Automated Curses Testing

Brett Lymn

Introduction

- Curses is a terminal independent interface to a terminal
- Optimises screen updates to minimise data sent to terminal
- Parts are complex and subtle
- Easy to introduce bugs that are hidden during simple testing
- Opportunity to test curses using ATF to provide ongoing testing

Inception

- Working on curses has always been difficult due to lack of easy testing method
- Previously something like vi(1) was used to check things "looked right" or some special test code written
- Using a curses application may not test all capabilities, i.e. inadequate test coverage
- Using purpose built code limits the testing to that facility, difficult to write and debug, tended not to be reused

Inception II

- In GSoC 2007 an automated test framework (a.k.a. atf) was created
- Imported into the NetBSD tree and testing made part of the build process on the NetBSD build servers
- Tests added to exercise many parts of the NetBSD codebase
- One glaring omission was curses

Requirements

- Curses can do timed reads, need to test this
- Assembles multiple characters into a "key symbol" e.g. arrows, need to test this
- Hard to just pipe characters at a test application - timeout is tricky as is synchronisation
- ATF will kill a misbehaving test but the timeout is relatively long, very easy for a test to cause long delays

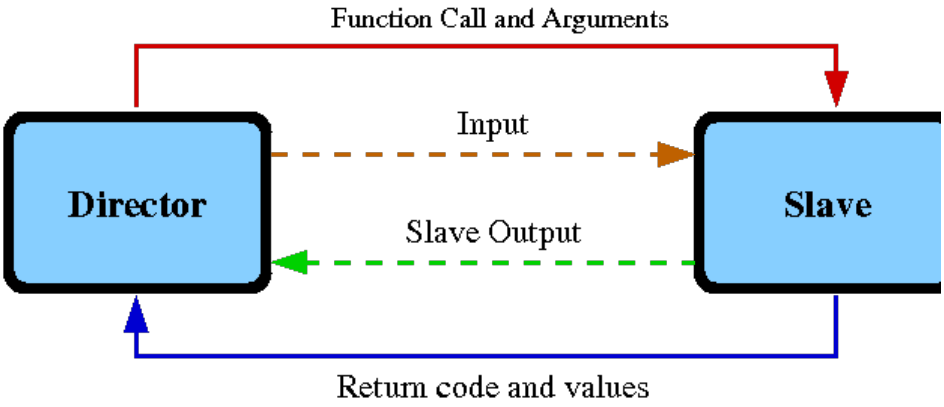
Requirements II

- Need something that makes it easy to create new tests
- redirecting stdout is *NOT* a tty!
- TIOCSETAW means output has to be drained promptly
- Simplify tests by reducing repetitive code as much as possible

Implementation

- Decided to use two interconnected programs
- One called the slave, this is the curses application and is able to run any curses function
- The other called the director that interprets a test file and drives the slave program
- Connected via a pseudo-tty and two pipes
- director saves output from slave for later checking

Process Interaction



Test Langua

- The director has a l
- able to set and use variables
- supports integers and strings
- can perform a simple bit or of integers
- validates return values

Test Language II

- supports nested include files up to a compile time limit (32 currently)
- note that variables are not scoped so include files can modify
- can define input for the slave
- can define timing of the input to the slave to test inter-character timeouts
- compares slave output against a file of expected output

Test Language - Returns

- Returns can be assigned to a variable or validated immediately
- Expected values can be any type or one of ERR, OK, NULL, NON_NULL
- Special command to validate the contents of a variable
- rules for return by reference simple, call parameters are shifted to the LHS in the order they occur, so:

```
int getyx(WINDOW *win, int y, int x)

Becomes:

call2 OK y x getyx $win
```

Test Language - Output Validation

- The director continuously saves output from the slave in a dynamic buffer
- This data can be validated against expected output at any stage during the test
- Writing the expected output is the most difficult, did not want to just capture current output
- Output analysis is made easier by a custom terminfo entry to make the output more readable
- Terminfo structured so it can be "translated" into real escape sequences mechanically
- Need to be careful about some terminfo capabilities so they don't affect curses output behaviour

Sample Test Files

```
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
```

```
include window
input "input\n"
call2 OK "input" wgetstr $win1
compare wgetstr.chk
call OK wrefresh $win1
compare wgetstr_refresh.chk
```

Conclusion

- More tests need to be written
- Already found some bugs due to analysis
- At the moment wide functionality cannot be tested, needs thought on how to handle this
- Framework need not be limited to just curses other curses based libraries like libform, libmenu could be tested