle piece of software. When making modifications it is always difficult to ensure that the modifications do not affect the operation of the curses library as a whole. One method to ensure that unintended effect from changes are detected is to use some form of automated testing to exercise the code. The NetBSD tree has a testing framework called atf(1) which is used to test various portions of the code tree. One part of the tree that was not being tested was the curses library. This paper describes the approach taken to perform testing of the curses library functions in a flexible and straightforward manner.

Inception

Making modifications to the internals of curses is alway difficult because changes in the beheviour of the library may not be immediately visible but, rather, may cause aberrations that are difficult to reproduce in simple usage. Previously, curses testing amounted to firing up a curses based application such as vi(1) and seeing if things "looked right" which, in reality, does not exercise much of the curses library at all. Another test method is writing a specific test-frame to exercise some new code which is very ad-hoc and specific for a single task. Once the code was working to the creators liking the test code was discarded which meant there was no on-going testing of the functionality. Both these approaches have resulted in bugs being introduced which were not detected for some time and when the bugs did appear they manifested themselves in environments that were extremely difficult to debug. Clearly there was a need to have some method to attempt to detect bugs or behaviour changes early to prevent such situations arising.

In 2007 a Google Summer of Code project mentored by NetBSD created an automated test framework which could perform automatic testing of subsystems in a controlled manner. This project was imported into the NetBSD tree and work began on generating tests that could be run as part of a regular build. Build servers have been set up that continuously build and test the NetBSD sources. These build servers can provide fast feedback not only when a build is broken by a commit but also if a commit causes the automated tests to fail. By providing fast feedback the window of commits that could have caused the problem is much narrower which simplifies the task of finding the modification that caused the problem. It was clear that adding the curses library into the automated test framework would be beneficial to the project by ensuring that the curses library is continuously tested and any regressions found rapidly. The challenge was how to properly automate the testing of all the curses functionality. Curses has functions that perform timed reads and return errors if a character has not been seen within a specified time out. It can also assemble multiple input characters into a symbolic key representation as the arrow keys, for example, tend to send a multiple character string to represent an arrow key press. Both of these facilities make it difficult to simply pipe a string of characters into a test program because all the characters will arrive at the input at once with no means of testing a timeout. A program could be written to pace the input to the test program but there is still a matter of timing. In some cases there may need to be multiple curses calls performed to set up the correct state to test a particular function which can add delays. The program performing the input pacing has no indication of when the program under test is ready to accept input. Although atf has a timeout that will kill the misbehaving test the timeout is relatively long and would cause long delays in the test run. Another problem with simply redirecting input is that stdin is not classified as a tty device so it behaves differently to a tty device. Attempting to set terminal attributes will result in an error return which will affect some curses calls causing them to return errors. Similarly, capturing the output from a curses program under test is problematic. Redirecting or piping the output to a file also results in stdout not being a tty device which can affect the output routines of curses introducing unwanted output artifacts. Also, in some instances, curses sets the input device characteristics using an ioctl that waits for the output to be drained to ensure output sent prior to the ioctl is not affected by the change in device characteristics. This means that output must be drained promptly otherwise the program under test will stall. Finally, generating test cases should be as simple and flexible as possible to allow the quick development of new tests that are able to properly exercise the capabilities of the curses library. To resolve the previously mentioned difficulties it was decided to use a pseudo-tty interface (pty) to provide a terminal interface to the curses program under test. This required a master program that could not only provide input, if required, to the test program but would also capture the output from the test program in a timely manner to prevent the test program from stalling. A possible approach would be to develop a specific test program for each facet of curses that needed to be tested but this would involve writing a lot of repetitive code and the problem of coordinating input would still remain.

Implementation

To resolve all these difficulties it was decided to use a single test program, called the slave, this slave program is capable of running any curses function with arguments provided by the master program called the director. The slave and director are connected via a pseudo-tty interface to provide the correct tty semantics required by curses. The two processes are also connected by two pipes, one pipe allows the director to pass commands and arguments to the slave for execution. Another pipe allows the slave to pass back the return status for the function along with any other return values. The director does the bulk of the work, it parses a test command file using a simple interpreter to gather the required function to run and its arguments and passes these on to the slave. The director then wait for the slave to complete the execution of the function. Since the director and slave are running in lock-step the director can have a very short timeout for the command execution and so terminate a misbehaving test quickly. The director language allows input to be defined prior to the call that is going to require it along with inter-character timing. The director keeps a list of curses function that require input and will provide the pre-defined input at the pre-defined rate when a function requiring input is executed. Return values from the slave are read from the return pipe and the values are either saved or validated for expected content, any mismatch is flagged as an error and the test is terminated. The director also captures any output from the slave, originally the thought was to simply capture and compare output data when routine that caused screen output were called but it was found that stalled the slave process due to curses calling ioctls that waited for the output to drain before making changes. Given this, the director continuously drains output from the slave and stores it in a dynamically allocated buffer for later comparison. The prevents the slave from stalling due to output backlog. A directive in the director language causes the director to compare a file containing expected output against the buffered slave output and, possibly, any pending slave output. It is an error if there is insufficient slave output but excess slave output is flagged as a warning by the director. Once an output comparison has been performed any excess data may be discarded, depending on the comparison directive used. A test only passes if all return values for all functions match their expected values and, if appropriate, the expected output data matches the data stream produced by the slave.

Test Language

The director test files are parsed using a simple custom interpreter. The language allows defining and later using variables, it is loosely typed the type being determined at assigment time. There are only two types supported, strings (though there are a few string sub-types) and integers. Strings support a some character substitutions to permit values useful to curses testing to be performed, these substitutions are:

\e	escape
\n	newline
\\	the \ character
\nnn	The character represented by the octal number nnn

Depending on what quotes enclose the string defines how the string is treated. A plain, ordinary, string is enclosed in double quotes ("), this string will be null terminated. An array of characters is enclosed in single quotes (’) this array will not be null terminated and may contain any character value. An array of the curses chtype which is a pair of of bytes, the first byte being the character attributes and the second being the actual character is enclosed in back-ticks (‘). Integers may be expressed in decimal (no prefix) or hexadecimal (prefixed with 0x) Variables are defined when first assigned, the only requirement for a variable is that is start with a alphabetic character. Once defined a variable can be reference by prefixing the variable name with a dollar sign ($). Both integers and variables holding integers may be logically or’ed together by enclosing the list in parentheses (()) and separating the list members with a vertical bar (|). This is handy for combining bit values such as character attributes.

As mentioned previously, the director uses a simple interpreted language to determine the steps involved in performing a test. The language directives are:

assign
: Assign a value to a variable

call
: Call a curses function, expect only one return value

call2
: Like call but expect two return values

call3
: Like call but expect three return values

call4
: Like call but expect four return values

check
: Validate a variable against an expected value

compare
: Compare the output from the slave against the contents of a file

comparend
: The same as compare but don’t discard excess output from the slave

delay
: Define the inter-character delay to be applied to an input string

include
: include another command file. This is used to reduce the amount of repetition in test files by including commonly executed sections. There is a fixed, compile time, limit to the number of nested include files. The limit is arbitrary and set to 32 at the moment. It should be noted that there is no scoping of variables so they can be defined and or modified in deeper nested includes and be available at the top level test file and all intermediates.

input
: defines a string to be used for input when a curses routine that reads input is called.

noinput
: prevents the director from erroring if there has not been input defined for a curses input routine. Used if there is input pending already.

By using the above directives the testing of most of the functions in curses can be achieved. Initial conditions can be created by calling various curses routines in the same manner as a real curses application would. One important thing to not is that the slave automatically calls the initscr() function so it is not necessary for this call to be included in any test. The director expects a fixed number of returns from the slave function call depending on the call that was performed, the number of returns from the slave is validated and an error will be raised if there is a mismatch. Returns can either be validated immediately or assigned to a variable. For immediate validation the following values can be used:

OK
: standard curses success return

ERR
: standard curses error return

NULL
: a null pointer has been returned

NON_NULL
: a pointer that is not null valued has been returned

An
: integer including a logical OR

A
: string

For curses functions that return multiple values the returns are listed to the left of the curses function in the call statement. The rules for this are simple, any argument in a curses function that is a pointer to a return is listed on the left hand side of the function in the order they appear in the original function argument list. Ordinary function arguments are listed on the right hand side of the curses function name.

Below is a sample of a test script that creates a window and prints a message into it:

```
include start
```

```
call win1 newwin 2 5 2 5
check win1 NON_NULL
call OK wprintw $win1 “%s” “hello”
call OK wrefresh $win1
compare wprintw_refresh.chk
```

The first line includes another test file that checks the curses start up sequence, this is a set of characters that curses sends to the terminal to initialise it ready for use. This start sequence is common to all tests so is a good candidate for inclusion. The second line is a call to the curses `newwin` function, this call has one return, the pointer to a window structure. This return is saved in the variable `win1` for later use. The third line is a validation of `win1` in this case it is checked to ensure that `newwin` has not returned a NULL pointer which would indicate an error. Assuming that there is a valid pointer in `win1` the director will execute the next line which is a call to `wprintw`. Here the previously assigned `win1` is used as a parameter to `wprintw` to specify the window to print into along with the other parameters for the function call. After the `wprintw` call the `wrefresh` function is called with the `win1` parameter to update the terminal with the results of the previous calls. The final line compares the output stream of the test sequence against the contents of the file `wprintw_refresh.chk`. It should be noted that the compare need not happen at the end of the test, the output comparison can be performed whenever there is a need to validate output and can be done multiple times throughout the test.

To simplify the analysis of the output stream from the slave the curses testframe uses a special terminfo entry. Normally a terminal terminfo entry has terminal specific escape sequences to affect the behaviour of the terminal. These escape sequences are difficult to read and their meanings can be hard to determine. For the purposes of testing the curses testframe uses a terminfo entry that has capabilities that are mostly the names of the capabilities themselves with a few exceptions. On of the exceptions are the arrow and function keys so realistic sequences can be used. Another exception is the capabilities that move the cursor a single character, these are a single character otherwise the curses optimisation routines will not use them since it would determine that they would result in more characters being output when compared to using an absolute positioning capability. The major difficulty with writing a test for curses is determining whether or not the output is correct. As has been noted previously, the final screen appearance may be deceptive as it may hide subtle errors or be inefficient in terms of the number of characters output. For this reason it was decided that simply capturing the current output was not a good strategy as this may enshrine bugs in the code as being correct. To avoid this problem the output of each test is analysed to ensure that the expected behaviour is observed. To assist with this analysis, the test director has a verbose mode that reports the output stream in a readable format, the previously mentioned readable version of the terminfo entry also assists with interpreting the output. Using the verbose mode of the test director the expected output for a test can be written this is a painstaking and tedious process but does mean that there should be less bugs in the output. By starting with tests that peform basic operations such as initialising the curses library, create windows and so forth a library of tests can be built that can then be included into more complex tests which somewhat eases the burden of writing tests.

Currently, the curses testframe does not cover wide characters mainly due to the challenges in properly representing a wide character in the test language. Another avenue for further exploration is the testing of other curses based libraries. Libraries such as libform and libmenu should be able to be tested simply by adding the support into the slave process.

Conclusion

The curses testframe has been integrated into the NetBSD tests tree and work is progressing, albeit slowly, on adding tests to exercise as many of the curses functions as possible. Already this work has shown fruits with a number of bugs being found and fixed simply due to the fact of the functions and output being closely scrutinised. Once major portion of the curses functions are covered by testing it opens the possiblity of undertaking some major renovations to the curses internals with the assurance that unintended effects of the changes made will be detected and the bugs fixed before the user community is affected thus making NetBSD a more stable and reliable platform.