Terminal Emulation for Automation and Testing of Character-Graphic Programs: A Code Walkthrough

Don Libes
National Institute of Standards and Technology
Manufacturing Systems Integration Division
Gaithersburg, MD

Abstract

This paper describes a technique that allows automation and testing of character-graphic programs using existing public-domain tools. Specifically, Tcl, Tk, and Expect are augmented with a terminal emulator in order to build a screen representation in memory. This screen can be queried in a high-level way and the interaction can be further controlled based on the screen representation.

One immediate use of this is to build a test suite for automating standards conformance of all of the interactive programs in POSIX 1003.2 (Interactive Shells and Utilities). This technique is portable and inexpensive. All the software described in this paper is free or in the public domain.

This paper assumes a thorough understanding of Expect, Tcl, and Tk.

Keywords: Conformance Testing; Expect; Interaction Automation; POSIX 1003.2; Regression Testing; Tcl; Tk; X Window System

Introduction

Most character-graphic programs are interactive and provide no means for non-interactive control. This is not surprising. Character-graphics are of use primarily to humans, and humans have limited short-term memory. For example, humans want to see context when typing a report. Even if every line has been manually entered, it is desirable to have the lines displayed to make up for limited human short-term memory.

Reprinted from the *Proceedings of the 21st Annual Trenton Computer Festival (TCF '96)*, Trenton, NJ, April 21-22, 1996.

In contrast, a computer has no need to see context. The computer effectively has infinite memory and has no trouble remembering what it has just done. For example, if the computer is entering an address into a template, the computer does not need to constantly refer back to see what it has already entered. Similarly, the computer does not need to check if it entered the information correctly.

This distinction between computer-interaction and human-interaction impacts programs. Many programs cannot be automated by computers or automatically switch to non-graphical modes when driven this way. This makes regression testing of character-graphic programs difficult. Because it is so difficult, it is rarely performed. If performed at all, it is usually very limited and requires special coding with significant expense.

This paper describes a general technique that allows automation and testing of character-graphic programs using portable and inexpensive tools. Specifically, Tcl, Tk, and Expect are augmented with a terminal emulator in order to build a screen representation in memory. This screen can be queried in a high-level way and the interaction can be further controlled based on the screen representation.

One immediate use of this is to build a test suite for automating standards conformance of all of the interactive programs in POSIX 1003.2 (Interactive Shells and Tools) [POSIX94].

Background

Tcl is an embeddable language library which can be linked to other applications. Tcl provides a fairly generic but reasonably high-level language. The language is interpreted and resembles the UNIX shell in many ways. Elements are also derived from C and LISP. Despite its mixed heritage, much of the excess baggage from these other languages has been omitted leaving a modest but capable language. The language consists of a core of features called Tcl (Tool Command Language).

Tcl is extensible. Two popular Tcl extensions are Tk and Expect. Tk enables control of graphic user interfaces. Expect enables control of interactive character-oriented interfaces. Both Tk and Expect can work together. For example, they can be used to layer a graphic user interface on top of an existing character-oriented program.

Tcl and Tk are described by [Ouster93]. Expect is described by [Libes90, 91, 95]. The remainder of the paper assumes a reasonable understanding of Expect, Tcl, and Tk.

Traditional Expect Processing in Non-character-graphic Programs

In non-character-graphic applications, characters are written on each line from left to right. After completing a line, characters are written to the next line. When the last line of the screen is filled, the screen is scrolled. The oldest line at the top of the screen is deleted, all the other lines are moved up, and new characters are written to the new line at the bottom of the screen.

Since characters appear in exactly the order that they are written, it is simple to wait for specific patterns. As characters arrive, they are appended to a buffer. The buffer can then be searched for the patterns of interest.

For example, suppose a program prompts with the string “**`yes or no:`**”. This prompt can be detected by waiting for exactly that string to appear in the output of the program.

Expect is a popular public-domain program that automates interactive programs. Using Expect, the actual command to wait for the string “**`yes or no:`**” is:

```
expect "yes or no:"
```

Expect has a rich set of built-in tools to describe patterns. However, they are all serial in nature. Expect sees a stream of characters and does not attempt to interpret the characters in a different order than they were received.

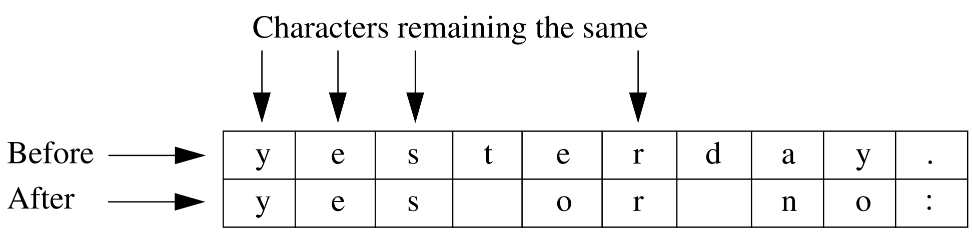
Expect Processing in Character-Graphic Programs

In contrast to non-character-graphic programs, character-graphic programs write characters to arbitrary character locations on a screen or window. For example, a DEC VT100 terminal can display a 24 by 80 grid of printable ASCII characters. Characters can only appear in discrete locations in the grid. However, the grid can be filled in any order and characters at any location may be replaced any time by other characters.

Special character sequences, usually beginning with an escape character, are used to position subsequent characters in the grid. I will refer to these as *positioning sequences*.

Because the grid may be filled in any order, it is not trivial to watch a stream of characters for patterns. Typically, such programs take advantage of characters that already exist on the screen to reduce the amount of characters that have to be produced to update the display.

For example, suppose a line on the screen contains “**`yesterday.`**”. If this is to be replaced with the “**`yes or no:`**” prompt, the program can rewrite the entire line with “**`yes or no:`**”. However, the program can achieve the same effect by replacing the “**`te`**” with “ **`o`**” and “**`day.`**” with “ **`no:`**”. This is shown in the following figure.



The output of this program to produce “**`yes or no`**” would be:

`yesterday.`<*positioning sequence*> **`o`**<*positioning sequence*> **`no:`**

The simple Expect command used earlier would not be able to match "**`yes or no`**" in such output. However, with an understanding of how to interpret the positioning sequences, it is possible to model the screen and match the string. In that case, the match is not made against the output directly. Instead, the match is made against the model of the screen.

Terminal Emulation

A screen may be modeled using *emulation*. Indeed, emulation is the basis for terminal emulators. Terminal emulators create a model of the screen and display it on a windowed system such as the X Window System. However, terminal emulators are not designed to support detection of patterns on the screen.

In this section, I present a terminal emulator that provides a framework upon which to perform screen analysis. The code presented below and on the next few pages defines the terminal emulator. Functionally, the emulator is capable of supporting sophisticated character-graphic programs such as vi [Lamb90] and emacs [Stall94]. Hooks are provided so that screen analysis can be done after each screen update.

For simplicity, the Tk system is used to implement the grid of characters. Tk is a system for controlling X11 graphics. Tk is convenient for maintaining the grid since Tk automatically displays the grid in a window. In practice, however, it is not necessary to display the grid. Indeed, nondisplay of the grid is useful when automating an existing program. In many cases, user comprehension of the character-graphic user interface is no longer necessary so there is no need to display it.

The grid is implemented using a Tk text widget. A text widget allows arbitrary placement of text within the grid. A number of other features are supported such as highlighting. I will describe these later. The code defines a text widget with a fixed-size display of 24 rows by 80 columns. The nature of the terminal emulator does not actually depend on this but some size must exist. Similarly, a shell is started, through which any programs under test will be invoked. For convenience, the name of the Tk widget is stored in the variable "**`term`**".

```
# tkterm - term emulator using Expect to control a Tk text widget

set rows 24          ;# number of rows in term
set cols 80          ;# number of columns in term

# start a shell and text widget for its output
set stty_init "-tabs"
eval spawn $env(SHELL)
stty rows $rows columns $cols < $spawn_out(slave,name)
set term_spawn_id $spawn_id

text $term -width $cols -height $rows
```

In theory, the task of understanding screen manipulation sequences is straightforward. However, in reality, it is complex. The complexity is due to many reasons that I will touch upon in the remainder of the paper. Some of the problems are:

- Many vendors use non-standard screen manipulation sequences.
- Even with a single screen manipulation definition, there is an infinite number of sequences that can generate a particular screen image.
- High-level databases and libraries exist to deal with the multi-vendor problem, however there is no single standard.
- Some programs do not follow the specifications described by the high-level databases/ libraries.

Intuitively, the way to build a terminal emulator is to figure out what the character sequences mean and model this in computer code. The solution presented in this paper is not far afield from that idea, but it gets there in a somewhat circuitous route.

First, it is necessary to understand that there is no standard terminal type. While there is an ANSI standard, it is so limited that all vendors extend it. Naturally, these extensions are rarely compatible with one another. Indeed, manufacturers often produce extensions that are different even within their own model lines.

Several attempts have been made to define high-level databases and software interfaces to understand these hundreds of definitions. However, these interfaces are for producing character sequences, not consuming them.

Given an arbitrary character sequence, there is no trivial way to figure out what it does. Presuming a particular terminal type simplifies the problem but does not necessarily make it solvable. For instance, while it is possible to invert the database descriptions, some of the sequences are proper subsets of one another. This makes it impossible to know which of two functions is being requested. This also arises when terminals are used beyond their documented limits. Sequences that were defined with only enough space for 24 rows, for instance, can theoretically match two different requests if a terminal with more than 24 rows is emulated. This is a common scenario with emulated terminals. Which request is correct can only be determined by the undocumented operation of the physical terminal itself. This can change from one release to the next and is not necessarily derivable via software.

To avoid these problems, I have taken the approach of an “ideal” terminal of my own design. By designing a terminal from scratch, I avoid the difficulties of having to deduce the characteristics of another existing terminal. There are two drawbacks of an ideal terminal:

- An ideal terminal cannot be automatically displayed on a terminal emulating a different type.
- A program that only generates output for a specific type will not necessarily display correctly on an ideal terminal.

Fortunately, both of these are moot. The first drawback is irrelevant partly because typically the emulator itself provides a display. I will later describe how to display the ideal terminal. In addition, the emulator can be augmented to consume characters meant for one terminal type and convert this into characters to drive yet another type. The second drawback is irrelevant because the programs of interest should not be tied to a particular terminal type but should be terminal independent. While it is possible to force the emulator to understand a particular terminal type, it can be much more difficult because the ideal terminal is invariably much simpler than any real terminal.

Defining Terminal Definitions

An arbitrary terminal definition would be meaningless if there were no way to inform programs of it, but the same databases as before serve this purpose. The approach taken by modern databases is to support arbitrary terminal types through the use of a terminal description language. Unfortunately, there is no single standard.

In UNIX environments, there are two "standards" – termcap and terminfo [Good91]. The presence of one of these can often be explained by the derivation of the system. termcap was invented at Berkeley and can be found on Berkeley-derived systems. terminfo was a redesign provided by AT&T and can be found on AT&T (i.e., SV) derived systems. Many systems support both and it is not uncommon to find half the utilities on the system using termcap and half using terminfo. Hence, the solution in this paper necessarily implements both. The script is forgiving in that it will run even if one of the two implementations fails due to its absence.

Fortunately, it is much easier to design a terminal description from scratch than it is to mimic an existing terminal description. The reason is that few sequences are actually mandatory. For instance, relative cursor motion can be simulated with absolute cursor motion. This one observation alone dramatically simplifies descriptions since there are often dozens of relative cursor motions which can be replaced by a single absolute cursor motion definition. Using a single, albeit more complex, definition also turns out to be more efficient than many relative cursor motion operations as I will explain later.

The following code establishes descriptions in both termcap and terminfo style using the ideal terminal type which is arbitrarily named "**tk**". The code succeeds even if termcap and terminfo are not supported on the system. This code actually has to be executed before the **spawn** shown earlier in order for the environment variables to be inherited by the process.

I will briefly describe the termcap definition. (The terminfo definition is very similar so I will skip those.) The definition is made up of several capabilities. Each *capability* describes one feature of the terminal. A capability is expressed in the form **xx=value**, where **xx** is a capability label and **value** is the actual string that the emulator receives. For instance the **up** capability moves the cursor up one line. Its value is the sequence: escape, "**[**", "**A**". These sequences are not interpreted at all by Tcl so they may look peculiar. The complicated-looking sequence (**cm**) performs absolute cursor motion. The row and column are substituted for each *%d* before it is transmitted. The character string "**\E**" is replaced with a true ASCII ESC (escape) character. The remaining capabilities are *nondestructive space* (**nd**), *clear screen* (**cl**), *down one line* (**do**), *begin standout mode* (**so**) and *end standout mode* (**se**). The actual definitions are based on the ANSI terminal definition [ANSI90]. This is a purely arbitrary choice.

```
set env(LINES) $rows
```

```
set env(COLUMNS) $cols

set env(TERM) "tk"
set env(TERMCAP) {tk:
     :cm=\E[%d;%dH:
     :up=\E[A:
     :nd=\E[C:
     :cl=\E[H\E[J:
     :do=^J:
     :so=\E[7m:
     :se=\E[m:
}

set env(TERMINFO) /tmp
set ttsrc "/tmp/tk.src"
set file [open $tksrc w]

puts $file {tk,
     cup=\E[%p1%d;%p2%dH,
     cuu1=\E[A,
     cuf1=\E[C,
     clear=\E[H\E[J,
     ind=\n,
     cr=\r,
     smso=\E[7m,
     rmso=\E[m,
}
close $file
catch {exec tic $tksrc}
exec rm $tksrc
```

For simplicity in this paper, the emulator only understands the generic standout mode rather than specific ones such as underlining and highlighting. The **term_standout** global variable describes whether characters are being written in standout mode. Text in standout mode is tagged with the tag **standout**, here defined by white characters on a black background.

```
set term_standout 0       ;# if in standout mode

$term tag configure standout \
         -background black \
         -foreground white
```

The text widget maintains the terminal display internally. It can be read or written in a few different ways. Access is possible by character, by line, or by the entire screen. Lines are newline delimited. It is convenient to initialize the entire screen (i.e., each line) with blanks. Later, this will allow characters to be inserted anywhere without worrying if the line is long enough already. In the **term_init** procedure (below), the “**insert $i.0**” operation adds a line of blanks to row **i** beginning at column 0.

```
proc term_init {} {
     set blankline [format %*s $cols ""]\n
     for {set i 1} {$i <= $rows} {incr i} {
```

```
            $term insert $i.0 $blankline
        }
```

For historical reasons, the first *row* in a text widget is 1 while the first *column* is 0. The variables **cur_row** and **cur_col** describe where characters are next written. Here, they are initialized to the upper-left corner.

```
        set cur_row 1
        set cur_col 0
```

The visible insertion cursor is maintained as a mark. It generally tracks the insertion point. Here, the cursor is also set to the upper-left corner.

```
        $term mark set insert $cur_row.$cur_col
    }
```

Once defined, the **term_init** procedure is called immediately to initialize the text widget.

```
    term_init
```

A few more utility routines are useful. The **term_clear** procedure clears the screen by throwing away the contents of the text widget and reinitializing it.

```
    proc term_clear {} {
        global term

        $term delete 1.0 end
        term_init
    }
```

The **term_down** procedure moves the cursor down one line. If the cursor is already at the end of the screen, the text widget appears to scroll. This is accomplished by deleting the first line and then creating a new one at the end.

```
    proc term_down {} {
        global cur_row rows cols term

        if {$cur_row < $rows} {
            incr cur_row
        } else {
            # already at last line of term, so scroll screen up
            $term delete 1.0 "1.end + 1 chars"

            # recreate line at end
            $term insert end [format %*s $cols ""]\n
        }
    }
```

There is no correspondingly complex routine to scroll up because the termcap/terminfo libraries never request it. Instead, they simulate it with other capabilities. In fact, the termcap/terminfo libraries never request that the cursor scroll past the bottom line either. However, non-character-graphic programs such as **cat** and **ls** do, so the terminal emulator understands how to handle this case.

The **term_insert** procedure writes a string to the current location on the screen. It is broken into three parts. The first part writes from anywhere on a line up to the end. If the string is long enough and wraps over several lines, the next section writes the full lines that wrap. Finally, the last section handles the last characters that do not make a full line. Characters are tagged with the standout tag if the emulator is in **standout** mode.

Each one of these sections does its work by first deleting the existing characters and then inserting the new characters. This is a good example of where termcap/terminfo fail to have the ability to adequately describe a terminal. The text widget is essentially always in "insert" mode but termcap/terminfo have no way of describing this.

One capability of which the script does not take advantage, is that termcap/terminfo can be told not to write across line boundaries. On that basis, this procedure could be simplified by removing the second and third parts. Again however, programs such as **cat** and **ls** expect to be able to write over line boundaries. The **term_insert** procedure does not worry about scrolling once the bottom of the screen is reached. **term_down** takes care of that already.

```
proc term_insert {s} {
     global cols cur_col cur_row
     global term term_standout

     set chars_rem_to_write [string length $s]
     set space_rem_on_line [expr $cols - $cur_col]

     if {$term_standout} {
          set tag_action "add"
     } else {
          set tag_action "remove"
     }

     ##################
     # write first line
     ##################

     if {$chars_rem_to_write > $space_rem_on_line} {
          set chars_to_write $space_rem_on_line
          set newline 1
     } else {
          set chars_to_write $chars_rem_to_write
          set newline 0
     }

     $term delete $cur_row.$cur_col \
                 $cur_row.[expr $cur_col + $chars_to_write]
     $term insert $cur_row.$cur_col [
          string range $s 0 [expr $space_rem_on_line-1]
     ]
```

```
    $term tag $tag_action standout $cur_row.$cur_col \
                  $cur_row.[expr $cur_col + $chars_to_write]

    # discard first line already written
    incr chars_rem_to_write -$chars_to_write
    set s [string range $s $chars_to_write end]

    # update cur_col
    incr cur_col $chars_to_write
    # update cur_row
    if $newline {
        term_down
    }

    ##################
    # write full lines
    ##################
    while {$chars_rem_to_write >= $cols} {
        $term delete $cur_row.0 $cur_row.end
        $term insert $cur_row.0 [string range $s 0 [expr $cols-1]]
        $term tag $tag_action standout $cur_row.0 $cur_row.end

        # discard line from buffer
        set s [string range $s $cols end]
        incr chars_rem_to_write -$cols

        set cur_col 0
        term_down
    }

    #################
    # write last line
    #################

    if {$chars_rem_to_write} {
        $term delete $cur_row.0 $cur_row.$chars_rem_to_write
        $term insert $cur_row.0 $s
        $term tag $tag_action standout $cur_row.0 \
                                $cur_row.$chars_rem_to_write
        set cur_col $chars_rem_to_write
    }

    term_chars_changed
}
```

At the very end of **term_insert** is a call to **term_chars_changed**. This is a user-defined procedure called whenever visible characters have changed. For example, the following code finds when the string **foo** appears on line 4:

```
proc term_chars_changed {} {
    global $term
    if {[string match *foo* [$term get 4.0 4.end]]} . . .
}
```

Some other tests suitable for the body of **term_chars_changed** are:

```
# Test if "foo" exists at line 4 col 7
if {[string match foo* [$term get 4.7 4.end]]}

# Test if character at row 4 col 5 is in standout mode
if {-1 != [lsearch [$term tag names 4.5] standout]} ...
```

Information can also be retrieved:

```
# Return contents of screen
$term get 1.0 end

# Return indices of first string on lines 4 to 6 that are
# in standout mode
$term tag nextrange standout 4.0 6.end
```

And here is possible code to modify the text on the screen:

```
# Replace all occurrences of "foo" with "bar" on screen
for {set i 1} {$i<=$rows} {incr i} {
     regsub -all "foo" [$term get $i.0 $i.end] "bar" x
     $term delete $i.0 $i.end
     $term insert $i.0 $x
}
```

The last utility procedure is **term_update_cursor**. It is called to update the visible cursor.

```
proc term_update_cursor {} {
     global cur_row cur_col term

     $term mark set insert $cur_row.$cur_col

     term_cursor_changed
}
```

The **term_update_cursor** procedure also calls a user-defined procedure, **term_cursor_changed**. A possible definition might be to test if the cursor is at some specific location:

```
proc term_cursor_changed {} {
     if {$cur_row == 1 && $cur_col == 0} ...
}
```

By default, both procedures do nothing:

```
proc term_cursor_changed {} {}
proc term_chars_changed {} {}
```

term_exit is another user-defined procedure. **term_exit** is called when the spawned process exits. Here is a definition that causes the script itself to exit when the process does.

```
proc term_exit {} {
```

```
        exit
}
```

The last user-defined procedure is **term_bell**. **term_bell** is executed when the terminal emulator needs its bell rung. The following definition sends an ASCII bell character to the standard output.

```
proc term_bell {} {
        send_user "\a"
}
```

Once all of the utility procedures are defined, the command to read the sequences is straightforward. For instance, a backspace character causes the current column to be decremented. A carriage-return sets the current column to 0.

Notice how simple the code is for absolute cursor motion. It is basically two assignment statements. Because it is so simple, there is no need to supply termcap/terminfo with information on relative cursor motion commands. They cannot be substantially faster.[1]

```
expect_background {
        -i $term_spawn_id
        -re "^\[^\x01-\x1f]+" {
                # Text
                term_insert $expect_out(0,string)
                term_update_cursor
        } "^\r" {
                # (cr,) Go to beginning of line
                set cur_col 0
                term_update_cursor
        } "^\n" {
                # (ind,do) Move cursor down one line
                term_down
                term_update_cursor
        } "^\b" {
                # Backspace nondestructively
                incr cur_col -1
                term_update_cursor
        } "^\a" {
                term_bell
        } eof {
                term_exit
        } "^\033\\\[A" {
                # (cuu1,up) Move cursor up one line
                incr cur_row -1
                term_update_cursor
        } "^\033\\\[C" {
                # (cuf1,nd) Nondestructive space
                incr cur_col
                term_update_cursor
        } -re "^\033\\\[(\[0-9]*);(\[0-9]*)H" {
```

1.The definition for nondestructive space might be seen as a concession to speed, but in fact it is required by some buggy versions of termcap which operate incorrectly if the capability not defined. The other relative motion capabilities are assumed by the terminal driver for non-character-graphic tools such as **cat** and **ls**.

```
            # (cup,cm) Move to row y col x
            set cur_row [expr $expect_out(1,string)+1]
            set cur_col $expect_out(2,string)
            term_update_cursor
        } "^\033\\\[H\033\\\[J" {
            # (clear,cl) Clear screen
            term_clear
            term_update_cursor
        } "^\033\\\[7m" {
            # (smso,so) Begin standout mode
            set term_standout 1
        } "^\033\\\[m" {
            # (rmso,se) End standout mode
            set term_standout 0
        }
    }
```

Finally, some bindings are provided. Bindings define how the emulator should handle user events such as user keystrokes and mouse motion. For example, the following statement defines a binding that applies to any keypress event. Upon occurrence of such an event, its action sends the corresponding ASCII character to the process. Keypress events that do have an associated ASCII character (such as "shift" and "control") are discarded.

```
    bind $term <Any-KeyPress> {
        if {"%A" != ""} {
            exp_send -i $term_spawn_id -- "%A"
        }
    }
```

The meta key is simulated by sending an ASCII escape. Most programs understand this convention, and it is convenient because it works over **telnet** links.

```
    bind $term <Meta-KeyPress> {
        if {"%A" != ""} {
            exp_send -i $term_spawn_id "\033%A"
        }
    }
```

These bindings are the same for any terminal and thus are not defined by explicit capabilities. Bindings that are unusual do require capabilities. For example, some terminals have function keys which generate a string of characters, typically unique to a particular brand of terminal. This behavior is described using a capability. For instance, the capability for function key 1 to send escape, "O", and "P" could be described in either of two ways:

:k1=\EOP:	*termcap-style*
:kf1=\EOP:	*terminfo-style*

The matching binding is:

```
    bind $term <F1> {exp_send -i $term_spawn_id "\033OP"}
```

Using The Terminal Emulator For Testing And Automation

It is possible to use the terminal emulator defined in the previous section to partially or fully automate or test character-graphic applications.

For instance, each expect-like operation could be a loop that repeatedly performs various tests of interest on the text widget contents. In the following code, the entrance to the loop is protected by "**tkwait var test_pats**". This blocks the loop from proceeding until the **test_pats** variable is changed. The variable is changed by the **term_chars_changed** procedure, invoked whenever the screen changes. Using this idea, the following code waits for a **%** prompt anywhere on the first line:

```
proc term_chars_changed {} {
     uplevel #0 set test_pats 1
}

while 1 {
     if {!$test_pats} {tkwait var test_pats}
     set test_pats 0
     if {[regexp "%" [$term get 1.0 1.end]]} break
}
```

Writing a substantial script this way would be clumsy. Furthermore, it prevents the use of control flow commands in the actions. One solution is to create a procedure that does all of the work handling the semaphore and hiding the **while** loop.

In [Libes90], a script is presented which partially automates the game of rogue, an adventure game. Because the game uses character graphics, the script can conceivably miss the patterns for which it is looking.

Using the techniques illustrated in this paper, it is possible to write a replacement script for rogue that fully understands the character graphics. The script is based on a procedure (shown later) called **term_expect**. This new script is similar to the earlier version except that instead of patterns, tests are composed of explicit statements. Any nonzero result causes **term_expect** to be satisfied whereupon it executes the associated action. For instance, the first test looks for **%** in either the first or second line on the screen. The meaning of the rest of the script should be obvious.

```
while 1 {
     term_expect {regexp "%" [$term get 1.0 2.end]}
     exp_send "rogue\r"
     term_expect \
          {regexp "Str: 18" [$term get 24.0 24.end]} {
               break
          } {regexp "Str: 16" [$term get 24.0 24.end]}
     exp_send "Q"
     term_expect {regexp "quit" [$term get 1.0 1.end]}
     exp_send "y"
}
```

In contrast to the original **rogue** script, there is no **interact** command at the end of this one. Because of the bindings, the script is *always* listening to the keyboard! If desired, this implicit interaction can be desabled by removing or overriding the **KeyPress** bindings that appear at the end of the terminal emulator.

Since the tests can be arbitrarily large lists of statements, they are grouped with braces. For example:

```
term_expect {
     set line [$term get 1.0 2.end]
     regexp "%" $line
} {
     action
} timeout {
     puts "timed out!"
}
```

Timeouts follow a similar syntax as before. A test for an eof is not provided since a terminal emulator should not exit just because the applications making use of it do so. In this example, a shell prompt is used to detect when the **rogue** program has exited.

The **term_expect** procedure lacks some of the niceties of **expect** and should be viewed as a framework for designing a built-in command.

The term_expect Procedure

An implementation of **term_expect** is shown in this section. The code assumes the presence of the terminal emulator shown in the previous section. Although the terminal emulator is necessary, the text widget and, indeed, Tk itself can be obviated by maintaining an explicit representation such as a list of strings representing rows of the terminal. However, even with Tk and the terminal emulator, the timeout and the scope handling makes the code intricate. Without them, the code would be more similar to the fragment on page 14.

Timeouts are implemented using an **after** command which sets a strobe at the end of the timeout period. In order to avoid an old **after** command setting the strobe for a later **term_expect** command, a new strobe variable is generated each time.[1] A global variable provides a unique identifier for this purpose and is initialized separately:

```
set term_counter 0          ;# distinguish different timers
```

The procedure begins by deciding the amount of time to wait before timing out. A local timeout is used if defined, otherwise the global timeout is used. If no global timeout is defined, 10 seconds is used. This behavior is exactly that of the real **expect** command.

```
proc term_expect {args} {
     upvar timeout local_timeout
     upvar #0 timeout global_timeout
```

1.Tk 4 promises to provide support for cancelling **after** commands. This would remove the need for separate strobe variables.

```
set timeout 10
catch {set timeout $global_timeout}
catch {set timeout $local_timeout}
```

Two unique global variables are used as strobes—to indicate that an event (data or timeout) has occurred. The **strobe** variable holds the name of a global variable changed when the terminal changes or the code has timed out. Later, the code will wait for this variable to change. To distinguish between the two types of events, **tstrobe** is another strobe changed only upon timeout. (It is possible to use a single tri-valued strobe, but the coding is much trickier.)

```
global term_counter
incr term_counter
global [set strobe _data_[set term_counter]]
global [set tstrobe _timer_[set term_counter]]
```

The **term_chars_changed** procedure is modified to fire the strobe. Note the use of double quotes around the body of **term_chars_changed** in order to allow substitution of the strobe command in this scope.

```
proc term_chars_changed {} "uplevel #0 set $strobe 1"
```

The next lines set the strobes to make sure that the screen image can be tested immediately since the screen could initially be in the expected state. The **after** command arranges for the timer strobe to be set later.

```
set $strobe 1          ;# force an initial test
set $tstrobe 0         ;# no timeout yet

if {$timeout >= 0} {
     set mstimeout [expr 1000*$timeout]
     after $mstimeout "set $strobe 1; set $tstrobe 1"
     set timeout_act {}
}
```

If the user omits the final action, the number of arguments will be uneven. Later code is simplified by adding an empty action in this case.

```
set argc [llength $args]
if {$argc%2 == 1} {
     lappend args {}
     incr argc
}
```

If the test is the bare string "**timeout**", its action is saved for later. Both the string and the action are removed from the list of tests.

```
for {set i 0} {$i<$argc} {incr i 2} {
     set act_index [expr $i+1]
     if {![string compare timeout [lindex $args $i]]} {
          set timeout_act [lindex $args $act_index]
          set args [lreplace $args $i $act_index]
```

```
            incr argc -2
            break
        }
    }
```

Now the procedure loops, waiting for the screen to be changed. A test first checks if the strobe has already occurred. If not, **tkwait** waits. This suspends the loop when no screen activity is occurring. Once the strobe occurs, the rest of the loop executes. If the timeout has occurred or any of the tests are true, the loop breaks so that the action can be evaluated.

```
    while {![info exists act]} {
        if {![set $strobe]} {
            tkwait var $strobe
        }
        set $strobe 0

        if {[set $tstrobe]} {
            set act $timeout_act
        } else {
            for {set i 0} {$i<$argc} {incr i 2} {
                if {[uplevel [lindex $args $i]]} {
                    set act [lindex $args [incr i]]
                    break
                }
            }
        }
    }
```

To keep the environment clean, the global strobe variables are deleted. If a timeout could occur in the future, the **unset** is similarly scheduled; otherwise the variables are deleted immediately. The **term_chars_changed** procedure is reset so that it does not continue setting the data strobe.

```
    proc term_chars_changed {} {}

    if {$timeout >= 0} {
        after $mstimeout unset $strobe $tstrobe
    } else {
        unset $strobe $tstrobe
    }
```

Finally, the action is evaluated. If a flow control command (such as **break**) was executed, it is returned in such a way that the caller sees it as well.

```
    set code [catch {uplevel $act} string]
    if {$code >  4} {return -code $code $string}
    if {$code == 4} {return -code continue}
    if {$code == 3} {return -code break}
    if {$code == 2} {return -code return}
    if {$code == 1} {return -code error \
                            -errorinfo $errorInfo \
                            -errorcode $errorCode $string}
    return $string
```

```
}
```

Another Example – Querying a Database

The following example connects to the Cornell University Library and makes a number of queries through its menu system. Interestingly, this library expects to drive a 3270 terminal. A 3270 terminal is not like a typical serial terminal and traditional programs such as **`telnet`** and **`rlogin`** do not support it. Thus, Expect uses the **`tn3270`** program to convert the 3270 interaction to a Curses-style character stream which can then be handled as usual.

First, the shell prompt is waited for and the 3270 emulator is started.

```
term_expect {regexp {.*[>%]} [$term get 1.0 3.end]}
exp_send "tn3270 notis.library.cornell.edu\r"
```

The next step is to get through the library's login interaction.

```
term_expect {regexp "desk" [$term get 19.0 19.end]} {
                exp_send "\r"
    }
```

Once in the library system, all the menus prompt the same way. Two utility routines are used to handle this repetitive situation.

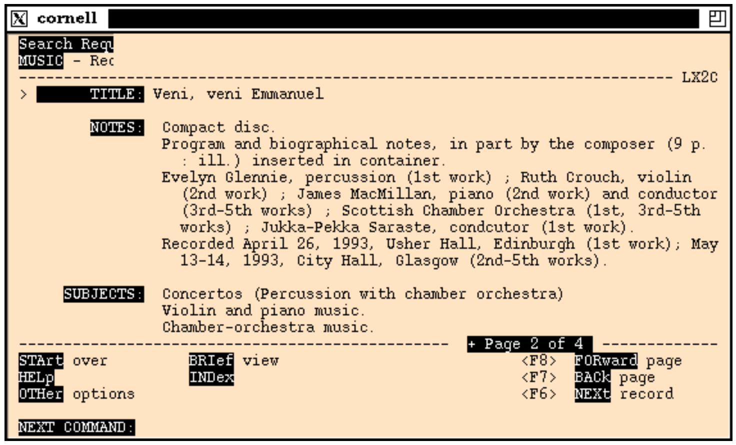
```
proc waitfornext {} {
  global cur_row cur_col term
  term_expect {expr {$cur_col==15 && \
                   $cur_row == 24 && \
                   " NEXT COMMAND:  " == [$term get 24.0 24.16]}} {}
}

proc sendcommand {command} {
  global cur_col
  exp_send $command
  term_expect {expr {$cur_col == 79}} {}
}
```

Now the interactions with the library are trivial. The remaining commands look for a book using the keywords *sound* and *scottish*. The first book is selected and its long form is displayed. Finally the next page of the long form is shown.

```
waitfornext
sendcommand "k=sound and scottish\r"
waitfornext
sendcommand "1\r"
waitfornext
sendcommand "lon\r"
waitfornext
sendcommand "for\r"
```

The view of the Tk terminal emulator, after these queries, is shown below:



A Tk-less implementation

The implementation shown above uses Tk. The drawback to Tk is that it requires the X Window System to be available. This might not be possible in some environments.

It is possible to perform terminal emulation and the expect operations without using Tk. The framework remains the same. The significant changes are:

- An array is used to maintain the screen representation instead of using a Tk text widget. Each line of the array models a line of the display. The functionality provided by the text widget object is similarly duplicated in an explicit Tcl procedure.
- The input stream is analyzed synchronously instead of asynchronously. In practical terms, this means that **`expect_before`** is used instead of **`expect_background`** to wait for character sequences. This simplifies the **`term_expect`** procedure. The strobes are no longer necessary since the waiting is explicit.

Availability

This software described in this paper is freely available. However, the author and NIST would appreciate credit if this software, documentation, ideas, or portions of them are used.

The scripts and programs described in this document may be ftp'd as pub/expect/expect.tar.Z from ftp.cme.nist.gov. The software will be mailed to you if you send the mail message "send pub/expect/expect.tar.Z" (without quotes) to library@cme.nist.gov.

Acknowledgments

Much of the development of Expect was funded by the NIST Scientific and Technical Research Services.

The Tk-less implementation described in this paper was done by Adrian Moriano, Cornell University. Adrian also wrote the script to interact with the Cornell University Library.

Thanks to Steve Ray, Josh Lubell, and Kathy Miles for proofreading this paper.

References

[ANSI90] *ANSI X3.64-1979 (R1990) - Additional Controls for Use with the American National Standards Code for Information Interchange*, ANSI, 1990.

[Good91] Berny Goodheart, *UNIX Curses Explained*, Prentice Hall, 1991.

[Lamb90] Linda Lamb, *Learning the vi Editor*, O'Reilly & Associates, Inc., ISBN 0-937175-67-6, October 1990.

[Libes90] Don Libes, "Expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, pp. 183-192, Anaheim, CA, June 11-15, 1990.

[Libes91] Don Libes, "Expect: Scripts for Controlling Interactive Programs", *Computing Systems*, pp. 99-126, Vol. 4, No. 2, University of California Press Journals, CA, Spring 1991.

[Libes95] Don Libes, *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, O'Reilly & Associates, Inc., pp. 602, ISBN 1-56592-090-2, January 1995.

[Ouster94] John K Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-63337-X, April 1994.

[POSIX94] *Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities*, Federal Information Processing Standards Publication 189, National Institute of Standards and Technology, October 11, 1994.

[Stall94] Richard Stallman, *GNU Emacs Manual*, Free Software Foundation, Inc., ISBN 1-88211404-3, July 1994.