Brief Table of Contents

Brief Table of Contents
Extended Table Of Contents
Preface
How To Read This Book
1. Introduction—What Is Expect? 1
2. Tcl—Introduction And Overview 23
3. Getting Started With Expect 71
4. Glob Patterns And Other Basics 87
5. Regular Expressions 107
6. Patterns, Actions, And Limits 129
7. Debugging Patterns And Controlling Output 165
8. Handling A Process And A User 185
9. The Expect Program 213
10. Handling Multiple Processes 233
11. Handling Multiple Processes Simultaneously 247
12. Send 271
13. Spawn 287
14. Signals 307
15. Interact 323
16. Interacting With Multiple Processes 349
17. Background Processing 371
18. Debugging Scripts 403
19. Expect + Tk = Expectk 429
20. Extended Examples 465
21. Expect, C, And C++ 491
22. Expect As Just Another Tcl Extension 513
23. Miscellaneous 523
   Appendix—Commands and Variables 531
   Index Of Scripts 537
   Index 541
# Extended Table Of Contents

**Brief Table of Contents** ............................ ix

**Extended Table Of Contents** ...................... xi

**Preface** .................................................. xxiii

Expect—Why Another Tool? ................................ xxiv
Tcl—A Little History ........................................ xxv
Acknowledgments ............................................ xxvii

**How To Read This Book** .............................. xxxi

Notational Conventions .................................. xxxii
Exercises .................................................... xxxiii

1: Introduction — What Is Expect? ................... 1

Ouch, Those Programs Are Painful! ................... 1
A Very Brief Overview .................................... 3
A First Script—dialback .................................. 3
Total Automation .......................................... 4
Differing Behavior When Running Non-Interactively .... 7
Partial Automation ........................................ 8
Dangerous, Unfriendly, Or Otherwise Unlikable User Interfaces ...... 9
Graphical Applications .................................... 10
A Little More About Tcl .................................. 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Control</td>
<td>12</td>
</tr>
<tr>
<td>Background Processes</td>
<td>12</td>
</tr>
<tr>
<td>Using Expect With Other Programs</td>
<td>13</td>
</tr>
<tr>
<td>Using Expect On UNIX</td>
<td>13</td>
</tr>
<tr>
<td>Using Expect On Other Operating Systems</td>
<td>13</td>
</tr>
<tr>
<td>Using Expect In Real Applications</td>
<td>14</td>
</tr>
<tr>
<td>Using Expect In Commercial Applications—Legalese</td>
<td>15</td>
</tr>
<tr>
<td>Obtaining Expect And The Examples</td>
<td>16</td>
</tr>
<tr>
<td>Expect And Tcl Resources</td>
<td>19</td>
</tr>
<tr>
<td>Exercises</td>
<td>22</td>
</tr>
</tbody>
</table>

2: *Tcl*—Introduction And Overview ........................................ 23

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everything Is A String</td>
<td>23</td>
</tr>
<tr>
<td>Quoting Conventions</td>
<td>25</td>
</tr>
<tr>
<td>Expressions</td>
<td>27</td>
</tr>
<tr>
<td>Braces—Deferring Evaluation</td>
<td>29</td>
</tr>
<tr>
<td>Control Structures</td>
<td>29</td>
</tr>
<tr>
<td>More On Expressions</td>
<td>37</td>
</tr>
<tr>
<td>Lists</td>
<td>38</td>
</tr>
<tr>
<td>More Ways To Manipulate Strings</td>
<td>46</td>
</tr>
<tr>
<td>Arrays</td>
<td>49</td>
</tr>
<tr>
<td>Indirect References</td>
<td>51</td>
</tr>
<tr>
<td>Handling Errors</td>
<td>53</td>
</tr>
<tr>
<td>Evaluating Lists As Commands</td>
<td>56</td>
</tr>
<tr>
<td>Passing By Reference</td>
<td>57</td>
</tr>
<tr>
<td>Working With Files</td>
<td>60</td>
</tr>
<tr>
<td>File I/O</td>
<td>61</td>
</tr>
<tr>
<td>Executing UNIX Commands</td>
<td>66</td>
</tr>
<tr>
<td>Environment Variables</td>
<td>67</td>
</tr>
<tr>
<td>Handling Unknown Commands</td>
<td>68</td>
</tr>
<tr>
<td>Libraries</td>
<td>68</td>
</tr>
<tr>
<td>Is There More To Tcl?</td>
<td>69</td>
</tr>
<tr>
<td>Exercises</td>
<td>70</td>
</tr>
</tbody>
</table>

3: Getting Started With Expect ............................................. 71

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The send Command</td>
<td>71</td>
</tr>
</tbody>
</table>
# Table of Contents

The expect Command .......................................................... 72  
Anchoring ........................................................................ 73  
What Happens When Input Does Not Match ......................... 74  
Pattern-Action Pairs ............................................................ 75  
Example—Timed Reads In The Shell ..................................... 77  
The spawn Command ............................................................ 78  
The interact Command .......................................................... 82  
Example—Anonymous ftp .................................................... 83  
Exercises ............................................................................ 85  

## 4: Glob Patterns And Other Basics ........................................ 87  
The * Wildcard .................................................................... 87  
More Glob Patterns ............................................................. 91  
Backslashes ......................................................................... 91  
Handling Timeout .................................................................. 94  
Handling End Of File (eof) ................................................... 98  
Hints On The spawn Command ........................................... 99  
Back To Eof ......................................................................... 100  
The close Command ............................................................. 101  
Programs That Ignore Eof .................................................... 103  
The wait Command ............................................................... 105  
Exercises ............................................................................ 106  

## 5: Regular Expressions .......................................................... 107  
Regular Expressions—A Quick Start ...................................... 107  
Identifying Regular Expressions And Glob Patterns ................ 109  
Using Parentheses To Override Precedence ......................... 110  
Using Parentheses For Feedback ........................................... 111  
More On The timed-read Script ............................................ 112  
Pattern Matching Strategy .................................................... 113  
Nested Parentheses .............................................................. 115  
Always Count Parentheses, Even Inside Of Alternatives .......... 116  
Example—The Return Value From A Remote Shell ................ 117  
Matching Customized Prompts ............................................. 120  
Example—a Smart Remote Login Script ............................... 122  
What Else Gets Stored In expect_out .................................... 124
More On Anchoring ................................................................. 125
Exercises ................................................................................. 126

6: Patterns, Actions, And Limits ............................................. 129
Matching Anything But ............................................................. 129
Really Complex Patterns ......................................................... 131
Really Simple Patterns ............................................................. 134
Matching One Line And Only One Line ............................... 135
Tcl's string match Command ................................................... 136
Tcl's regexp Command ............................................................ 137
Tcl's regsub Command ............................................................ 138
Ignoring Case ........................................................................... 139
All Those Other String Functions Are Handy, Too ............. 140
Actions That Affect Control Flow ......................................... 140
Example—rogue ..................................................................... 141
Character Graphics ................................................................. 142
More Actions That Affect Control Flow .............................. 143
Matching Multiple Times ....................................................... 145
Recognizing Prompts (Yet Again) .......................................... 147
Speed Is On Your Side ............................................................ 148
Controlling The Limits Of Pattern Matching Input .......... 149
The full_buffer Keyword ......................................................... 151
Double Buffering ................................................................... 152
Perpetual Buffering ................................................................. 154
The Politics Of Patterns .......................................................... 154
Expecting A Null Character ................................................... 155
Parity ....................................................................................... 157
Length Limits .......................................................................... 158
Comments In expect Commands ......................................... 158
Restrictions On expect Arguments ....................................... 159
eval—Good, Bad, And Ugly ................................................... 160
Exercises ................................................................................ 162

7: Debugging Patterns And Controlling Output ..................... 165
Pattern Debugging ................................................................. 165
Enabling Internal Diagnostics ................................................. 171
# Table of Contents

Logging Internal Diagnostics ..................................................... 173
Disabling Normal Program Output ............................................. 174
The log_user Command ............................................................. 175
Example—su2 ......................................................................... 178
Recording All Expect Output ..................................................... 180
Sending Messages To The Log .................................................... 182
About File Names .................................................................... 182
Log And Diagnostic State ......................................................... 182
Exercises ................................................................................. 183

8: Handling A Process And A User ............................................ 185
The send_user Command .......................................................... 185
The send_error Command ......................................................... 187
The expect_user Command ....................................................... 192
Dealing With Programs That Reprompt ..................................... 193
Dealing With Programs That Miss Input ................................... 196
Sleeping .................................................................................. 196
Line Versus Character–Oriented And Other Terminal Modes ....... 197
Echoing .................................................................................... 199
Prompting For A Password On Behalf Of A Program ................ 201
Security And Insecurity ............................................................ 202
Resetting The Terminal Upon Exit ........................................... 204
More On The stty Command ....................................................... 204
The system Command ............................................................... 207
Redirecting The Standard Input Or Output ............................... 209
The expect_tty Command .......................................................... 210
The send_tty Command ............................................................ 210
Exercises ................................................................................. 211

9: The Expect Program ............................................................ 213
Expect—Just Another Program .................................................. 213
Invoking Scripts Without Saying “expect” ................................. 215
Rewriting The #! Line ............................................................... 216
The .exp Extension .................................................................. 217
The — And Other Flags ............................................................ 217
The —c Flag .......................................................................... 218
The -f Flag ........................................................................................................220
Writing The #! Line ..........................................................................................221
The -i Flag .......................................................................................................221
The -n And -N Flags ........................................................................................221
The -d Flag .......................................................................................................223
The -D Flag .....................................................................................................223
The -b Flag .......................................................................................................224
The -l Flag .......................................................................................................224
The interpreter Command ...............................................................................225
Exercises ...........................................................................................................232

10: Handling Multiple Processes ................................................................. 233
The spawn_id Variable ....................................................................................233
Example—chess Versus chess ........................................................................234
Example—Automating The write Command ..................................................237
How exp_continue Affects spawn_id ..............................................................238
The Value Of spawn_id Affects Many Commands .........................................238
Symbolic Spawn Ids .......................................................................................239
Job Control ......................................................................................................240
Procedures Introduce New Scopes .................................................................241
How Expect Writes Variables In Different Scopes ........................................243
Predefined Spawn Ids ....................................................................................245
Exercises ...........................................................................................................246

11: Handling Multiple Processes Simultaneously ...................................... 247
Implicit Versus Explicit Spawn Ids ................................................................247
Waiting From Multiple Processes Simultaneously ........................................249
Example—Answerback ..................................................................................250
Which Pattern Goes With Which Spawn Id ....................................................252
Which Spawn Id Matched ..............................................................................253
Spawn Id Lists ................................................................................................ 254
Example—Connecting Together Two Users To An Application ....................255
Example—Timing All Commands ................................................................ 256
Matching Any Spawn Id Already Listed ..........................................................259
The expect_before And expect_after Commands ..........................................259
Indirect Spawn Ids ..........................................................................................268
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercises</td>
<td>270</td>
</tr>
<tr>
<td><strong>12: Send</strong></td>
<td></td>
</tr>
<tr>
<td>Implicit Versus Explicit Spawn Ids</td>
<td>271</td>
</tr>
<tr>
<td>Sending To Multiple Processes</td>
<td>272</td>
</tr>
<tr>
<td>Sending Without Echoing</td>
<td>273</td>
</tr>
<tr>
<td>Sending To Programs In Cooked Mode</td>
<td>274</td>
</tr>
<tr>
<td>Sending Slowly</td>
<td>275</td>
</tr>
<tr>
<td>Sending Humanly</td>
<td>278</td>
</tr>
<tr>
<td>Sending Nulls</td>
<td>281</td>
</tr>
<tr>
<td>Sending Breaks</td>
<td>281</td>
</tr>
<tr>
<td>Sending Strings That Look Like Flags</td>
<td>282</td>
</tr>
<tr>
<td>Sending Character Graphics</td>
<td>283</td>
</tr>
<tr>
<td>Comparing send To puts</td>
<td>283</td>
</tr>
<tr>
<td>Exercises</td>
<td>285</td>
</tr>
<tr>
<td><strong>13: Spawn</strong></td>
<td></td>
</tr>
<tr>
<td>The Search Path</td>
<td>287</td>
</tr>
<tr>
<td>Philosophy—Processes Are Smart</td>
<td>288</td>
</tr>
<tr>
<td>Treating Files As Spawned Processes</td>
<td>289</td>
</tr>
<tr>
<td>Opening Tys</td>
<td>290</td>
</tr>
<tr>
<td>Bugs And Workarounds</td>
<td>291</td>
</tr>
<tr>
<td>Process Pipelines And Pys</td>
<td>291</td>
</tr>
<tr>
<td>Automating xterm</td>
<td>293</td>
</tr>
<tr>
<td>Checking For Errors From spawn</td>
<td>296</td>
</tr>
<tr>
<td>spawn –noecho</td>
<td>298</td>
</tr>
<tr>
<td>Example—unbuffer</td>
<td>299</td>
</tr>
<tr>
<td>Obtaining Console Output</td>
<td>300</td>
</tr>
<tr>
<td>Setting Pty Modes From spawn</td>
<td>300</td>
</tr>
<tr>
<td>Hung Pys</td>
<td>302</td>
</tr>
<tr>
<td>Restrictions On Spawning Multiple Processes</td>
<td>303</td>
</tr>
<tr>
<td>Getting The Process Id From A Spawn Id</td>
<td>304</td>
</tr>
<tr>
<td>Using File I/O Commands On Spawned Processes</td>
<td>304</td>
</tr>
<tr>
<td>Exercises</td>
<td>305</td>
</tr>
</tbody>
</table>
14: Signals ................................................................. 307
Signals ........................................................................ 307
Signals In Spawned Processes ....................................... 310
Notes On Specific Signals ............................................ 311
When And Where Signals Are Evaluated ....................... 318
Overriding The Original Return Value ......................... 320
Using A Different Interpreter To Process Signals .............. 321
Exit Handling ............................................................. 321
Exercises ................................................................... 322

15: Interact ................................................................. 323
The interact Command ................................................ 323
Simple Patterns ........................................................ 324
Exact Matching .......................................................... 327
Matching Patterns From The Spawned Process ............... 328
Regular Expressions .................................................... 328
What Happens To Things That Do Not Match ................ 331
More Detail On Matching ............................................ 332
Echoing ...................................................................... 333
Avoiding Echoing ....................................................... 335
Giving Feedback Without –echo ................................... 335
Telling The User About New Features ......................... 336
Sending Characters While Pattern Matching .................. 337
The continue And break Actions .................................. 339
The return Action ....................................................... 339
The Default Action .................................................... 341
Detecting End-Of-File ................................................. 342
Matching A Null Character ......................................... 343
Timing Out .................................................................. 343
More On Terminal Modes (Or The –reset Flag) ............... 344
Example—Preventing Bad Commands ......................... 346
Exercises ................................................................... 347

16: Interacting With Multiple Processes ......................... 349
Connecting To A Process Other Than The Currently Spawned Process ........................................ 349
Connecting To A Process Instead Of The User ................ 350
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example—rz And sz Over rlogin ................................................. 351</td>
</tr>
<tr>
<td>Redirecting Input And Output ....................................................... 353</td>
</tr>
<tr>
<td>Default Input And Output .............................................................. 354</td>
</tr>
<tr>
<td>Controlling Multiple Processes—kibitz ............................................ 355</td>
</tr>
<tr>
<td>Combining Multiple Inputs Or Outputs ............................................ 358</td>
</tr>
<tr>
<td>Which Spawn Id Matched .................................................................. 359</td>
</tr>
<tr>
<td>Indirect Spawn Ids .......................................................................... 359</td>
</tr>
<tr>
<td>An Extended Example—xkibitz ............................................................ 361</td>
</tr>
<tr>
<td>Exercises ......................................................................................... 369</td>
</tr>
</tbody>
</table>

**17: Background Processing** ......................................................... 371

| Putting Expect In The Background .................................................. 371 |
| Running Expect Without A Controlling Terminal .............................. 372 |
| Disconnecting The Controlling Terminal ......................................... 373 |
| The fork Command ........................................................................ 374 |
| The disconnect Command ................................................................ 375 |
| Reconnecting ................................................................................ 378 |
| Using kibitz From Other Expect Scripts ......................................... 380 |
| Mailing From Expect ...................................................................... 383 |
| A Manager For Disconnected Processes—dislocate ............................ 384 |
| Expect As A Daemon ....................................................................... 392 |
| Example—Automating Gopher And Mosaic telnet Connections ............ 397 |
| Exercises ......................................................................................... 401 |

**18: Debugging Scripts** ................................................................. 403

| Tracing ......................................................................................... 403 |
| Logging ......................................................................................... 404 |
| Command Tracing ......................................................................... 405 |
| Variable Tracing ......................................................................... 406 |
| Example—Logging By Tracing ......................................................... 407 |
| UNIX System Call Tracing .............................................................. 408 |
| Tk And tkinspect .......................................................................... 409 |
| Traditional Debugging ................................................................... 410 |
| Debugger Command Overview And Philosophy ................................ 412 |
| Stepping Over Procedure Calls ...................................................... 413 |
| Stepping Into Procedure Calls ....................................................... 415 |
19: Expect + Tk = Expectk ......................................................... 429
Tk—A Brief Technical Overview .................................................. 430
Expectk ................................................................................. 432
The send Command ................................................................. 433
An Extended Example—tks passwd ............................................. 434
Using Tk Widgets To Prompt For Passwords .............................. 444
The expect Command And The Tk Event Loop ............................. 445
The expect_background Command ............................................. 446
Multiple Spawn Ids In expect_background ................................. 447
Background Actions ................................................................ 447
Example—A Dumb Terminal Emulator ....................................... 447
Example—A Smarter Terminal Emulator .................................... 448
Using The Terminal Emulator For Testing And Automation ....... 458
Exercises ............................................................................... 462

20: Extended Examples ............................................................... 465
Encrypting A Directory .............................................................. 465
File Transfer Over telnet ........................................................... 467
You Have Unread News—tknewsbiff ........................................ 475
Exercises ............................................................................... 489

21: Expect, C, And C++ .............................................................. 491
Overview ............................................................................... 492
Linking .................................................................................... 493
Include Files .......................................................................... 493
# Table of Contents

Prys And Processes ................................................................. 494
Allocating Your Own Pty ...................................................... 498
Closing The Connection To The Spawned Process ..................... 499
Expect Commands ................................................................. 500
Regular Expression Patterns .................................................. 502
Exact Matching ...................................................................... 504
Matching A Null ..................................................................... 504
What Characters Matched ...................................................... 504
When The Number Of Patterns Is Not Known In Advance .......... 506
Expecting From Streams ....................................................... 507
Running In The Background ................................................... 507
Handling Multiple Inputs And More On Timeouts ..................... 508
Output And Debugging Miscellany .......................................... 509
Pty Trapping ......................................................................... 510
Exercises ............................................................................. 510

22: Expect As Just Another Tcl Extension ................................. 513
Adding Expect To Another Tcl-based Program ......................... 513
Differences Between Expect And The Expect Extension In Another Program .................................................. 515
Adding Extensions To Expect .................................................. 516
Adding Extensions To Expectk ............................................... 517
Creating Scriptless Expect Programs ...................................... 518
Functions And Variables In The Expect Extension .................. 518
Exercises ............................................................................. 522

23: Miscellaneous .................................................................. 523
Random Numbers ................................................................... 523
Example—Generating Random Passwords ............................... 524
The Expect Library .................................................................. 526
Expect Versions ..................................................................... 527
Timestamps ............................................................................ 528
The time Command .................................................................. 529
Exercises ............................................................................. 530

Appendix—Commands and Variables ....................................... 531
Commands And Flags .............................................................. 531
Variables ........................................................................................................535

Index Of Scripts ..........................................................................................537

Index .............................................................................................................541
Expect is a program to control interactive applications. These applications interactively prompt and expect a user to enter keystrokes in response. By using Expect, you can write simple scripts to automate these interactions. And using automated interactive programs, you will be able to solve problems that you never would have even considered before.

Expect can save you hours of drudgery, and this book will start you on your way. This first chapter is an overview of Expect. I will describe some simple applications and present some short scripts. However, the explanations are not intended to be complete but rather to whet your appetite and give you a taste of the good things to come. In the following chapters, I will revisit all of the concepts mentioned in this chapter and cover them in more detail.

**Ouch, Those Programs Are Painful!**

`fsck`, the UNIX file system check program, can be run from a shell script only with the `-y` or `-n` options. The manual defines the `-y` option as follows:

> Assume a yes response to all questions asked by `fsck`; this should be used with extreme caution, as it is a free license to continue, even after severe problems are encountered.

The `-n` option is safer, but almost uselessly so. This kind of interface is inexcusably bad, yet many programs have the same style. `ftp`, a file transfer program, has an option that disables interactive prompting so that it can be run from a script. But it provides no way to take alternative action should an error occur.
Expect is a tool for controlling interactive programs. It solves the \texttt{fsck} problem, providing all the interactive functionality non-interactively. Expect is not specifically designed for \texttt{fsck} and can handle errors from \texttt{ftp} as well.

The problems with \texttt{fsck} and \texttt{ftp} illustrate a major limitation in the user interface offered by shells such as \texttt{sh}, \texttt{csh}, and others (which I will generically refer to as "the shell" from now on). The shell does not provide a way of reading output from and writing input to a program. This means the shell can run \texttt{fsck}, but only by missing out on some of its useful features. Some programs cannot be run at all from a shell script. For example, \texttt{passwd} cannot be run without a user interactively supplying the input. Similar programs that cannot be automated in a shell script are \texttt{telnet}, \texttt{crypt}, \texttt{su}, \texttt{rlogin}, and \texttt{gdb}. A large number of application programs are written with the same fault of demanding user input.

Expect was designed specifically to interact with \textit{interactive} programs. An Expect programmer can write a script describing the dialogue. Then the Expect program can run the "interactive" program non-interactively. Expect can also be used to automate only parts of a dialogue, since control can be passed from the script to the keyboard and vice versa. This allows a script to do the drudgery and a user to do the fun stuff.
A Very Brief Overview

Expect programs can be written in C or C++, but are almost always written using Tcl.† Tcl is an interpreted language that is widely used in many other applications. If you already use a Tcl-based application, you will not have to learn a new language for Expect.

Tcl is a very typical-looking shell-like language. There are commands to set variables (set), control flow (if, while, foreach, etc.), and perform the usual math and string operations. Of course, UNIX programs can be called (exec). I will provide a quick introduction to the language in Chapter 2 (p. 23).

Expect is integrated on top of Tcl and provides additional commands for interacting with programs. Expect is named after the specific command that waits for output from a program. The expect command is the heart of the Expect program. The expect command describes a list of patterns to watch for. Each pattern is followed by an action. If the pattern is found, the action is executed.

For example, the following fragment is from a script that involves a login. When executed, the script waits for the strings “welcome”, “failed”, or “busy”, and then it evaluates one of the corresponding actions. The action associated with busy shows how multiple commands can be evaluated. The timeout keyword is a special pattern that matches if no other pattern matches in a certain amount of time.

```tcl
expect {
    *welcome* break
    *failed* abort
    timeout abort
    *busy* {
        puts *busy*
        continue
    }
}
```

A First Script — dialback

It is surprising how little scripting is necessary to produce something useful. Below is a script that dials a phone. It is used to reverse the charges so that long-distance phone calls are charged to the computer. It is invoked with the phone number as its argument.

---

† Tcl is pronounced “tickle”. Some people are uncomfortable using this particular word in a formal setting. In that case, I recommend either saying the letters (i.e., “tee cee ell”) or coming up with a name suitable to your audience (e.g., “macho stud language”). Whatever works for you.
spawn tip modem
expect "connected"
send "ATD$argv\r"
# modem takes a while to connect
set timeout 60
expect "CONNECT"

The first line runs the tip program so that the output of a modem can be read by expect and its input written by send. Once tip says it is connected, the modem is told to dial using the command ATD followed by the phone number. The phone number is retrieved from argv, which is a variable predefined to contain the original argument with which the script was called.

The fourth line is just a comment noting that the variable being set in the next line controls how long expect will wait before giving up. At this point, the script waits for the call to complete. No matter what happens, expect terminates. If the call succeeds, the system detects that a user is connected and prompts with "login:".

Actual scripts do more error checking, of course. For example, the script could retry if the call fails. But the point here is that it does not take much code to produce useful scripts. This six-line script replaced a 60Kb executable (written in C) that did the same thing!

In Chapter 16 (p. 351), I will talk more about the dialback concept and show a different way to do it.

**Total Automation**

Earlier I mentioned some programs that cannot be automated with the shell. It is difficult to imagine why you might even want to embed some of these programs in shell scripts. Certainly the original authors of the programs did not conceive of this need. As an example, consider passwd.

passwd is the command to change a password. The passwd program does not take the new password from the command line. Instead, it interactively prompts for it—twice. Here is what it looks like when run by a system administrator. (When run by users, the interaction is slightly more complex because they are prompted for their old passwords as well.)

```
# passwd libes
Changing password for libes on thunder.
New password:
Retype new password:
```

This is fine for a single password. But suppose you have accounts of your own on a number of unrelated computers and you would like them all to have the same
Tcl—Introduction And Overview

Expect does not have its own special-purpose language. Expect uses Tcl, a popular language for embedding in applications. Tcl provides lots of basic commands such as if/then/else, while, and set. Expect extends the language with commands such as expect and interact.

This chapter is an introduction and overview of Tcl. While not covering all of Tcl, this chapter does provide everything that the rest of the book depends on, and this is enough to write virtually any Expect script. Even if you already know Tcl, you may find it helpful to read this chapter. In this chapter, I will emphasize things about Tcl that you may not have thought much about before.

You probably want to get on with using Expect, and I can understand the urge to skip this chapter in the hopes of learning as little Tcl as possible so you can put Expect to work for you now. Please be patient and it will all fit together that much more easily.

If you do skip this chapter and you find yourself wondering about points in the other chapters, turn back to this chapter and read it.

A few concepts will not be covered here but will be explained as they are encountered for the first time in other chapters. The index can help you locate where each command is first defined and used.

I will occasionally mention when a particular Tcl command or feature is similar to C. It is not necessary that you know C in order to use Tcl, but if you do know it, such statements are clues that you can rely on what you already know from that language.

Everything Is A String

The types of variables are not declared in Tcl. There is no need since there is only one type: string. Every value is a string. Numbers are strings. Even commands and variables
are strings! The following commands set the variable name to the string value "Don", the variable word to the value "foobar", and the variable pi to "3.14159".

    set name Don
    set word foobar
    set pi 3.14159

Variable names, values, and commands are case sensitive. So the variable "name" is different than "Name".

To access a variable's value, prefix the variable name with a dollar sign ($). The following command sets the variable phrase to foobar by retrieving it from the variable word.

    set phrase $word

Variable substitutions can occur anywhere in a command, not just at the beginning of an argument. The following command sets the variable phrase2 to the string "word=foobar".

    set phrase2 word=$word

You can insert a literal dollar sign by prefixing it with a backslash. The following command sets the variable money to the value "$1000".

    set money \$1000

The backslash is also a useful way to embed other special characters in strings. For example, "\t" is a tab and "\b" is a backspace. Most of the Standard C backslash conventions are supported including \ followed by one to three octal digits and \x followed by any number of hex digits.† I will mention more of these escapes later.

    # stick control-Z in a variable
    set controlZ \032
    # define control-C
    set controlC \x03
    # define string with embedded control-Z and tab
    set oddword foo\032bar\tgorp

A command beginning with "#" is a comment. It extends to the end of the line. You can think of "#" as a command whose arguments are discarded.

Multiple commands can be strung together if they are separated by a semicolon. A literal semicolon can be embedded by prefacing it with a backslash.

    set word1 foo; set word2 bar    ;# two commands
    set word3 foo\;bar             ;# one command

† The use of \0 by itself to represent the null character is the only escape not supported. I will describe how to handle nulls in Chapter 6 (p. 155).
The "; #" sequence above is a common way of a tacking comments on the end of a line. The ";" ends the previous command and the "#" starts the comment. Writing the "; #" together avoids the possibility of having your comment unintentionally accepted as additional arguments because of a forgotten semicolon.

Commands are normally terminated at the end of a line, but a backslash at the end of a line allows multi-line commands. The backslash-newline sequence and any following whitespace behaves as if it were a single space. Later, I will show other ways of writing multi-line commands.

    set word \
    really-long-string-which-does-not-quite-fit-on-previous-line

**Quoting Conventions**

Tcl separates arguments by whitespace (blanks, tabs, etc.). You can pass space characters by double quoting an argument. The quotes are not part of the argument; they just serve to keep it together. Tcl is similar to the shell in this respect.

    set string1 "Hello world"

Double quotes prevent ";" from breaking things up.

    set string2 "A semicolon ; is in here"

Keeping an argument together is all that double quotes do. Character sequences such as $, \t, and \b still behave as before.

    set name "Sam"
    set age 17
    set word2 "My name is $name; my age is $age;"

After execution of these three commands, word2 is left with the value "My name is Sam; my age is 17;".

Notice that in the first command Sam was quoted, while in the second command 17 was not quoted, even though neither contained blanks. When arguments do not have blanks, you do not have to quote them but I often do anyway—they are easier to read. However, I do not bother to quote numbers because quoted numbers simply look funny. Anyway, numbers never contain whitespace.

You can actually have whitespace in command names and variable names in which case you need to quote them too. I will show how to do this later, but I recommend you avoid it if possible.
**Return Values**

All commands return values. For example, the `pid` command returns the process id of the current process. To evaluate a command and use its return value as an argument or inside an argument, embed the command in brackets. The bracketed command is replaced by its return value.

```
set p "My pid is [pid]."
```

The `set` command returns its second argument. The following command sets `b` and `a` to 0.

```
set b "[set a 0]"
```

When it comes to deciding what are arguments, brackets are a special case. Tcl groups everything between matching brackets, so it is not necessary to quote arguments that already have all their whitespace enclosed entirely within brackets. The following are all legal.

```
set b [set a 0]
set b "[set a 0]"
set b "[set a 0]hello world"
set b [set a 0]hello
```

After execution of this last command, `a` is set to "0" and `b` is set to "0hello".

With only one argument, `set` returns the value of its first argument.

```
set c [set b]
```

Calling `set` with one argument is similar to using the dollar sign. Indeed, the previous command can be rewritten as `set c $b`. However they are not always interchangeable. Consider the two commands:

```
set $phello
set [set p]hello
```

The first returns the value of the variable `p_hello`. The second returns the value of the variable `p` concatenated with the string "hello".

The $ syntax is shorter but does not automatically terminate the end of the variable. In the rare cases where the variable just runs right into more alphanumeric characters, the one-argument `set` command in brackets is useful.

The one-argument `set` command is also useful when entering commands interactively. For example, here is what it looks like when I type a command to the Tcl interpreter:‡

‡ When Tcl is installed, it creates a program called `tclsh` (which stands for "Tcl shell" but is usually pronounced "ticklish"). `tclsh` is a program that contains only the Tcl commands and interpreter. Typing directly to `tclsh` is not the usual way to use Tcl, but `tclsh` is convenient for experimenting with the basic Tcl commands. `tclsh` actually prompts with a bare "$", but I show it here as "tclsh> " so that you cannot confuse it with the C-shell prompt.

...
Three commands are central to the power of Expect: send, expect, and spawn. The send command sends strings to a process, the expect command waits for strings from a process, and the spawn command starts a process.

In this chapter, I will describe these commands and another one that is very useful: interact. To best understand this chapter, it will help to have some basic familiarity with Tcl. If you are wondering about a command that is not explained, look it up in the index for a reference in the previous chapter and read about it there.

**The send Command**

The send command takes a string as an argument and sends it to a process. For example:

```
send "hello world"
```

This sends the string "hello world" (without the quotes). If Expect is already interacting with a program, the string will be sent to that program. But initially, send will send to the standard output. Here is what happens when I type this to the Expect interpreter interactively:

```
expect1.1> send "hello world"
hello world
```

The `send` command does not format the string in any way, so after it is printed the next Expect prompt gets appended to it without any space. To make the prompt appear on a different line, put a newline character at the end of the string. A newline is represented by "\n".
expect1.1> send "hello world\n"
hello world
expect1.2>

If these commands are stored in a file, speak, the script can be executed from the UNIX command line:

```
% expect speak
hello world
%
```

With a little magic it is possible to invoke the file as just “speak” rather than “expect speak”. On most systems it suffices to insert the line “#!/usr/local/bin/expect” and say “chmod +x speak; rehash”. I will explain this in more detail in Chapter 9 (p. 215). For now, just take it on faith.

The expect Command

expect is the opposite of send. The expect command waits for a response, usually from a process. expect can wait for a specific string, but more often expect is used to wait for any string that matches a given pattern. Analogous to send, the expect command initially waits for characters from the keyboard. Using them, I can create a little conversation:

```
expect "hi\n"
send "hello there!\n"
```

When run, the interaction looks like this:

```
hi
hello there!
```

I typed the string hi and then pressed return. My input matched the pattern “hi\n”. Ideally, a return would be matched with “\r”; however, the UNIX terminal driver translates a return to “\n”. As you will see later on, it is rarely necessary to have to worry about this mapping because most of Expect’s interactions occur with programs not users, anyway. Nonetheless, it is occasionally useful to expect input from people. Plus, it is much easier to experiment with Expect this way.

If expect reads characters that do not match the expected string, it continues waiting for more characters. If I had typed hello followed by a return, expect would continue to wait for “hi\n”.  

---

† It actually reads from standard input which is typically the keyboard. For now, I will treat them as if they were the same thing.
‡ You can disable this behavior by saying "stty -icrnl" to the shell, but most programs expect this mapping to take place so learn to live with it.
When the matching string is finally typed, `expect` returns. But before returning, `expect` stores the matched characters in a variable called `expect_out(0,string)`. All of the matched characters plus the characters that came earlier but did not match are stored in a variable called `expect_out(buffer)`. `expect` does this every time it matches characters. The names of these variables may seem odd, but they will make more sense later on.

Imagine the following script:

```plaintext
  expect "hi\n"
  send "you typed <$expect_out(buffer)>"
  send "but I only expected <$expect_out(0,string)>"
```

The angle brackets do not do anything special. They will just appear in the output, making it clear where the literal text stops and the variable values start. When run the script looks like this:

```
Nice weather, eh?
  hi
  you typed <Nice weather, eh?
  hi>
  but I only expected <hi>
```

I typed "Nice weather, eh?" <return> "hi" <return>. `expect` reported that it found the `hi` but it also found something unexpected: "Nice weather, eh?\n".

**Anchoring**

Finding unexpected data in the input does not bother `expect`. It keeps looking until it finds something that matches. It is possible to prevent `expect` from matching when unexpected data arrives before a pattern. The caret (^) is a special character that only matches the beginning of the input. If the first character of the pattern is a caret, the remainder of the pattern must match starting at the beginning of the incoming data. It cannot skip over characters to find a valid match. For example, the pattern `^hi` matches if I enter "hiccup" but not if I enter "sushi".

The dollar sign ($) is another special character. It matches the end of the data. The pattern `hi$` matches if I enter "sushi" but not if I enter "hiccup". And the pattern `^hi$` matches neither "sushi" nor "hiccup". It matches "hi" and nothing else.

Patterns that use the ^ or $ are said to be **anchored**. Some programs, such as `sed`, define anchoring in terms of the beginning of a line. This makes sense for `sed`, but not for `expect`. `expect` anchors at the beginning of whatever input it has received without regard to line boundaries.
When patterns are *not* anchored, patterns match beginning at the earliest possible position in the string. For example, if the pattern is hi and the input is philosophic, the hi in philo is matched rather than the hi in sophic. In the next section, this subtlety will become more important.

**What Happens When Input Does Not Match**

Once `expect` has matched data to a pattern, it moves the data to the `expect_out` array as I showed earlier. The matched data is no longer eligible to be matched. Additional matches can only take place with new data.

Consider the following fragment:

```plaintext
expect "hi"
send "$expect_out(0,string) $expect_out(buffer)"
```

If I execute these two commands, Expect waits for me to enter hi. If I enter philosophic followed by a return, Expect finds the hi and prints:

```
hi phi
```

If I execute the two commands again, Expect prints:

```
hi losophi
```

Even though there were two occurrences of hi, the first time `expect` matched the first one, moving it into `expect_out`. The next `expect` started from where the previous one had left off.

With simple patterns like these, `expect` always stops waiting and returns immediately after matching the pattern. If `expect` receives more input than it needs, that input is remembered for the possibility of matching in later `expect` commands. In other words, `expect` buffers its input. This allows `expect` to receive input before it is actually ready to use it. The input will be held in an `input buffer` until an `expect` pattern matches it. This buffer is internal to `expect` and is not accessible to the script in any way except by matching patterns against it.

After the second `expect` above, the buffer must hold c\n. This is all that was left after the second hi in philosophic. The \n is there, of course, because after entering the word, I pressed return.

What happens if the commands are run again? In this case, `expect` is not going to find anything to match hi. The `expect` command eventually *times out* and returns. By default, after 10 seconds `expect` gives up waiting for input that matches the pattern. This ability to give up waiting is very useful. Typically, there is some reasonable amount of time to wait for input after which there is no further point to waiting. The choice of 10 seconds is good for many tasks. But there is no hard rule. Programs almost never
In This Chapter
• Glob (Shell-style) Patterns
• Keyword Patterns: eof, timeout
• Ending Processes
• Ending Scripts

Glob Patterns And Other Basics

In the last chapter, I showed some simple patterns that allow you to avoid having to specify exactly what you want to wait for. In this chapter, I will describe how to use patterns that you are already probably familiar with from the shell—glob patterns. I will also describe what happens when patterns do not match. I will go over some other basic situations such as how to handle timeouts. Finally I will describe what to do at the ends of scripts and processes.

The * Wildcard

Suppose you want to match all of the input and the only thing you know about it is that hi occurs within it. You are not sure if there is more to it, or even if another hi might appear. You just want to get it all. To do this, use the asterisk (*). The asterisk is a wildcard that matches any number of characters. You can write:

    expect "hi**
    send "$expect_out(0,string) $expect_out(buffer)"

If the input buffer contained "philosophic\n", expect would match the entire buffer. Here is the output from the previous commands:

    hilosophic
    philosophic

The pattern hi matched the literal hi while the * matched the string "losophic\n". The first p was not matched by anything in the pattern so it shows up in expect_out(buffer) but not in expect_out(0,string).
Earlier I said that * matches any number of characters. More precisely, it matches the longest string possible while still allowing the pattern itself to match. With the input buffer of "philosophic\n", compare the effects of the following two commands:

```plaintext
expect *hi**
expect *hi*hi*
```

In the first one, the * matches losophic\n. This is the longest possible string that the * can match while still allowing the hi to match hi. In the second expect, the * only matches losop, thereby allowing the second hi to match. If the * matched anything else, the entire pattern would fail to match.

What happens with the following command in which there are two asterisks?

```plaintext
expect **hi**
```

This could conceivably match in two ways corresponding to the two occurrences of "hi" in the string.

<table>
<thead>
<tr>
<th>possibility (1)</th>
<th>* matches</th>
<th>hi matches</th>
<th>* matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>philosop</td>
<td>hi</td>
<td>c\n</td>
<td></td>
</tr>
<tr>
<td>possibility (2)</td>
<td>p</td>
<td>hi</td>
<td>losophic\n</td>
</tr>
</tbody>
</table>

What actually happens is possibility (1). The first * matches philosop. As before, each * tries to match the longest string possible while allowing the total pattern to match, but the *'s are matched from left to right. The leftmost *'s match strings before the rightmost *'s have a chance. While the outcome is the same in this case (that is, the whole pattern matches), I will show cases later where it is necessary to realize that pattern matching proceeds from left to right.

**At The Beginning Of A Pattern Is Rarely Useful**

Patterns match beginning at the earliest possible character in a string. In Chapter 3 (p. 74), I showed how the pattern hi matched the first hi in philosophic. However, in the example above, the subpattern hi matched the second hi. Why the difference?

The difference is that hi was preceded by "*". Since the * is capable of matching anything, the leading * causes the match to start at the beginning of the string. In contrast, the earliest point that the bare hi can match is the first hi. Once that hi has matched, it cannot match anything else—including the second hi.

In practice, a leading * is usually redundant. Most patterns have enough literal letters that there is no choice in how the match occurs. The only remaining difference is that the leading * forces the otherwise unmatched leading characters to be stored in
expect_out(0,string). However, the characters will already be stored in expect_out(buffer) so there is little merit on this point alone.†

* At The End Of A Pattern Can Be Tricky

When a * appears at the right end of a pattern, it matches everything left in the input buffer (assuming the rest of the pattern matches). This is a useful way of clearing out the entire buffer so that the next expect does not return a mishmash of things that were received previously and things that are brand new.

Sometimes it is even useful to say:

    expect *

Here the * matches anything. This is like saying, "I don't care what's in the input buffer. Throw it away." This pattern always matches, even if nothing is there. Remember that * matches anything, and the empty string is anything! As a corollary of this behavior, this command always returns immediately. It never waits for new data to arrive. It does not have to since it matches everything.

In the examples demonstrating * so far, each string was entered by a person who pressed return afterwards. This is typical of most programs, because they run in what is called cooked mode. Cooked mode includes the usual line-editing features such as backspace and delete-previous-word. This is provided by the terminal driver, not the program. This simplifies most programs. They see the line only after you have edited it and pressed return.

Unfortunately, output from processes is not nearly so well behaved. When you watch the output of a program such as ftp or telnet (or cat for that matter), it may seem as if lines appear on your screen as atomic units. But this is not guaranteed. For example, in the previous chapter, I showed that when ftp starts up it looks like this:

```
% ftp ftp.uu.net
Connected to ftp.uu.net.
Name (ftp.uu.net:don):
```

Even though the program may have printed "Connected to ftp.uu.net.\n" all at once—perhaps by a single printf in a C program—the UNIX kernel can break this into small chunks, spitting out a few characters each time to the terminal. For example, it might print out "Conn" and then "ecte" and then "d to" and so on. Fortunately, computers are so fast that humans do not notice the brief pauses in the middle of

† The more likely reason to see scripts that begin many patterns with "*" is that prior to Expect version 4, all patterns were anchored, with the consequence that most patterns required a leading "*".
output. The reason the system breaks up output like this is that programs usually produce characters faster than the terminal driver can display them. The operating system will obligingly wait for the terminal driver to effectively say, "Okay, I've displayed that last bunch of characters. Send me a couple more." In reality, the system does not just sit there and wait. Since it is running many other programs at the same time, the system switches its attention frequently to other programs. Expect itself is one such "other program" in this sense.

When Expect runs, it will immediately ask for all the characters that a program produced only to find something like "Conn". If told to wait for a string that matches "Name*: ", Expect will keep asking the computer if there is any more output, and it will eventually find the output it is looking for.

As I said, humans are slow and do not notice this chunking effect. In contrast, Expect is so fast that it is almost always waiting. Thus, it sees most output come as chunks rather than whole lines. With this in mind, suppose you wanted to find out the version of ftp that a host is using. By looking back at the output, you can see that it is contained in the greeting line that begins "220" and ends with "ready.". Naively, you could wait for that line as:

```expect ^220$```

If you are lucky, you might get the entire line stored in $expect_out(0,string). You might even get the next line in there as well. But more likely, you will only get a fragment, such as "220 f" or "220 ftp.UU.VE". Since the pattern 220* matches either of these, expect has no reason to wait further and will return. As I stated earlier, expect returns with whatever is the longest string that matches the pattern. The problem here is that the remainder of the line may not have shown up yet!

If you want to get the entire line, you must be more specific. The following pattern works:

```*220*ready.*```

By specifying the text that ends the line, you force expect to wait for the entire line to arrive. The "." is not actually needed just to find the version identifier. You could just make the pattern:

```*220*re*```

Leaving off the e would be too short. This would allow the pattern to match the r in server rather than ready. It is possible to make the overall pattern even shorter by looking for more unusual patterns. But quite often you trade off readability. There is an art to choosing patterns that are correct, yet not too long but still readable. A good guideline is to give more priority to readability. The pattern matching performed by Expect is very inexpensive.
The previous chapter described glob patterns. You were probably familiar with them from the shell. Glob patterns are very simple and are sufficient for many purposes. Hence they are the default style of pattern matching in `expect` commands. However, their simplicity brings with it limitations.

For example, glob patterns cannot match any character `not` in a list of characters, nor can glob patterns match a choice of several different strings. Both of these turn out to be fairly common tasks. And while both can be simulated with a sequence of several other commands, Expect provides a much more powerful and concise mechanism: regular expressions.

**Regular Expressions—A Quick Start**

In order to jumpstart your knowledge of regular expressions (`regexp` for short), I will start out by noting the similarities. As the following table of examples shows, every glob pattern is representable by a regular expression. In contrast, some regular expressions cannot be represented as glob patterns.

For example, both the glob pattern `foo` and the regular expression `foo` match the literal string "foo". Backslash works in the usual way, turning the following character into its literal equivalent. "^" and "$" also work the same way as before. Regular expression ranges work as before, plus they can also be used to match any character `not` in the range by placing a "^" immediately after the left bracket. (I will show more detail on this later.) Besides this, the only significant differences in the table are the last two lines which describe how to match any single character and any number of any characters.

Except for "., *, each of the patterns in the table is called an *atom*. A * appended to an atom creates a pattern that matches any number (including zero) of the particular atom.
Table 5.1. Comparison of glob patterns and regular expressions.

<table>
<thead>
<tr>
<th>glob</th>
<th>regexp</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>literal s</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>literal *</td>
</tr>
<tr>
<td>^</td>
<td>^</td>
<td>beginning of string</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>end of string</td>
</tr>
<tr>
<td>[a-z]</td>
<td>[a-z]</td>
<td>any character in the range a to z</td>
</tr>
<tr>
<td>[^a-z]</td>
<td></td>
<td>any character not in the range a to z</td>
</tr>
<tr>
<td>?</td>
<td>.</td>
<td>any single character</td>
</tr>
<tr>
<td>*</td>
<td>.</td>
<td>any number of characters</td>
</tr>
</tbody>
</table>

For example, the regular expression “a*” matches any string of a’s, such as “a”, “aa”, “aaaaaa” and “”. That last string has no a’s in it at all. This is considered a match of zero a’s.

The pattern [0-9]* matches strings made up of integers such as “012” and “888”. Notice that the atom does not have to match the same literal value each time. When matching “012”, the range “[0-9]” first matches “0”, then it matches “1”, and finally matches “2”.

You can use ranges to construct more useful patterns. For example [1-9][0-9]* matches any positive integer. The first atom matches the first digit of the number, while the remaining digits are matched by the “[0-9]*”.

C language identifiers can be matched with the pattern “[a-zA-Z_][a-zA-Z0-9_]*”. This is similar to the previous pattern. In both cases the first character is restricted to a subset of the characters that can be used in the remaining part of the string.

In both cases the * only applies to the immediately preceding range. That is because the * only applies to the immediately preceding atom. One range is an atom; two ranges are not an atom.

Atoms by themselves and atoms with a * appended to them are called pieces. Pieces can also consist of atoms with a + or ? appended. An atom followed by + matches a sequence of one or more matches of the atom. An atom followed by ? matches the atom or the empty string. For example, “a+” matches “a” and “aa” but not “”, while “a?” matches “a” and “” but not “aa”. The pattern “0x[0-9a-fA-F]+” matches a hexadecimal number in the C language such as “0x0b2e” or “0xfffff”. The pattern “-?[1-9][0-9]*” matches positive or negative integers such as 1, 10, 1000, -1, and -1000. Notice how the [1-9] range prevents a zero from being the first digit, avoiding strings like -05 and 007.
In this chapter, I will describe the limits of patterns, including what to do when you hit them. I will also cover the darker side of range patterns—matching anything but certain characters. In the second half of the chapter, I will go into more detail on pattern actions including flow control. Finally, I will cover some miscellaneous pattern matching issues that do not fit anywhere else.

**Matching Anything But**

As I said in Chapter 5 (p. 113), pattern pieces match as many characters as possible. This makes it a little tricky to match a single line, single word, or single anything. For example, the regular expression `.*
` matches a single line, but it also matches two lines because two lines end with a `\n`. Similarly, it matches three lines, four lines, and so on. If you want to read lines one at a time from another program, then you cannot use this kind of pattern. The solution is to use the `^`

In Chapter 3 (p. 73), I showed that the `^` matches the beginning of the input buffer. When `^` is the first character of a regular-expression range, it means **match anything but the given characters**. For example, the regular expression `[^ab]` matches any character except a or b. The pattern `[^a-zA-Z]` matches any character but a letter.

A range can be used to build larger patterns. The pattern `[^ ]*` matches the longest string not including a blank. For example, if the input buffer contained "For example, if the input buffer contained ", the following expect command could be called repeatedly to match each word in the input.

---

† To match any character but a `^`, use the pattern `[^ ]`. To match a `^` outside a range, quote it with a backslash. To match a `^` inside a range, put it in any position of the range but the first. To match any character but a `^`, use the pattern `[^^]`.

---
expect -re "\([^ ]*\)" *

The range matches each word and the result is stored in $expect_out(1,string). The space at the end of the word is matched explicitly. Without the explicit space, the input buffer is left beginning with a space ("cow jumped ...") and subsequent matches return the null string before the first space.

Remember that the length of the match is important, but only after the starting position is taken into account. Patterns match the longest string at the first possible position in the input. In this example, 0 characters at column 0 are successfully matched even though the pattern can also match 3 characters at column 1. Because column 0 is before column 1, the earlier match is used.

There is no explicit means to match later matches than earlier ones, but often it is possible to simply pick a more descriptive pattern. In this example, the space can be skipped over. Alternatively, the * can be replaced by a + to force the pattern to be at least one letter. This effectively skips over the space between the words without the need to explicitly match it.

\[ expect -re "\[^ ]+" \]

Now the word is stored in "expect_out(0,string)". Because the pattern does not match whitespace, there is no need to select pieces of it, and the parentheses are no longer needed, simplifying the pattern further.

Here is the opening dialogue greeting from Uunet’s SMTP server. SMTP is the mail protocol used by most Internet computers. The server is normally controlled by a mail program to transfer mail from one host to another, but you can telnet to it directly and type commands interactively. The telnet program adds the first three lines, and Uunet sends back the line that begins “220”:

\[ % telnet relay1.uu.net smtp \]
Tying 192.48.96.5 ...
Connected to relay1.uu.net.
Escape character is `\[`".  
220 relay1.UU.NET Sendmail 5.61/UNINET-internet-primary ready at
Mon, 22 Feb 93 23:13:56 -0500

In the last line (which wraps over two physical lines on the page), the remote hostname appears immediately after the “220”. In order to match and extract the hostname, use the following command:

\[ expect -re "\n220 \([^ ]+\)" * \]

There are several subtle things about this command. First of all, the SMTP protocol dictates that responses are terminated by \r\n and that the initial response to the connection begins with the string 220 followed by the host or domain identification. Thus, you are guaranteed to see the string “220”.

Copyrighted material
In this chapter, I will discuss the generation and suppression of certain types of output, including normal and diagnostic output. Diagnostic output includes information helpful for debugging pattern matching problems. I will discuss debugging of script control flow and scripts as a whole in Chapter 18 (p. 403).

Pattern Debugging

In the last couple of chapters, I described how to write patterns. Clearly there are some tricky issues of which you have to be aware. Writing effective patterns is a challenging art for several reasons.

First, you have to know the rules for constructing patterns. Second, you have to understand the rules for expressing them in Tcl. Third, you have to know what characters are in the string you expect. Misunderstanding any one of these steps can cause you to write patterns that do not match.

When patterns do not match as intended, a common symptom is that the script executes very slowly. For example, the following is a fragment of a script to log in. It ought to execute quickly.

```tcl
expect "Login: "
send "don\r"
expect "Password: "
send "swordfish\r"
```

However, on a typical system this fragment takes 20 seconds to execute instead of one or two seconds. There are two problems. The first is with the patterns. The first pattern says to expect "Login: " but on a typical UNIX system the prompt is "login: ".

Copyrighted material
Instead of matching, the script waits for more input for another 10 seconds (because that is the default timeout). After 10 seconds, `expect` times out. Since there is no timeout action, the user is not informed that the pattern failed to match. The `expect` simply returns and control passes to the next command in the script.

This kind of mistake is not uncommon. Part of the reason is that on a UNIX system, the default login prompt starts with a lowercase letter while the password prompt starts with an uppercase letter. This kind of inconsistency, rampant in many interactive interfaces, is ignored by most users and naturally shows up in scripts like these.

The second problem is due to another user-interface inconsistency. There is a space character at the end of the "Password: " pattern. But the actual prompt received is "Password: ", which does not have a space at the end! This type of incorrect pattern is an easy mistake to make because most prompts include a trailing space—but not the one for the password. Repeating the earlier logic, the script waits for another 10 seconds.

Scripts with these kinds of errors may work, but with snail-like speed. One way to find out the problem is to ask Expect what it is doing internally.

The command "exp_internal 1" causes Expect to print diagnostics describing some of its internal operations. Among other things, this includes the comparisons that occur within the `expect` command. Here is the script using "exp_internal 1" along with a `telnet` command.

```bash
spawn telnet uunet.uu.net
exp_internal 1
expect "Login: 
send "don\r"
expect "Password: 
send "swordfish\r"
```

When run through the first `expect-send` sequence, the output starts out like this:

```bash
spawn telnet uunet.uu.net
expect: does "* (spawn_id 5) match glob pattern "Login: "? no
Trying
expect: does "Trying " (spawn_id 5) match glob pattern "Login: "? no
192.48.
expect: does "Trying 192.48." (spawn_id 5) match glob pattern
 "Login: "? no
96.2 ..
expect: does "Trying 192.48.96.2 .. " (spawn_id 5) match glob pattern
 "Login: "? no
```
In this chapter, I will describe how to use the `send` and `expect` commands to interact with the user and a process in the same script. For scripts that require passwords, a common approach is to interact with the user only to get the password and then to automate the remainder of the program. I will describe how to do this in a secure manner and I will also describe other topics related to passwords and security.

It is desirable to suppress character echoing while prompting for passwords. I will describe how to do this along with a broader discussion of terminal modes and how you can control them to achieve a variety of other effects.

**The `send_user` Command**

In Chapter 3 (p. 71), `send` was used to print strings to the the standard output. The first program in that chapter printed out "hello world" and was just one command:

```
send "hello world\n"
```

However, once a process has been spawned, the `send` command no longer prints to the standard output but instead sends strings to the spawned process. In the following script, the `send` command sends its argument to the spawned process, `ftp`:

```
spawn ftp ftp.uu.net
expect "Name"
send "anonymous\r"
```
The **expect** command works the same way. Initially, it reads from Expect's standard input but as soon as a process has been spawned, **expect** reads from the process.

If the process dies and a new process is spawned, **send** and **expect** refer to the new process. In Chapter 10 (p. 233), I will describe how to use **send** and **expect** to communicate with two processes simultaneously. Communicating with a process and a user is a special case of this. Because it is so common, it merits special commands in Expect.

The **send_user** command sends strings to the standard output just the way **send** does when Expect starts. Both can initially be used to send strings to the standard output. But after a process is spawned, **send** sends strings to the process while **send_user** continues to send strings to the standard output. The command **send_user** is so named because normally the standard output is immediately sent to the user. Of course, if the standard output is redirected, the **send_user** command sends strings according to that redirection.

A common use of **send_user** is to issue informative messages on the progress of an interaction with a spawned process. This is helpful when **log_user** has been invoked to suppress the normal output of a process. For example, here is the **ftp-rfc** script from Chapter 3 (p. 82). The **log_user** command has been added to suppress output from **ftp**, and a couple of **send_user** commands have been added.

```bash
#!/usr/local/bin/expect --
# retrieve an RFC (or the index) from uunet via anon ftp

if ([$length $argv] != 1) {
    send "usage: ftp-rfc [-index|#]\n"
    exit
}

set timeout -1
log_user 0

send "spawning ftp\n"
spawn ftp ftp.uu.net
expect "Name"

send_user "logging in as anonymous\n"
send "anonymous\r"
expect "Password:"
send "don@libes.com\r"
expect "ftp> "
send "cd inet/rfc\r"
expect "ftp> "
send "binary\r"
expect "ftp> "
```
In this chapter, I will cover Expect’s command-line arguments and describe more about how Expect scripts fit in with other UNIX utilities. I will also focus on the difference between running Expect interactively versus non-interactively.

Expect — Just Another Program

To the operating system, Expect is just another program. There is nothing special about it. For example, it has attributes similar to many other programs you are familiar with:

- Expect has standard input, standard output, and standard error. They can be read from and written to. They can be redirected.

- Expect can be run in the background from the command-line using & or from cron or at.

- Expect can be called from other programs, such as C programs, shell scripts, awk scripts, and even other Expect scripts.

Expect is also an interpreter, and it shares attributes of most other interpreters:

- Expect supports the #! convention.

- Expect can be run interactively, taking commands from the keyboard, or non-interactively, taking commands from scripts or standard input.

- Expect takes flags or can pass them on to scripts.

Like most interpreters, Expect takes a file name as an argument, and uses it as a source from which to read commands.
% expect script.exp

If you want to pass additional information to the script, you can do so just by putting it at the end of the command line.

% expect script.exp foo bar 17

Inside the script, this information can be found in the variable argv. The value of argv can be manipulated as a list. Here is a script called "echo.exp". It echoes each argument, prefaced with its index in argv.

#!/usr/local/bin/expect --
set argc [llength $argv]
for (set i 0) {$i<$argc} {incr i} {
    puts "$argv $i: [lindex $argv $i]"
}

When I run this script from the command line with some random arguments, it looks like this:

% expect echo.exp foo bar "17 and a half"
arg 0: foo
arg 1: bar
arg 2: 17 and a half

The last argument was kept together by double quoting it. This is a shell mechanism. Without the double quotes, the shell breaks apart arguments that are separated by whitespace. If present, the double quotes are stripped off before passing the arguments to Expect. Notice that this is exactly the same way that Tcl uses double quotes.

The script name is not included in the argument list. The arguments are only those of the script, not of Expect. This is convenient in many scripts because the argument list can be directly used without stripping out the command name.

The script name is stored in the variable argv0. Adding the command "puts "argv0: $argv0\n" as the first line of the example script causes it to print an additional line:

argv0: echo.exp
arg 0: foo
arg 1: bar
arg 2: 17 and a half

The expect from the original command line vanishes as if the script itself had been called without it. There is little reason to have the expect in there anyway. Obviously, the Expect script knows that it is an Expect script. But there is an even more important reason to get rid of it—so that a script sees the same arguments whether it is invoked as "expect script" or just "script". I will show later why this is so useful.
In this chapter, I will describe how to build scripts that communicate with multiple processes. Using multiple processes, you can build scripts that do much more than simple automation. For instance, you can connect programs together or borrow the facilities of one to enhance those of another. You can also do it transparently so that it seems like a single program to anyone running the script.

**The spawn_id Variable**

In the following script, two processes are spawned. The first is bc, an arbitrary precision arithmetic interpreter. The second is a shell. By default, send and expect communicate with the most recently spawned process. In this case, the following expect reads from the shell because it was spawned after bc.

```
    spawn bc
    spawn /bin/sh
    expect $prompt       ;# communicate with /bin/sh
```

Why is this? When a spawn command is executed, the variable spawn_id is set to an identifier that refers to the process. The spawn_id variable is examined each time send and expect are called. send and expect know how to access the process by using the value in spawn_id.

If another process is spawned, spawn_id is automatically set to an identifier referring to the new process. At this point, send and expect then communicate with the new process. In this example, “spawn bc” stored an identifier into spawn_id, but “spawn /bin/sh” replaced that with a new identifier to the shell process. The following expect command therefore communicates with the shell.
It is possible to communicate with the old process by setting `spawn_id` back to the identifier for that process. `spawn_id` is not special in this regard. It is read or written using the same commands that access other variables. For example:

```bash
spawn bc
set bc_spawn_id $spawn_id ;# save bc's spawn id

spawn /bin/sh
set shell_spawn_id $spawn_id ;# save shell's spawn id

set spawn_id $bc_spawn_id ;# talk to bc
send "scale=50;r"
```

Clearly, the value of `spawn_id` is very important. Indeed, the process whose identifier is stored in `spawn_id` is known as the currently spawned process. In the script above, `bc` is initially the currently spawned process, and then `/bin/sh` becomes the currently spawned process. When `spawn_id` is reset by an explicit `set` command, `bc` once again becomes the currently spawned process.

While not the only ones, the UNIX program `bc` and the related program `dc` are very useful to have spawned while other programs are running. Both `bc` and `dc` are capable of arbitrary precision mathematics. For example, suppose you are interacting with a process which requires you to multiply some very large number together but does not provide support itself to do it. Just change to the spawn id from `bc` and get the answer through an interaction like this:

```bash
send "1234567897293847923*234229384318401298334234874\r"
expect -re "\n([.])\r\n"
```

Here is an interaction with `dc` to change a decimal number to the oddball base of 6.

```bash
send "1928379182379871\r6op\r"
expect -re "\n([.])\n([.])\r\n"
```

Both of these leave the result in `expect_out(1,string)`.

**Example — chess Versus chess**

Very useful results can be produced by communicating with multiple processes. A simple but amusing example is the problem of having one chess process play a second chess process. In order to accomplish this, the standard output of one process must be fed to the standard input of another, and vice versa.
It is possible to communicate with the old process by setting `spawn_id` back to the identifier for that process. `spawn_id` is not special in this regard. It is read or written using the same commands that access other variables. For example:

```plaintext
spawn bc
set bc_spawn_id $spawn_id ;# save bc's spawn id

spawn /bin/sh
set shell_spawn_id $spawn_id ;# save shell's spawn id

set spawn_id $bc_spawn_id ;# talk to bc
send "scale=50\r"
```

Clearly, the value of `spawn_id` is very important. Indeed, the process whose identifier is stored in `spawn_id` is known as the currently spawned process. In the script above, `bc` is initially the currently spawned process, and then `/bin/sh` becomes the currently spawned process. When `spawn_id` is reset by an explicit `set` command, `bc` once again becomes the currently spawned process.

While not the only ones, the UNIX program `bc` and the related program `dc` are very useful to have spawned while other programs are running. Both `bc` and `dc` are capable of arbitrary precision mathematics. For example, suppose you are interacting with a process which requires you to multiply some very large number together but does not provide support itself to do it. Just change to the spawn id from `bc` and get the answer through an interaction like this:

```plaintext
send "1234567897293847923*234229384318401298334234874\r"
expect -re "\n\n"n(.*))*\r\n
Here is an interaction with `dc` to change a decimal number to the oddball base of 6.

send "1928379182379871\r6op\r"
expect -re "\n\n(.*)\r\n"

Both of these leave the result in `expect_out(1,string)`.

**Example — chess Versus chess**

Very useful results can be produced by communicating with multiple processes. A simple but amusing example is the problem of having one `chess` process play a second `chess` process. In order to accomplish this, the standard output of one process must be fed to the standard input of another, and vice versa.
In the previous chapter, I introduced the concept of a spawn id and how the spawn_id variable could be used to change the attention of Expect commands between multiple processes. In this chapter, I will demonstrate a mechanism that provides a more explicit way of denoting the current spawn id. Explicitly naming spawn ids makes it possible to handle multiple spawn ids in the same command.

I will also cover the `expect_before` and `expect_after` commands, which can greatly simplify scripts by performing common tests (such as for `eof` and `timeout`) in only a single command of the script.

**Implicit Versus Explicit Spawn Ids**

The previous chapter demonstrated various ways of interacting with two processes, an `ftp` process and a `write` process. By setting the variable `spawn_id`, the `send` and `expect` commands can communicate with either process. Here is an example of that from the previous chapter:

```plaintext
set spawn_id $ftp
send "$file1\r"; expect "220*ftp> "

set spawn_id $write
send "successfully retrieved file\r"

set spawn_id $ftp
send "$file2\r"; expect "220*ftp> "
```
It is also possible to supply `send` and `expect` with an explicit parameter representing a spawn id. In this case, the commands do not use the `spawn_id` variable. Instead the spawn id is passed as an argument following the flag `-i`. For example:

```
send -i $write "successfully retrieved file\r"
```

This command sends the string to the `write` process. The value of `spawn_id` is irrelevant, as `send` uses the value following the `-i` flag. The value of `spawn_id` remains what it was before the command.

Using this line, you can rewrite the earlier fragment as:

```
set spawn_id $ftp
send *get $file1\r*;  expect *220*ftp> *

send -i $write "successfully retrieved file\r"

send *get $file2\r*;  expect *220*ftp> *
```

The "send -i" sends the string to the `write` process while all the other commands communicate with the `ftp` process.

Using `-i` is convenient when the script only has to send one thing to another process while a lot of interaction is occurring with the currently spawned process.

The `-i` flag is also supported by the `expect` command. The `expect` commands in the previous example could have been written:

```
expect -i $ftp *220*ftp> *
```

If multiple patterns are used, they are all tested against the output from the spawn id following the `-i` flag. In the following example, `expect` executes action if either of the 220 or 550 codes are returned by the `ftp` process.

```
expect {
  -i $ftp
  *220*ftp> * action
  *550*ftp> * action
}
```

Notice how the braces enclose all of the arguments including the `-i` flag. Patterns can be specified on the same line as the `-i` flag. Do whatever you think is most readable.

Here is another rendition of the command. It looks different but does the same thing.

```
expect {
  -i $ftp *220*ftp> * action
  *550*ftp> * action
}
```
In this chapter, I will provide more detail about the `send` command, including its ability to send strings with special timings between the letters of a string. I will revisit the concepts from the previous two chapters—dealing with multiple processes—in the context of `send`. Finally, I will describe some interactions between `send` and other parts of Expect such as how to `send` without echoing.

The descriptions in this chapter will explicitly refer to the `send` command, but most of them apply to the related commands `send_user`, `send_error`, and `send_tty`.

**Implicit Versus Explicit Spawn Ids**

The previous chapter showed the differences between controlling `expect` with `spawn_id` versus using the `-i` flag. The `send` command can be controlled in the same way. For example, the two lines are equivalent—both send the string `foo` to the process corresponding to the spawn id in the `proc` variable.

```plaintext
set spawn_id $proc; send "foo"
send -i $proc "foo"
```

While the first line is longer, setting the spawn id is simpler if a single process is the focus of interaction for a group of commands. For example, if a login is performed, the implicit method (using `spawn_id`) looks like this:

```plaintext
set spawn_id $proc
expect "login:"
send "$name\r"
expect "Password:"
send "$password\r"
expect "$prompt"
```
Using explicit -i parameters requires more characters and is more difficult to read:

```bash
expect -i $proc "login:"
send -i $proc "$name\r"
expect -i $proc "Password:"
send -i $proc "$password\r"
expect -i $proc "$prompt"
```

Setting spawn_id makes it possible to easily localize which process is being interacted with. If the process has to be changed to a different one, only the set command has to change.

Procedures are excellent ways of localizing variables, thereby reducing complexity. For example, the following procedure takes a spawn id as an argument and performs a login interaction.

```bash
proc login {id} {
    set spawn_id $id

    expect "login:"
send "$name\r"
expect "Password:"
send "$password\r"
expect "$prompt"
}
```

If only a single process is being controlled, it is convenient to name the formal parameter spawn_id. Then, no explicit set is needed. It occurs implicitly at the time the procedure is called. Here is the same procedure definition rewritten using this technique:

```bash
proc login {spawn_id} {
    expect "login:"
send "$name\r"
expect "Password:"
send "$password\r"
expect "$prompt"
}
```

### Sending To Multiple Processes

Unlike the `expect` command, the `send` command has no built-in support for sending to multiple processes simultaneously. The support is not necessary, since it is possible to achieve the same effect by writing a group of `send` commands or by writing one in a
Besides starting processes, `spawn` can be used to begin interactions with files and pipelines. In this chapter, I will go into detail on the `spawn` command. I will also cover ptys—what they are, how to control them, and their features and pitfalls.

**The Search Path**

The `spawn` command follows the “usual” rules in finding programs to invoke. Both relative and absolute filenames are acceptable. If a filename is specified with no directory at all, the value of the environment variable `PATH` is treated as a list of directories and each directory is searched until the given file is found. This searching is performed by the operating system and behaves identically to the way that programs are found from shells such as the Bourne shell and the C shell.

```
spawn /bin/passwd  ;# absolute
spawn passwd      ;# relative
```

In some cases, naming programs absolutely is a good idea. In other cases, relative names make more sense. If you do use relative names, it is a good idea to set `PATH` explicitly. For example:

```
set env(PATH) */bin:/usr/bin
```

Setting the path avoids the possibility of picking up local versions of utilities that users might otherwise have on their paths. Users at other sites may need to change the path, but the single definition at the top of a script makes the path easy to change.

While resetting the path is easy enough, there are circumstances when it makes more sense to refer to programs absolutely. For example, your system may have several versions of a utility (e.g., BSD, SV, POSIX, GNU). Naming a utility absolutely may be safer than using the path.
Rather than embedding a name literally, use variables so that all the references in a script can be changed easily if necessary. For example, you might localize a particular `telnet` as:

```bash
set telnet /usr/ucb/telnet
```

Later in the script, `telnet` would then be started this way:

```bash
spawn $telnet
```

Sets of programs that live in different subdirectories under a common directory could be localized with separate variables such as in this example:

```bash
set root "/usr/kidney"
set bindir "$root/bin"
set testdir "$root/test/bin"
set demoprog "$bindir/nephron-demo"
```

Scripts that use these initializations would have `spawn` commands such as:

```bash
spawn $bindir/control
spawn $testdir/simulate
spawn $demoprog
```

If you have a number of scripts using these definitions, they can be stored in a common file that is sourced by the scripts at start-up.

Like Tcl's `exec` command and the C-shell, `spawn` also supports the tilde notation. A tilde preceeding the first component of a filename is understood to mean the home directory corresponding to the user named by that component. For example, the following command spawns the `pike` program from the files of a user named `shaney`:

```bash
spawn ~shaney/bin/pike
```

**Philosophy — Processes Are Smart**

The previous chapter demonstrated how to open files or devices and send commands to them with `puts`. This technique calls upon another program to do the handling of the device. At first glance, you might consider this inefficient. Why run two programs when one will do? Consider that reusing another program allows you to isolate all the device-specific problems in one place. If the `tip` program already knows about serial devices and how, for example, to choose among baud rates, why burden other programs with the same knowledge?

Ideally, if you just had one program that knew everything about, say, your serial devices, you would not need any others. Or perhaps, other programs could call upon your serial device program when they needed to access a serial device. This is
In This Chapter:
- Handling ^C
- Generating And Taking Actions On Signals
- The wait Command
- Taking Action When The Script Ends

If at all possible, avoid signals. They are tricky to use correctly, and signal-handling code is perhaps the most difficult to debug. Despite these warnings, there are situations in which signals are the only solution. In this chapter, I will describe the reasons why you may have to deal with signals and how to handle them. I will also present related details of the wait and exit commands.

Signals

Signals are software interrupts. They can be generated for a variety of reasons such as the pressing of certain keystrokes. In cooked mode, pressing control-C usually generates an interrupt signal in the foreground process. Processes can also generate signals in other processes—or even in themselves. This is commonly referred to as sending or raising or generating a signal. Finally, the operating system can generate signals for a number of reasons, such as if a power failure is imminent and the system is about to halt. For more in-depth information on signals, read your local man pages.

Specific signals are commonly referred to in several ways. For example, signal number 9 is usually written as SIGKILL in C programs. However, many utilities (e.g., kill) only accept 9 or KILL (without the SIG prefix). Expect accepts all three forms (9, KILL, or SIGKILL). For clarity in this book, I like to use the C-style although I will give examples of why the other two forms are occasionally useful.

The exact list of signals varies from one system to another but modern systems include those shown in the following table. There are others but the signals shown here are the ones you are most likely to deal with in an Expect script.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>hangup</td>
</tr>
<tr>
<td>SIGINT</td>
<td>interrupt</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>quit</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>kill</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>pipe write failure</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>software termination</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>stop (really “suspend”)</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>keyboard stop</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>continue</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>child termination</td>
</tr>
<tr>
<td>SIGWINCH</td>
<td>window size change</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>user-defined</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>user-defined</td>
</tr>
</tbody>
</table>

Assuming you have permission, these signals can be generated by using the kill command from a shell script or \texttt{exec kill} from an Expect script. For example, from an Expect script the following command sends an interrupt signal to process 1389.

\texttt{exec kill -INT 1389}

Expect processes can receive as well as generate signals. In the example above, if process 1389 is an Expect process, upon receiving a signal, the process looks for a command that is associated with the signal. An associated command is known as a signal \texttt{handler} or \texttt{trap}. If there is a handler, it is evaluated. When the handler has completed execution, the script (usually) returns to what it was doing before the signal arrived.

The association between a signal and its handler is created by the \texttt{trap} command. Only one handler can be associated with a signal at a time. If you make the association from within a procedure, the association remains in effect even after the procedure returns. Each association replaces the previous one for the signal of the same name.

For example, the following command causes a script to print "\texttt{bye bye}" and then to exit if an interrupt signal (SIGINT) is received.

\texttt{trap \{send_user \"bye bye\"; exit\} SIGINT}
In earlier chapters, `interact` was used in a very simple way. In reality, the `interact` command simplifies many tasks and opens up a world of new problems that can be solved. In this chapter, I will describe the more common uses for `interact`. In the next chapter, I will focus on using `interact` with multiple processes.

**The `interact` Command**

In Chapter 3 (p. 82), I introduced the `interact` command in the context of a script to automate `ftp`. The script carried out the initial part of the procedure—entering the user name and password—and then returned control to the user by calling `interact`.

The `interact` command is much more flexible than that example demonstrated. `interact` can also:

- execute actions when patterns are seen from either a user or a spawned process
- allow the user to take control of a spawned process, and return control to the script for further automated interaction, any number of times
- suppress parts or all of an interaction
- connect two or more spawned processes together, pairwise or in other combinations

Many of the things `interact` does can also be done by `expect`, but `interact` can do them more easily and efficiently. In this sense, `interact` is a higher-level command than `expect`. In other ways, `expect` and `interact` are duals. They do the same thing but have a very different way of looking at the world. As I explain `interact`, I will frequently bring up `expect` to compare the similarities and contrast the differences between the two.
In its simplest form, the interact command sets up a connection between the user and the currently spawned process. The user's terminal is put into raw mode so that the connection is transparent. It really feels like the user is typing to the process.

In its basic form, the interact command connects a user and spawned process.

If the spawned process exits, the interact command returns and the next line of the script is executed. In a simple script such as the anonymous ftp script (aftp), the interact command is the last line in the script, so the script simply exits when interact does.

**Simple Patterns**

Like the expect command, the interact command can execute actions upon detecting patterns. However, interact and expect behave very differently in many respects.

The syntax for specifying patterns and actions is similar to expect. Patterns and actions are listed as pairs of arguments. For example, the following interact command causes the date to be printed if ~d is typed by the user.

```
interact "~d" {puts [exec date]}
```

By default, a matched pattern is not sent on to the process. (Later, I will show how to change this behavior.) Thus, in this example, the process never sees the ~d.

Unlike the expect command, interact continues after matching a pattern and executing an action. interact continues shuttling characters back and forth between the user and process. It also continues matching patterns. In the example above, the ~d pattern can be matched again and again. Each time, the action will execute.

As with the expect command, additional pattern-action pairs may be listed. Also, all the arguments may be surrounded by a pair of braces, provided they do not all appear
The previous chapter had numerous examples, all showing how to create a connection between a user and the currently spawned process. The `interact` command does this by default, but it is possible to create connections in other ways. In this chapter, I will cover how to use the `interact` command with a process other than the user and currently spawned process, or with multiple processes.

**Connecting To A Process Other Than The Currently Spawned Process**

Like many of the other commands in Expect, the `interact` command accepts the `-i` flag to indicate a spawn id to be used in place of the currently spawned process. For example:

```bash
spawn telnet
set telnet $spawn_id
spawn ftp
interact -i $telnet
```

In this example, `interact` connects the user to the `telnet` process. Without the "-i $telnet", the `ftp` process is connected.

Output from the process is tested against any patterns appearing after the `-i` flag. In other words, the `-i` behaves as if a `-o` flag had also appeared.
Connecting To A Process Instead Of The User

Just as the -i flag allows substitution of one side of the connection created by interact, the -u flag allows substitution of the other side. Specifically, the -u flag identifies a process to be used instead of the user.

```
spawn procl
set procl $spawn_id
spawn proc2
interact -u $procl
```

The interact command above connects the input of proc1 to the output of proc2 and vice versa. The processes interact as shown in the following figure.

```
process selected by spawn_id
proc2  Expect  proc1
        process selected by -u flag
```

*With the -u flag, the interact command connects two spawned processes together.*

In the figure, there is no user involved. The user is still present but does not participate in the interact connection. The user keystrokes are not read nor is there any output to the user.

In Chapter 11 (p. 251), I showed how to have two chess processes communicate using expect commands in a loop. This could be rewritten using "interact -u" with the second chess process replacing the user. The original script was an expect command wrapped in a while loop, but the interact command loops internally. So using interact would avoid the extra command, making the script slightly shorter.

The chess problem is a familiar example but contrived—there is no real reason to use interact since the moves are synchronized. But in real uses, "interact -u" leads to much shorter programs.

Here is a script fragment to perform dialback, the act of having a computer dial a modem and connect to a user. This is useful in situations where the user cannot or does not want to be responsible for the connection. For example, dialback allows the phone charges to be billed to the computer rather than the user.
It is useful to run scripts in the background when they are totally automated. Then your terminal is not tied up and you can work on other things. In this chapter, I will describe some of the subtle points of background processing.

As an example, imagine a script that is *supposedly* automated but prompts for a password anyway or demands interactive attention for some other reason. This type of problem can be handled with Expect. In this chapter, I will discuss several techniques for getting information to and from background processes in a convenient manner.

I will also describe how to build a `telnet` daemon in Expect that can be run from `inetd`. While rewriting a `telnet` daemon is of little value for its own sake, with a few customizations such a daemon can be used to solve a wide variety of problems.

**Putting Expect In The Background**

You can have Expect run in the background in several ways. You can explicitly start it asynchronously by appending `&` to the command line. You can start Expect and then press `^Z` and enter `bg`. Or you can run Expect from `cron` or `at`. Some systems have a third interface called `batch`. Expect can also put itself into the background using the `fork` and `disconnect` commands.

† `cron`, `at`, and `batch` all provide different twists on the same idea; however, Expect works equally well with each so I am going to say "cron" whenever I mean any of `cron`, `at`, or `batch`. You can read about these in your own man pages.
The definition of *background* is not precisely defined. However, a background process usually means one that cannot read from the terminal. The word *terminal* is a historic term referring to the keyboard on which you are typing and the screen showing your output.

Expect normally reads from the terminal by using `expect_user`, "gets stdin", etc. Writing to the terminal is analogous.

If Expect has been started asynchronously (with an & appended) or suspended and continued in the background (via bg) from a job control shell, `expect_user` will not be able to read anything. Instead, what the user types will be read by the shell. Only one process can read from the terminal at a time. Inversely, the terminal is said to be controlling one process. The terminal is known as the *controlling terminal* for all processes that have been started from it.

If Expect is brought into the foreground (by fg), `expect_user` will then be able to read from the terminal.

It is also possible to run Expect in the background but without a controlling terminal. For example, cron does this. Without redirection, `expect_user` cannot be used at all when Expect lacks a controlling terminal.

Depending upon how you have put Expect into the background, a controlling terminal may or may not be present. The simplest way to test if a controlling terminal is present is to use the `stty` command. If `stty` succeeds, a controlling terminal exists.

```plaintext
if [catch stty] {
    # controlling terminal does not exist
} else {
    # controlling terminal exists
}
```

If a controlling terminal existed at one time, the global variable `tty_spawn_id` refers to it. How can a controlling terminal exist at one time and then no longer exist? Imagine starting Expect in the background (using "&"). At this point, Expect has a controlling terminal. If you log out, the controlling terminal is lost.

For the remainder of this chapter, the term *background* will also imply that the process lacks a controlling terminal.

**Running Expect Without A Controlling Terminal**

When Expect has no controlling terminal, you must avoid using `tty_spawn_id`. And if the standard input, standard output, and standard error have not been redirected,
A description of debugging techniques could easily fill an entire book or more—and rightfully so. In any software development, debugging often takes more time than the programming (not to mention design\(^\dagger\)).

The need for debugging is even more exaggerated in Tcl where many people code while thinking. The interpretive nature of the language makes this seductively easy.

Good design principles can help avoid getting stuck in a quagmire. Careful thought can prevent many common traps. In the same way, careful (and imaginative) thought is helpful in solving problems. Some people claim never to use debuggers. I am not one of them, but sometimes it helps to walk away from the keyboard and simply think through what could possibly be happening—or what cannot possibly be happening but is anyway.

**Tracing**

In this section, I will briefly recap some useful techniques and mention a few more that are very useful for tracking down problems. All of these have to do with tracing control. I will go into other debugging techniques later.

Simple output commands (using `puts` and `send`) can go a long way towards finding problems. However, putting commands in and taking them out is a hassle. You can conditionalize their execution with a variable.

```tcl
if (verbose) {
    puts "..."
}
```

\(^\dagger\) Assuming you did any.
Rather than having raw `if/puts` commands throughout your code, it is cleaner to isolate them in one place, such as a procedure called `vprint` (for "verbose print").

```c
proc vprint {msg} {
    global verbose
    if {$verbose} {puts "$msg"}
}
```

Later if you decide to redirect output to a file or window, you only have to change the one output command in `vprint`.

This idea can be augmented in many different ways. For example, the following definition prints the procedure name before the message.

```c
proc vprint {msg} {
    global verbose
    if {$verbose} {
        puts "[$index [info level -1] 0]: $msg"
    }
}
```

## Logging

It is often useful to write information to files so that you can study it later. You can write log files yourself or you can use Expect's logging functions.

The commands `log_user`, `log_file`, and `exp_internal` can all be helpful while debugging. These commands can also be controlled indirectly through procedures similar to the `puts` example above.

I will summarize what these commands do. The `log_user` command controls whether the output of spawned processes is seen. In most scripts, you want to leave this set one way or the other, but it is nice to have the flexibility to turn it off and on during development. The `log_user` command is described further in Chapter 7 (p. 175).

The `log_file` command is related to the `log_user` command. However, `log_file` has almost no uses other than for debugging. The `log_file` command records everything from a spawned process. Even output suppressed via "log_user 0" can be recorded. The `log_file` command is further described in Chapter 7 (p. 180).

The `exp_internal` command is another command that is useful only for debugging. The `exp_internal` command enables the printing of internal information, mostly concerning pattern matching. This command was discussed in Chapter 7 (p. 165).
Index Of Scripts

This is an index to all scripts or substantial fragments of code.

Symbols
#! rewriting 216

A
aftp 83, 144
anonymous ftp 83, 144
answerback 250, 340
area of circle 36

B
bc 233

C
chdir 469
chess 235, 251, 492
choose 390
command loop 231, 346

D
date 177
desk calculator 233
dialback 3, 350
disconnect 375
dislocate 354, 384
Dvorak 325

E
echo arguments 214
emulating ftp 467
encryptdir 465
encrypting a directory 465
escape 231, 390
exit handler 321
exp_popen 495
expect_four_byte_int 156
Expectk example
terminal emulator 449
tknewsbiff 475
tkpasswd 434

F
factorial 29, 31
fastest machine 253
Fibonacci numbers 34
Fibonacci numbers, recursively 35
file transfer 351
firstline 136, 178
fixline1 216
fsck 8, 339
ftp
tover telnet 467
put 133
ftpcmd 242, 243
ftp-tfc 186
G
gdb, testing 6
getpass 199, 206

H
hunt, the game of 148

I
idle fakeout 343
inactivity handler 330
interact, complex 361

J
joint control 255

K
keyboard, Dvorak 325
kibitz 255, 355
under cron 380

L
login 272
lognumber 338

M
mail 383
mapwindow 478
maxtime 98, 100
mkfifo 385
mkpasswd 525

N
news, wakeup on unread 475
noidle 343

O
OCLC 398
Online Computer Library Center 398

P
passmass 154
passwd 5
passwd GUI 434
password
generator 525
prompt for 377
querying on behalf of a program 201
password, query for 199
password_backspace 445
ping 97, 100, 101, 178
preventing bad commands 346
print array elements and values 53
prompt
default Expect 228
styles 376
tset 147
putentry 133

Q
quadratic formula 54
quadratic formula (call by reference) 58

R
racing with rlogin 253
random numbers 524, 525
random passwords 525
reading lines in a file 62
reading lines in a file (fast) 62
reading nulls 156
reconnecting 378
record a session 180
record phone numbers 338
rlogin
is a test of speed 254
rsh 118, 119
with same directory 123
rlogin with same directory 122
rn (read news) 195
robohunt 148
rogue 141, 458
rsh via rlogin 118, 119
rz 351

S
script 180, 337
script, a better version 334
send slowly 277
send_slow 277
set password on multiple machines simultaneously 154
SIGCHLD handler 314
SIGTSTP handler 316
SIGWINCH handler 316, 367
slow send 276
slow telnet 152
spawn_ftp 244
split host and domain 137
static variables, emulating 59
stop script after some time 98
su2 179, 200
sum a list 39
sz 351

t

telnet 494
  slow 152
  with same directory 341
ten_chars_per_sec 276
term 460
term_expect 460
terminal emulator 449
timeallcmds 256
timed reads 77
timed_read 77, 112
timed-read 77
timing all commands 256
tip 196
tkpasswd 434
tkterm 449
traceproc 406
trap all signals 318
tset 147
two users connected to an application 255

u
unbuffer 299
unknown 68
unmapwindow 477

v
verbose 469
vi 104
Index

Italicized page numbers denote the most definitive references. Other references denote discussion or examples.

&. 66, 372
   disadvantages to disconnect, 377
&&, 27
(\[\text{r}]\).\n, 145
*, 6, 27, 32, 63, 87, 107, 153
   at end of pattern is tricky, 89
   at start of pattern is redundant, 88
+, 27
   as procedure name, 363, 365
->, 228
., 108
   as widget separator or top-level
   widget, 430
*: 107, 116, 331
*\n, 129
+. 251
   implicit, 357
.crypt suffix, 466
.cshrc, 11
.dislocate, 386, 391
.expect.rc, 222, 232
.netrc, 11, 22
.newssrc, 481
.o files, 517
.profile, 11
.rc, 22, 221, 232, 516, 521
.rhosts, 13
.tknewsbiff, 476
/, 27, 63, 430
/\X, 329
/bin/cat, 526
032 (A2), 468
-1, 81, 502
1990 Winter USENIX Conference, xxvi
1991 Winter USENIX Conference, xxvii
2>, 293
3Com, 14
7 versus 8-bit connection, 352, 467

A
absolute cursor motion, 456
Academic Press, 20
Accardo, Thomas, xxix
actions, 75
  background, 447
active file, 482
adb, 9
adding extensions
  to Expect, 516
  to Expectk, 517
Addison-Wesley, 19, 394
Advanced UNIX Programming, 305
after, 459, 461-462
aftp, 83, 143, 163, 232
aftp.exp, 293
AIX 3.2, 291
ALARM, 509
Amdahl, 14
anchor, 73
differences between glob and regular
  expression, 126
and, 27
anonymous ftp, 83
ANSI terminal, 462
answerback, 250, 340
any_spawn_id, 259, 261
API, 70
append, 46, 49, 56
arbitrary precision arithmetic, 233, 246
Archie, 18, 22
argc, 215, 515
arguments
do not interpret remaining, 217-218
  varying, 39
argv, 77, 215, 515
argv0, 214-215
arithmetic, arbitrary precision, 246
array, 49, 362
  information, 52
  names, 52, 367
  size, 52
arrow keys, simulating, 286
asleep, falling, 211
assert, 183
associativity, 28
at, 371
AT&T, 15
atom, 107, 110
auto_path, 526
Autoconf, xxviii
automatic Expect script writer, 270
automation
  of non-UNIX systems, 14
  partial, 8, 82
  total, 4
avoiding echo, 335

B
background, 372
actions, 447
  expect library, 507
patterns, 446
  processing, 371
backquotes, 78
backslasses, 24, 109, 125
  in patterns, 91
backup, 14
  from cron, 13
Baha'i World Centre, 14
base, change, 234
batch, 371
Baughman, Sue, xxix
Bayona, Miguel Angel, xxix
bc, 233-234, 246
bedroom, spawning, 78
begin standout mode, 450
behavior differences of Expect versus
  Expect as an extension, 513, 515
benchmarking, 529
Betz, David, 429
bg, 240, 371-372
bidirectional
communication over pipes, 305
process pipelines, 292
biff, for news, 475
Bimmler, 331
binaries Expect, 492
bind, 431, 437, 442, 444–445, 448, 457, 487
bindings, 432
for Expect, 492
bold, xxxiii
books
Advanced UNIX Programming, 305
Computer Lib, 121
Internet System Handbook, 394
Numerical Recipes in C, 524
Practical Programming in Tcl and
Tk, 20
Software Solutions in C, 20
Tcl and the Tk Toolkit, 19
TCP/IP Illustrated, Volume 1, 394
X Window System Series, 429
boolean operators, 27
 Bourne shell, xxvi, 70
braces, 29, 32
in lists, 40
nesting, 33
open, 33
Bradfute, Todd, xxix
branch, 109–110, 115
break, 34, 141, 228–229, 461, 474
breakpoint
action, 422
limitations, 424
by expression, 420
by line number and filename, 419
by pattern match, 420
defining, 419
deleting, 426
general form, 425
listing, 425
bridges, 14–15
Brighton, Allan, 59
broken pipe signal, 315
Brown, Thomas, xxix
browsing global variables, 409
BSD, 449
buffering
input, 74
interact, 332
output perpetually, 154
bug, OS, 291
button, 430, 438
clicks, 431
C
C, xxv, 70, 491, 513, 517, 519
subroutine, 524
–c, 217–218, 232
c, 418, 423
C language identifier, 108
C shell, xxvi, 70
C++, 491, 513, 517, 519
special instructions for, 514
caching regular expressions, 503
Cambridge University Press, 524
canonical input buffer size, 286
capability, 450
caret, 73
carriage-return linefeed, 80
cat, 289, 300, 467
command to remote system, 471
–u, 292, 488, 526
cat_buffers, 526
catch, 53–54, 58, 60, 67, 296, 318, 334, 367,
372, 374, 438, 442, 469, 473, 480,
482, 489, 527
catching errors, 53
causing errors, 55
cp, 493
cd, 470
CD-ROM, 17
ceil, 27
CenterLine Software, 14
cert.scie.cmu.edu, 400
change the base of a number, 234
character graphics, 142
output, 283
character translations, 325
check dictionary, 438
checkbutton, 438
chess, 12, 234, 262
timed, 246
with expect –i, 251
with interact, 350
chess.exp, 17
child, 391
termination signal, 308, 313
CHILDKILLED, 313
chmod, 72, 215
choose, 390
chunking of output, 89–90, 152
Cisco, 14
ci, 450
clear screen, 450
close, 101, 249, 472, 474, 482, 499
file, 60
implied at end of script, 102
–slave, 295, 364
cm, 450
COLUMNS, 450
command
name clashes, 514–515
timestamps, 256
tracing, 405
commands
debugger, 412
evaluating lists as, 56
comments, 24, 70
common mistakes, 258, 495, 518
comp.lang.tcl, 21, 463
comp.source.unix, 400
compilation of regular expression, 503
compiled
Expect, 518
Expect programs, 492
compress, 467
CompuServe, 275
Computer Lib, 121
Computer Security FAQ, 203
Computerized Processes Unlimited, 21
ComputerVision, 15
concat, 42–43
conference proceedings, 21
connecting to background processes, 381
console output, 300
contacting multiple users, 382
continuation lines, 33
continue, 34, 143, 228, 339, 482
execution, 418
signal, 308, 316
control structures, 29
controlling terminal, 245, 372
controlling unreliable processes, 15
converting from floating-point to integer, 27
cooked mode, 89
co-processes, xxiv
copying file through firewalls, 15
copyright information, 15
core, 312
cos, 27
Courier, xxxii–xxxiii
cron, 12, 371, 373, 380, 401
backup, 13
crypt, 2, 200
directory, 465
cryptdir, 465
csh, 226
C-style real number pattern, 140
cu, 289–290
current scope, 417
currently spawned process, 234
Curses, 10
output, 283
testing, 458
cursor motion, 456
cursor positioning character sequences, 142
Cygnus Support, 15, 21

D
d, 417
daemon, 392
telnet, 392
Data General, 14–15
databases, 14
date, 314
versus timestamp, 530
dbg, 426
dbg array, 424
Dbm, 69
dbx, 410, 412
dc, 234
DCL, 330
deadlock, not a problem, 305
debtron, 355
debug, 411
   1, 410
   --now, 411
debugger, 223, 410
   commands, 412
   flag abbreviations, 420
   help, 426
   output compression, 428
   output width, 426
   prompt, 410
   repeat last action, 412
   testing, 6
debugging, 403, 509
   disconnected processes, 377
   logfile, 180
   patterns, 165
decryptdir, 465
default, 101
deferring evaluation, 29
defining breakpoints, 419
definitions, xxxii
   DejaGnu, 15
deleting
   breakpoints, 426
   files with funny names, 211
Delmas, Sven, 432
demos, 14
destroy, 438
device drivers as programs, 289
diagnostics, 171
   output, 165, 167
   redirecting from shell, 173
dial
   pager, 85
   phone numbers from a list, 85
dialback, 3, 369, 401
   via interact, 350
dictionary
   check, 438
   time to load, 522
diff, 67
Digital Equipment's Western Research Laboratories, 19
direct spawn id, 268
directory encryption, 465
disconnect, 371, 375, 379, 388, 401, 519
   advantages over &, 377
   disconnected processes
   debugging, 377
   reconnecting, 378
   strategy for getting passwords, 373
dislocate, 17, 230, 354, 390
   example interaction, 388
   manager for disconnected processes, 384
DISPLAY, 123
   -display, 433, 476
displaying widgets, 431
distance, 59
DO, 393
do, 450
Dodd, David, xxix
dog, 328
domainname, 84
DONT, 393
dos, 328
DOS TSR, 348
dot rc file, 221
DOTDIR, 222, 478
double buffering, 152
down one line, 450
DTR, 196
duck: quack, 117
dumb terminal emulator, 447
dump, 9
   estimating number of tapes, 126
   reprompting, 194
Dvorak, 17, 330, 347
   keyboard, 325
dynamically generating procedures, 232

\$E

ECHO, 393
echo, 119, 199, 273
   avoidance, 335
   interact, 333, 335
   passwords with asterisks, 444, 463
echo.exp, 215, 219
edit-compile-debug cycle, Expect is
easier, 491
efficiency, 359, 440
element, 506
doubbling up, 326
e Else, 31, 478, 483
euuchs, 328, 333
else, 31
europe, 157
Elvis, 96
eval, 56, 59, 341, 368
emacs, xxiv–xxv, 344
expect, 160
shell, 226, 301
spawn, 99
email address pattern, 137
evaluating
emulator, 448
commands in other scopes, 59
dumb terminal, 447
lists as commands, 56
Encore Computer, 15
evaluation stack, 405, 410, 414
encrypting a directory, 465
event loop, 432
encryption function, 5
expect, 445
eof file, 98
event manager, 508
end of file, 98
event manager, 508
end standout mode, 450
endp, 505–506
exact, 327
ENOMEM, 502
pattern, 332, 504
entry, 437, 442
exec, 9, 66, 81, 196, 466
env, 67, 120
2>, 293
env(PATH), 287
cp, 473
date, 338
deltrim files with funny names, 211
compress, 474
deltrim files with funny names, 211
kill, 296, 308, 345, 367
mv, 67
~STOP, 346
play, 477
~STOP 0, 317, 468
rm, 67
tilde substitution, 67
rm, 67, 466, 473–474
umdecode, 473
unencode, 474
versus system, 207
xterm, 295
termcap, 494
xterm, 295
–e, 488
execclp, 494
execute command before script, 217–218, 232
execute UNIX commands, 66
execvp, 494
exercises, a note about, xxxiii
exit, 230, 374, 390, 431, 478, 484
exit, 390
handling, 321, 515
implied at end of script, 100
–noexit, 516
–onexit, 321, 367, 516
reset terminal, 204

status, 97
rsh, 117
value, 178
exp_prefix, 311, 433, 501, 515
EXP_ABORT, 509
exp_app_exit, 520
exp_autoallocpty, 498-499
exp_buffer, 505-506
exp_buffer_end, 505
exp_case, 506
exp_child_exec_prelude, 498
exp_close_in_child, 499
exp_close_tcl_files, 499
exp_cmdfile, 520
exp_cmdfilenname, 520
exp_cmdlinecmds, 520
exp_compiled, 501, 503, 505
exp_console, 497
exp_continue, 145, 147-148, 249, 268, 357, 472, 483
affects spawn_id, 238
--continue_timer, 146
exp_cook, 520-521
exp_debug, 311
exp_debugfile, 509-510
exp_disconnect, 508
expDisconnected, 508, 519
exp_end, 492, 501-503
EXP_EOF, 502
exp_error, 521
exp_exact, 501, 504
exp_exec_library, 526
exp_exit, 311, 521
exp_exit_handlers, 521
exp_expect (no such command), 515
exp_expectl, 501-509
exp_expectv, 506-507
exp_fexpectl, 492
exp_full_buffer, 502
EXP_FULLBUFFER, 502
exp_getpid, 520
exp_glob, 492, 501-503
Exp_Init, 513-514, 516
exp_interactive, 520
exp_internal, 166, 171, 174, 404, 519
-ex, 174
exp_interp, 520
exp_interpret_cmdfile, 521
exp_interpret_cmdfilenname, 518, 521
exp_interpret_rcfiles, 521
exp_interpreter, 521
exp_is_debugging, 509-510, 519
exp_library, 526
exp_logfile, 509
exp_logfile_all, 509
exp_loguser, 509-510, 519
exp_main_exp.c, 517-518
exp_main_tk.c, 517
exp_match, 505
exp_match_end, 505
exp_match_max, 506
exp_null, 501, 504
exp_open, 304
--leaveopen, 305
exp_parse_argv, 521
exp_pclose, 500
exp_pid, 304, 496, 500
exp_popen, 492, 494-497, 499-500
exp_pty, 498
exp_pty_slave_name, 497
exp_readenv, 509
exp_regexp, 501-503, 507
exp_remove_nulls, 504
EXP_RESTART, 509
exp_send, 409, 433, 483, 515
-ex, 489
exp_slave_control, 510
exp_spawnl, 494-496, 500
exp_spawnv, 494-495, 500
exp_stty_init, 497, 499
exp_tcl_debugger_available, 520
EXP_TIMEOUT, 502, 508
exp_timeout, 502, 509
exp_ttcopy, 497
exp_ttyinit, 497
exp_type, 506
enum, 501
exp_version, 527
Expect
adding extensions to, 513, 516
as an extension, 513
binaries, 492
bindings, 492
buffering, 74
compiled, 518
experimenting with, 225
finding, 18
in the background, 371
job control, 274
library, 429, 492–493, 510, 514, 517, 526
    in background, 507
    Tcl-less, 519
Makefile, 514
philosophy, xxv
prompt, 71
README, 16
retrieve latest version, 106
running interactively, 221
script, 72
scriptless, 518
send, 433
software distributions, 17
version, 527, 530
with no controlling terminal, 372
expect, 71–72, 345, 357, 466, 471, 482–483
\r\n, 119
argument ambiguity, 159
    –brace, 160
    braces, 76
common mistakes, 166, 177, 258
control flow in actions, 140
double buffering, 152
    –ex, 134
from files, 500
from multiple processes
    simultaneously, 249
from streams, 507
function side-effects, 504
function side-effects picture, 505
    –gl, 109–110, 135, 160
    –i, 248, 488
    associates with which
    pattern?, 252
    list of spawn ids, 255
    versus spawn_id, 248
indentation, 75
    –indices, 124
    internal comments, 158
    internal timer, 146
    library functions, 504
    multiple timeouts, 253
    –nocase, 139, 211
    no-op action, 190
    –notransfer, 154
    null, 155
    output, recording all, 180
    pattern-action pairs, 75
    –re, 109, 135, 472, 475
    regular expression limits, 158
    repeating, 145
    speed, 148
    Tk event loop, 445
expect program, 217–218, 232
    –, 218, 224
    –, 217–218
    #! guidelines, 221
    arguments, 214
    –b, 217, 224
    –c, 515, 520
    multiple occurrences, 219
    –D, 311, 410
    0, 411
    enable debugger, 217, 223
    –d, 515
        print internal diagnostics, 217, 223
    –f, 217, 220
    –i, 520
        run interactively, 217, 221
invoking without saying
    “expect”, 215
    –n, 218, 221
prompt, 232
putting into background, 371
expect.h, 493, 496–497
expect.rc, 521
expect_after, 101, 247, 259–261
    actions, 267
    –info, 266
expect_background, 446, 448, 456
    –info, 447
expect_before, 101, 247, 259–261
    actions, 267
-all, 267
common problems, 264
complex example, 262
-info, 266
-noindirect, 269
EXPECT_DEBUG_INIT, 312, 411
expect_four_byte_int, 285
expect_out, 7, 297
  tracing, 406
  writing, 243
expect_out(0,string), 73
expect_out(1,string), 111, 236, 473
expect_out(2,string), 111
expect_out(9,string), 111
expect_out(buffer), 73, 96, 111, 124, 408
  tracing, 407
expect_out(spawn_id), 261, 359
expect_out(X,end), 124
expect_out(X,start), 124
expect_out(X,string), 124
EXPECT_PROMPT, 120, 148, 341, 467
expect_tcl.h, 514, 519
expect_tty, 210
/dev/tty, 210
expect_user, 192, 197, 199, 238, 291, 340,
  345, 372-373, 468, 470
expectd.proto, 398
ExpectTerm, 10
Expectk, 10, 432-433
  -flags, 433
expectl, 500
expectv, 500
experimenting, 225
exposure, widget, 431
expr, 28, 443, 469, 487, 524
expressions, 27, 37
  unquoted strings in, 37
Extended Tcl, 69
extensions, 69, 513
  others, 522

F
factorial, 29
falling asleep, 211
FAQ, 20
  Computer Security, 203
script to retrieve, 401
fastest system, find, 253
Faught, Danny, xxix
fclose, 499
FD_SET, 508
fopen, 495
feedback without echo, 335
fexact, 500
fexactv, 500
fg, 240, 372
fgetc, 496, 507
Fibonacci, 34
fifo, 299, 378
  creation, 388
  nonportable, 385
eof, 292
fifo_pair_create, 387
fifo_pair_remove, 387, 391
ffos, 246, 292, 356, 378, 498

figure
  expect function side-effects, 505
  interact, 324
    -input/output, 354
    -u, 350
kibitz, 356
tknewsbiff, 475
tkpasswd GUI, 435
xkibitz, 365
file, 217
  access, 60
  atime, 65
  close, 60
  delete, 67
  dirname, 64
  executable, 65, 298
  exists, 65, 480
  extension, 64, 466
  flush, 61
  gets, 62
  information, 65
  isdirectory, 65
  isfile, 65
  open, 60
  owned, 65
  read, 62
  readable, 65, 479, 481
remove, 67
rename, 67, 70
rootname, 64
seek (to location), 62
size, 65
stat, 66
tail, 64, 479
tell (location), 62
transfer over telnet, 467
type, 65
unique name, 70
writable, 65
write, 61
FILE *, 492
file identifier, 60
converting to spawn id, 304
filename
manipulation, 64
matching, 63
reserved, 224
files as spawned processes, 289
filtering bad commands, 346
find, 136
fastest system, 253
finger, 392
finish, 418
firstline, 305
FirstSearch, 398
fixline1, 216-217
flag, 218
floor, 27
flush, 61, 304
focus, 437, 442
for, 30, 461, 481
foreach, 39, 238, 367, 466, 484-485, 487
fork, 297, 371, 374, 388, 401, 508
non-intuitive side-effects, 374
formal parameters, 34
format, 46, 487, 525
four byte integer pattern, 156
fprintf, 492, 496, 504
fputc, 496
frame, 436-437
fread, 496
freak biological accident many years ago, xxix
free, 504
Free Software Report, 15
Frequently Asked Questions List, 20
Frisbee, 381
fscanf, 496
fsck, 1, 8, 339
ftp, 1, 11-12, 22, 78, 80, 103, 185, 189, 336, 344, 349, 392, 467
anonymous, 83
by mail, 19
directory retrieval, 131-132
library, 511
mirror, 270
prompt, 190
recursive, 17, 133, 163, 336
rewriting, 22
with retry, 143
ftpcmd, 242
ftp-rfc, 17, 187-188
full_buffer, 151
fun stuff, 2
fwrite, 496

G
game
chess versus chess, 234
robohunt, 148
rogue, 142, 458
Garamond, xxxii
GCT, 17
gdb, 2, 6, 410, 412
GENie, 275
get_old_password, 442
get_users, 438
gethostbyname, 131
getpass, 199, 211
better version, 206
gets, 62, 304, 474, 482
stdin, 345, 372
gibberish, due to parity, 157
GL, 69
~gl, 109-110
glob, 63, 67, 478
patterns, 87, 91
command, 36
scope, 413
variables, 36
glomerular nephritis, 325
GNU C and C++ testing, 15
GNU, 211
gone fishing!, 2
Good party lash night!, 279
Gopher, 392, 397
grab, 442
grep, 440
avoiding, 438
GUI
builder, 432
passwd, 17
guidelines, 221
gzip, 489

H
handler
exit, 321
signal, 308
handling errors, 53
hangup, 102
signal, 308, 314
help, debugger, 426
hexadecimal, 27
pattern, 106, 108
highlighting, 451
history, 226
event number, 228
nextid, 228
redo, 227
Holst, Wade, 21
Holzmann, Gerard, xxix
HOME, 222
home directory, 182
Hopkirk, Mike, 20
host-independent audio program, 480
hostname, 84
hotkeys, 348, 463
Houdini, 96
HP, 14, 510
HTML browsers and converters, 20
Huebner, Rob, xxix
hugs and kisses, 331
hung pty, 302, 305
hunt-and-peck, 280
HUP, 400

I
–I is supported by most commands, 249
I like cheesecake, 138
IAC, 393
IBM, 14–15
idle workstations, good use for, 400
if, 31, 443, 469, 471, 473–474, 478, 481, 483, 485, 489
ile, 226
illegal instruction, 313
in.fingerd, 392
in.ftpd, 392
inactivity monitor, 347
in-band error-reporting protocol, 358
include files, 493
incr, 30, 366, 461
indigestion, system, 153
indirect spawn id, 268, 359
inetd, 204, 371, 392, 395, 400
inifio, 378
infinite timeout, 344
infinity, 278
info, 69
commands, 52
exists, 51, 218, 366, 413, 481–482, 485
globals, 52
level, 52, 60
procs, 52
script, 52
info@radio.com, 401
information on current patterns, 266
Init functions, order of, 516
input limit in cooked mode, 274
insecurity, 202
insert mode, 452
integer pattern, 108
inter_return, 229–230, 340, 344
interact, 8–9, 82, 123, 249, 259, 317, 323, 337, 357, 359, 368, 391, 459, 468

Copyrighted material
argument ambiguity, 160
-brace, 325
break, 339
buffering, 332
caracter translations, 325
caracters that do not match, 331
continue, 339
default action, 341
-echo, 334-335
echo, 333, 335
efficiency, 359
eof, 342, 353
-ex, 327
figure, 324, 354
-I, 349, 351, 353, 360
-indices, 329
-input, 353-355, 358, 360-361, 368
inter_return, 340
job control emulation, 360
matching is different than
-expect, 329
multiple processes, 378
-nobuffer, 337-338, 340
null, 343
-o, 328, 340, 349, 351
optimized for speed, 408
-output, 353-355, 358, 360-361, 368
-re, 329
redirecting to /dev/null, 359
-reset, 317, 344, 346, 355
to prevent character loss, 346
return, 338-340
returning in the caller, 340
simple patterns, 324
terminal mode, 226
timeout, 343, 353
-u, 295-296, 351-354, 360, 378
figure, 350
interact_out, 361
interact_out(0,end), 329
interact_out(0,start), 329
interact_out(0,string), 329
interact_out(buffer) does not exist, 329
interact_out(spawn_id), 359
interacting with background
processes, 381
interactive disconnected processes, 223
Interactive Line Editor, 226
Internet System Handbook, 394
Internet Talk Radio, 401
Internet Town Hall, 401
 interoperability, 394
Interpret As Command, 393
interpreter, 225, 229-230, 341, 363, 368,
  390
     inter_return, 229
     multiple, 321
     prompt, 226-227, 516
interrupt signal, 308
ioctl, 510
italics, xxxii

J
jim, 357
job control, 7, 12, 240
     interact, 360
     shell, 310
shell-style can't be automated, 240
Jobs, Steve, 176
Johnston, Barry, xxix
join, 45-46, 64, 526
joining lists into string, 45
journals, 21

K
k1, 457
KABORT, 358
KDATA, 358
kdestroy, 376
Kenna, xxix, 435
Kerberos, 5, 13, 376
kermit, 196, 289
kernel-generated stop signal, 316
Kernighan, Brian, xxix
keyboard stop signal, 308
keystrokes, 431
kf1, 457
kibitz, 17, 256, 355, 357-359, 382
dynamic, 361
figure, 356
implementation, 356
M

macho stud language, 3
Mackerras, Paul, 216
mail
  commercial provider, 275
  process, 383
  protocol, 130
  sending from Expect, 383
  verify address, 175
make_selection, 443
Makefile, 514, 517
man pages, xxiii, 19
manager
  disconnection, 379
  for disconnected processes, 384
Mariano, Adrian, xxix
Martin Marietta, 14
master
  and slave, 294
  pty, 294
match anything but the given
  characters, 129
match_max, 150, 249, 408
  -d, 150
maxtime, 98, 100, 163, 232
Mayer, Neils, 429
McLennan, Michael, 60
message-of-the-day, 122
meta key, 457
mirror, ftp, 270
miscroute, 121
missing return, 345
mkdir, 66
mkfifo, 385
mkpasswd, 17, 524
modem
  AT command in a loop, 211
  dial, 85
  monitoring usage, 338
  testing and setting, 14
  works by hand but not by script, 196
modulo, 27
monitor
  inactivity, 347
  modem usage, 338
Moore, Jeff, xxix
more, 149
Morse, Will, xxix
Mosaic, 392, 397
  slick but stupid, 397
Motif, 10, 429
Motorola, 14
moving up or down the stack, 417
Mulroney, Lenore, xxix
Mulroney, Sue, xxviii
multiple commands on a single line, 24
multiple processes, 233
  interacting with one another, 234
multi-user control, 17
mv, 67
N
N, 415, 423
n, 413, 415, 423
named pipes, 246
National Cancer Institute, 14
National Institute of Standards and
  Technology, xxviii, 16
NAWS, 401
Negotiate About Window Size, 401
Nelson, Ted, 121
netpipe, 401
netstat, 401
network
  devices, 14
  pipe, 401
  slow, 152
new features, telling the user about, 336
newgrp, 211
newline, 79
news, unread, 475
newsgroup comp.lang.tcl, 21
NIS, 5, 434
NIST, xxviii, 16
NNTP, 482
nondestructive space, 450, 456
no-op, 190
not, 27
NSUBEXP, 505
NULL, 504
null character, 24, 155, 343, 396, 504
O
O'Reilly & Associates, 429
O'Reilly, Tim, xxix
Obfuscated C and Other Mysteries, 401
Obfuscated C Code Contest, xxvi
OCLC, 398
octal, 27
octal integer pattern, 109
older workstations, good use for, 400
one line pattern, 135
Online Computer Library Center, 398
open, 60, 63, 284, 289, 292, 472, 474, 482, 523
pipeline, 526
w+, 284, 290, 292
with interactive programs, 284
or, 27
Oracle, 69
order of Init functions, 516
OSF, 429
Ousterhout, John, xxvi, xxviii, 19–20, 28, 69
outfifo, 378
output
disabling, 174
merged with standard error, 175
output chunking, 89–90
overview of Tk, 430

P
p, 428
pack, 431, 435–438, 442, 479
pager, 85
Paisley, Scott, xxv, xxvii
parallel set, 70
parameters, xxxii, 34
parentheses
for pattern feedback, 111, 115–116
limit of nine, 116
override expression precedence, 28
override pattern precedence, 110
parity, 157, 249, 291
pair, 68, 412, 428
partial automation, 82
pass by reference, 36, 57
pass by value, 36
passmass, 17, 155
passwd, 4–5, 7, 200, 436
executing on multiple hosts simultaneously, 155
GUI, 434
rewriting, 5
password, 525
echo with asterisks, 444, 463
entering twice, 377, 441, 466
generating random, 524
prompt, 375
reuse, 376
stored in script, 375
password_backspace, 445
PATH, 215, 287
pattern
any character but ], 129
any character but ^, 129
any character but a letter, 129
background, 446
backslashes in, 91
computer generated, 135
C-style language identifier, 108
C-style real number, 140
debugging, 165
demail address, 137
exact, 327, 332
four-byte integer, 156
ftp listings, 132
ftp prompt, 190
full_buffer, 151
hexadecimal, 106, 108
integer, 108
longest string not including a blank, 129
ls output, 132
message-of-the-day, 122
octal integer, 109
one and only one line, 135–136
politics of, 154
prompt, 119, 121, 125, 147, 167
range, 91
readable, 90
real number, 110
really slow, 165
Roman numbers, 106
shell-style, 32, 91
simple, 134
SQL, 135
switch, 133
Tcl integer, 109
timeout (literal), 106, 110
type prefix, 109
pattern match
  limits, 149
  strategy, 113-114, 130
pattern-action pairs, 75
pclose, 500
Pendleton, Bob, 226
Perl, xxiv-xxv, 70
perpetual buffering, 154
Phelps, Tom, 19
PHIGS, 69
physiologist, 325
pi, 36
pid, 304, 471, 473, 524
pid2whatever, 362
pidfile_read, 386
pidfile_write, 386
piece, 110
Pike, Rob, 288
ping, 96, 178
pipe, 498-499
  bidirectional communication, 305
    write failure signal, 308
pipeline, 61, 284
  compression through, 489
play audio, 480
politics of patterns, 154
poll, 292, 303, 508
popen, 495, 500
POSIX, 13, 155, 313, 334, 375, 385
  stty, 205
POSIX.1, 367
power failure, 307
Practical Programming in Tcl and Tk, 20
precedence, levels of, 28
predefined spawn ids, 245
  any_spawn_id, 259
Prentice Hall, 20, 305
Press et al, 524
preventing bad commands, 346
Prime Time Freeware, 17
print array, 68
printf, 497, 521
proc, 34, 438, 445, 468-471, 473, 477-478, 480, 482-487, 489, 524
dynamically generating, 232
procedure, 34
  call stack, 414
  introduces new scope, 241
  recursive, 35
tiny ones are very useful, 242
process, 78
  bidirectional pipelines, 292
  identifier, 233
procomm, 289
programs
  as device drivers, 289
  reuse, 288
  that ignore eof, 103
prompt, 121, 125, 167
  changing, 120, 228, 232, 389
default, 467
  ftp, 190
  interpreter, 226-227, 516
password, 375
  by proxy, 201
patterns, 147
  vulgar pattern, 211
prompt1, 228, 362
prompt2, 228
protocol
  in-band error-reporting, 358
  negotiation, telnet, 395
proxy account, 357
ps, 105, 200
pseudo-random, 278
pseudoterminal, 292
pty, 292, 294, 305
  allocation, 303, 498
  hung, 302
  modes, 300
  three-step initialization, 301
  trapping, 510
ptyfix, 303, 305
puts, 27, 229, 284-285, 304, 412, 472, 478, 483
file, 61
   --nonewline, 61, 284
   versus send, 188, 283
   versus send_user, 182
pwd, 122
Python, xxv

Q
quack said the duck, 117
quadratic formula, 54
quit signal, 308, 312
quotes, xxxiii
quoting conventions, 25

R
r, 418, 423
race condition, 193
radiobutton, 436
random, 278, 524, 530
   number generator, 523
   password generation, 524
random_init, 524
range pattern, 91
rcp, when you cannot use, 351
--re, 109
read, 62, 495
   commands from file, 217, 220
   commands from standard input, 218, 224
   from script, one line at a time, 217, 224
readability, 90
readline (GNU), 226
real number pattern, 110
reap, 105
reasonable error, 279
reconnecting disconnected processes, 378
recursive
   ftp, 17, 133, 163, 336
   procedures, 35
Red Alert!, 475
redirection
   shell, 173
   standard I/O, 209
regexp, 107, 137, 175, 189, 364, 465, 482, 503
   --indices, 138
   to delete last character in word, 445
   versus string match, 530
regsub, 138, 216
   --all, 139
regular expression, xxviii, 107, 328, 332, 421, 502
   atom, 107
   branch, 109, 115
   caching, 503
   compilation, 503
   figure, 115
   further reading, 20
   limits, 158
   POSIX.2 style, 114
rehash, 72, 215
remote shell exit status, 117
remove_nulls, 155, 249
   --d, 156
remsh, 97
rename files, 70
repeaters, 15
reprompting, 193
Request For Comments, 80
reserved file name, 224
resize, 250, 431
retrieve an RFC, 17, 186
   character on input, 72, 79
   --code, 462
   from interact, 338, 340
   from interpreter, 227
   from procedure, 418
   missing, 345
   multiple values, 58
   value, 26
   overriding, 320
reuse, 288
reverse, 70
rewriting
   ftp, 22
   passwd, 5
   rewriting the #! line, 216
RFC
1073 (Window Size), 401
854 (Telnet), 80, 393
959 (FTP), 80
retrieve, 186
rftp, 17, 133, 163, 336
RHBOMB, 121
rlogin, 2, 126, 200, 316, 340, 351–352, 357, 402
daemon, 316
script, 122
versus rsh, 118
with same directory, 122
rlogin-cwd, 17
for telnet, 341
rlogind, 316
rm, 66
–f, 334, 475
rn
behaves oddly in background, 373
reprompting, 195
RNG, 523–524
Robbins, Arnold, xxviii
robohunt, 148
robust, 81
Rochkind, Marc, 305
rogue, 141–142, 458–459
rogue.exp, 17
Roman number pattern, 106
Rose, Marshall, 394
round, 27
routers, 14–15
rsh, 97, 126
exit status, 117
versus rlogin, 118
rtfm.mit.edu, 401
rup, 232
rz, 352, 369, 467, 489
over rlogin, 351

S
s, 423
Sandia National Laboratories, 15
sane, 158, 301, 497
Savoye, Rob, xxviii, 15
SB, 393
scalar, 51
scan, 46–47, 140, 487
regular to control character, 330
scanf, 140
Scheme, xxv
Schlumberger, 15
Schumacher, Dale, 20
scientific notation, 27
scope
current, 417
global, 413
identifier, 417
script, 180, 183, 259, 334, 337, 348, 463
design, 231
robust, 81
send_humany, 285
with command timestamps, 256
scriptless Expect programs, 518
scrollbar, 430, 437, 479
SE, 393
se, 450
search path, 287
security, 200, 202
by file protection, 202
by host protection, 203
problem, 434
testing, 14
seek, 62
select, 292, 303, 444, 508
select_next_nopassword, 443
semicolon, 24
SEND, 393
send, 71, 284–285, 433, 466, 471–474
–, 282
–break, 282
–h, 278
good typist, 280
humanly, 278
–i, 248, 271
long running, 276
not affected by log_user, 176
–null, 281
–raw, 285
-s, 277
slow, 275, 277
Tk, 409
Tk versus Expect, 433, 515

to multiple processes, 272

unknown strings, 283

versus puts, 188, 283

without echo, 273, 276

send -1, 352

send_error, 187, 373

send_human, 278, 281

send_log, 182, 377

send_slow, 277, 281, 285

send_tty, 210, 284, 466–467

send_user, 7, 185, 198, 284, 335, 337, 340, 373, 468, 470, 472

--raw, 198

translation, 345

versus puts, 182

sendexpect, 191

sendmail, 130

sequences, 448

serial interface, 351

not ready, 196


parallel, 70

setbuf, 495

Sex, xxvi

SGA, 393

shadow passwords, 5

Shaneys, Mark V., 288

shape, 279

SHELL, 468

shell

backquotes, 78

 Bourne, 178

echo, 119

 has peculiar behavior as daemon, 392

job control, 7, 372

limit, 150

prompt, 273

rc file, 11

 testing, 7

shell redirection, 173

shell-style, 91

Shen, Sam, xxix, 409

shortcuts, expect_before and

expect_after, 259

SIG prefix, 309

SIG_DFL, 310, 312, 315

SIG_IGN, 309, 314–315, 498

SIGALRM is reserved, 318

SIGCHLD, 308, 313–314, 366, 500

SIGCLD, 313

SIGCONT, 308, 316–317

SIGHUP, 308, 310, 314, 498

SIGILL, 313

SIGINT, 308–309, 311–312, 317, 319, 515

SIGKILL, 307–308, 312

signal, 274, 489, 498, 509

9, 307

 avoid if possible, 307

evaluation, 318

generation, 307

 handler, 308, 515

 problems, 319

 in spawned process versus

 Expect, 310

 power failure, 307

 sending, 307

 specifications, 307

signal.h, 318

SIGPIPE, 308, 310, 315

SIGQUIT, 308, 312

SIGSTOP, 308, 316, 318

SIGTERM, 308, 312, 498, 515

SIGTSTP, 308, 316

SIGUSR1, 308–309, 312, 317, 319

SIGUSR2, 308–309, 312, 317

SIGWINCH, 308, 315, 361

Silicon Graphics, 14

simulating 500 users simultaneously, 15

 simulations, 14

 sin, 27

 sins, covering up, 187

 SIPP, 69

 slave, pty, 294

 sleep, 99, 195–197, 277

 UNIX versus Expect, 530

 slowing rotation of the Earth, 528

 smart terminal, 448

 emulator, 448

SMTP, 130, 175, 494

 connect to port, 392
software
  distributions with Expect, 17
  interrupt, 307
    signal, 311
  quality assurance, 14
  termination signal, 308, 312
  testing, 15
Software Solutions in C, 20
sort, 436
source, 36, 68
  $expect_library/expect.rc, do not, 218, 221
  ~/.expect.rc, do not, 218, 221
Sparc, 440
spawn, 71, 78, 284, 287, 337, 351, 388, 488, 523
  /bin/sh –c for redirection, 292
  absolute versus relative names, 287
  catching errors, 296–297
    –console, 331
eval, 99
functions, 494
hints, 99
  –ignore, 310, 315
in the bedroom, 78
  –leaveopen, 290
multiple processes, 233
  –noecho, 298, 368
  –nottycopy, 301
  –nottyinit, 302
  –open, 289, 292, 304, 379, 482, 526
tty device, 290
  –pty, 294, 364
redirect or pipeline, 291
sh –c, 466
telnet, 482
  within a procedure is tricky, 100
spawn id
  converting to file identifier, 304
direct, 268
indirect, 268, 359
integer representation, 239
lists, 254
multiple, 358
predefined, 245
separate input buffers, 249
useful as array index, 239
spawn_id, 233–234, 238, 242, 247, 274, 360, 407
  affects many commands, 238
job control, 240
read implicitly, 241
returning, 244
versus –i, 248
writing, 243
spawn_out(slave,fd), 295
spawn_out(slave,name), 294, 316, 364, 368
Spencer, Henry, xxviii–xxix, 16, 20
split, 45–46, 64
  /etc/passwd, 45
  filename, 45
  versus gets in a while loop, 440
splitting strings into lists, 45
sprintf, 46
SQL, 69
  patterns, 135
sqrt, 27
sscanf, 46
stack, 414
  evaluation, 405, 410
  moving up or down, 417
Standard C, 155
standard error, 172, 245
  sending to, 187
standard I/O buffering, 7
standard input, 72, 245
  redirection, 209
standard output, 172, 245
  disabling, 174
  redirection, 209
    from shell, 173
standout mode, 451
startup, 505
static variables, 59
stderr, correct use of, 188
stdio.h, 493, 497
stepping into procedure calls, 415
stepping over multiple commands, 415
stepping over procedure calls, 413
Stevens, W. Richard, xxix, 394
stop signal, 308, 316
strace, 405, 408
  −info, 406
strategy
  for catching errors, 225
  script design, 231
strftime, 528
string, 46
  append, 49
  command, 47
  compare, 461, 466, 479, 481
  everything is a, 23
  first, 48
  formatting, 46–47
  index, 48
  last, 48
  length, 48
  match, 47, 91, 136, 485
  match versus regexp, 530
  parsing, 46
  range, 48
  reverse, 70
  tolower, 48
  toupper, 48
  trim operations, 48
  trimright, 78, 112, 140
string manipulation, 46
strlen, 521
strobes, 460
struct exp_case, 506
struct timeval *, 508
stty, 198, 204, 206, 251, 290, 302, 315, 325,
  364, 372–373, 471
<, 290
  −a, 207
  arguments, 205
  columns, 315
  −echo, 199, 205, 274, 290, 294
  echo, 199
  −icrnl, 72
  istrip, 291
  −ocrnl, 348
  POSIX, 205
  −raw, 468, 470
  raw, 290–291, 294, 470
  rows, 315
sane, 301, 377
stty_init, 301
  −istrip, 302
su, 2
  with logging, 183
su2, 178, 200
subpattern, 110
sum, 39
Sun, 440
Sun User Group, 17
superuser, 442
support, 21
suspend
  character, 316
  signal, 308
SV, 449
switch, 32, 133, 137, 231, 477, 484, 501, 503
syslog, 377
system
  call tracing, 408
  inactivity monitor, 330
  indigestion, 153
  versus exec, 207
sz, 352, 369, 467, 489
  over rlogin, 351
T

tag2pid, 362
talk, 331
Tcl, 11
  abbreviation, 11
  archive, 20
  best and worst things about, 70
  extensions, 21, 513
  idea database, 21
  library, 493, 514, 517, 530
  Makefile, 514
  philosophy, xxv
  pronunciation, 3
  reference material, 19
  shell, 26, 513
  source directory, 513
  total access to, 231
Tcl and the Tk Toolkit, 19, 28, 69
tcl.h, 514, 517
TCL_ERROR, 513, 521
Tcl_Eval, 227
  argument restrictions, 518
Tcl_Init, 513–514, 516
tcl_interactive, 223
Tcl_Interp, 520–521
tcl_prompt1, 516
tcl_prompt2, 516
tclAppInit.c, 513–514
Tcl-DP, 513
TclRegComp, 503–504
tclRegexpError, 504
tclsh, 26, 513, 515, 519
  –f, 515
      interpreter, 516
TclX, 69
TCP, 467
TCP/IP Illustrated, Volume 1, 394
tcp_wrapper, 400
Tektronix, 14
telephone exchange data collection, 14
tell, 62
telnet, 2, 8, 14, 126, 130–131, 166, 169, 196,
  204, 250, 325, 340, 349, 384, 392,
  397–398, 467–468, 496, 519
  behaves oddly in background, 373
daemon, 392
      written in Expect, 371, 393
escape character, 230
file transfer over, 467
IAC, 393
needs code to force exit, 239
protocol, 393
      negotiation, 395
      record session, 402
slow, 152
style escapes, 336
to all different ports, 162
to multiple hosts, 239
telnet-cwd, 341
ten_chars_per_sec, 276
TERM, 373, 450
term_chars_changed, 454, 460
term_clear, 452
term_cursor_changed, 455
term_down, 452
term_exit, 455
term_expect, 458–459, 463
term_init, 451
term_insert, 453
term_standout, 451
term_update_cursor, 455
TERM_CAP, 450
termcap, 286, 449, 452
terminal, 372
      ANSI, 462
      controlling, 372
      emulate multiple, 463
      emulator, 17
      scroll bars, 462
      testing, 458
identification code, 250
mode, 226
parameters, 158
servers, 15
type tk, 450
terminal driver, 72, 119
      buffers, 197
      echo, 273
      limitations, 274
      translations, 274
terminal-generated stop signal, 316
TERMINFO, 450
terminfo, 286, 449–450, 452
testing, 440
      GNU C and C++, 15
      interactive programs, 6
      security, 14
      shell, 7
      software, 15
Testing Foundations, 17
text, 437
      widget, 451, 463
Thompson, Ken, 12, 235
tilde, 288
time, 529
timelcmds, 256
timed-read, 17, 77, 112, 126, 218
timed-run, 17
timeout, 3, 94–95, 242, 249, 327, 343, 353,
  483
      –1, 81
      as local variable, 488
infinity, 344
literal pattern, 106, 110
no association with spawn id, 253
none, 81
read implicitly, 241
variable, 75
timestamp, 257, 303, 528
versus date, 530
timing
a read in the shell, 77
dictionary loading, 522
other commands, 529
out, 74
problems, 153
tip, 4, 196, 282, 288-291, 351, 398, 468
Tk, 10, 283, 409, 429, 432
devicer, 428
overview, 430
send, 409, 433
suppressing button highlighting, 222
terminal emulator, 511
widget hierarchy, 295
tk
colormodel, 437
terminal type, 450
tk_entryBackspace, 445
tk_entrySeeCaret, 445
tk_strictMotif, 222
tkinspect, 409
TkMan, 19
tknewsbiff, 17, 475
tkpasswd, 17, 434, 463
figure, 435
TkSteal, 295, 513
tkterm, 17, 449
example tests, 454
tkwait, 442, 461
todd, bennett, xxix
tool command language, 11
toplevel, 438, 442
tput, 283
tr -d \015, 348
trace, 53, 408
procedure, 423
tracing, 403
command, 405
system calls, 380, 408
variable, 406
transcript, 305
trap, 308
-code, 320
-interp, 321
-max, 318
-name, 309
-number, 309
pty, 510
SIG_DFL, 310
SIG_IGN, 309
truss, 380, 408
tset, 147
TSR, 348
TSS HAS GONE DOWN, 121
tty_spawn_id, 245, 372
unset, 375
ttyp0, 294
TTTYPE, 393
typeahead, 195
typescript, 334
U
u, 417
U.S. Department of Commerce, xxviii
UART not ready, 196
unbuffer, 17, 299
uncompress, 81, 473
underlining, 451
underscore convention, 478
unencrypted passwords in files,
NOTI, 203
unhang a printer, 17
uniquely-named file, 70
University of California at Berkeley, xxvi, 16
University of Toronto, 16
University of xxx xxxxxxx
xxxxxxx, 397
UNIX, 14, 375
versus VMS, 154
unix, 328
UNIX epoch, 528
unknown, 68, 226, 362
unread news, 475
unset, 53, 367, 462, 489
uplevel, 59, 241, 417, 458, 462, 480, 485
upvar, 57, 241, 244, 417
    #0, 482
USENET Software, 17
user_read, 231
user_spawn_id, 245, 354, 356
user-defined signal, 308, 317
users, contacting multiple, 382
uencode, 471–472
Unnet, 80, 130, 170

V
variables
    information, 51
    static, 59
    tracing, 53, 406
    unsetting, 53
varying argument lists, 39
verbose mode, 404
verify mail address, 175
versions of Expect, 530
    testing, 527
vi, 344
    autoabbreviation, 326
Virgen, Larry, 20
vmesh, 328, 333
VMS, 330
    versus UNIX, 154
vms, 328
vprint, 404
vrfy, 175
vt100, 395
vulgar prompt pattern, 211
vxWorks board, 348

W
w, 416–418
    -compress, 428
    -width, 426
wait, 105, 249, 313, 466, 483
    -nowait, 106
    return value, 313
wait for prompts before sending, 79, 81
waiting for a process, 105
watch, 476
weak_password, 440
weather, 17
Weibull distribution, 278
Welch, Brent, xxix, 20
What does that have to do with
    news!? , 480
where am I, 416
while, 29, 231, 339, 479, 482, 484
widget, 430
    classes, 431
    display, 431
    exposure, 431
    terminal emulator, 511
WILL, 393
Willison, Frank, xxix
WINCH, 367
window
    change event, 431
    size change signal, 308, 315
WINTERP, 429
wm, 437, 477–478, 481, 487
wonders of UNIX, the, 318
WONT, 393
World Bank, 14
World Wide Web, 20
WorldCat, 398
write, 241, 492, 495
    to multiple people, 237
WWW, 20, 397

X
X, 315, 361
    authority file, 124, 126
    client –display flag, 433
    for interprocess communication, 410
    GUI builder, 432
    resize, 250
    window id, 295, 364
X Consortium, 295
X Window System, 10, 222, 283, 428–429
X Window System Series, 429
X11
    R5, 294, 367
    R6, 295
Xerox, 14
Exploring Expect

Expect is quickly becoming a part of every Unix user's toolbox. It allows you to automate telnet, ftp, passwd, rlogin, and hundreds of other applications that normally require human interaction. Using Expect to automate these applications will allow you to speed up tasks and, in many cases, solve new problems that you never would have even considered before.

For example, you can use Expect to test interactive programs with no changes to their interfaces. Or, you can wrap interactive programs with Motif-like front-ends to control applications by buttons, scrollbars, and other graphic elements with no recompilation of the original programs. You don't even need the source code! Expect works with remote applications, too. Use it to tie together internet applications including telnet, Archie, ftp, Gopher, and Mosaic.

Don Libes is the creator of Expect as well as the author of this book. In Exploring Expect, he provides a comprehensive tutorial on all of Expect's features, allowing you to put it immediately to work on your problems. In a down-to-earth and humorous style, he presents numerous examples of challenging real-world applications and how they can be automated using Expect to save you time and money.

Expect is the first of a new breed of programs based on Tcl, the Tool Command Language that is rocking the computer science community. This book provides an introduction to Tcl and describes how Expect applies Tcl's power to the new field of interaction automation. Whether your interest is in Expect or interaction automation or you simply want to learn about Tcl and see how it has been used in real software, you will find Exploring Expect a treasure trove of easy-to-understand and valuable information.

"Expect was the first widely used Tcl application, and it is still one of the most popular. This is a must-know tool for system administrators and many others."

—John Ousterbou, creator of Tcl

"Expect is an absolutely wonderful, marvelous program. It is one of the most useful tools I've seen in 15+ years of Unix backing. Expect is going to save us several thousand dollars in licensing fees in the next year alone, some inestimable number of programming hours, and allow us to provide our users much better service than we otherwise could have."

—John W. Pierce, Department of Chemistry and Biochemistry
University of California, San Diego

"It is a mystery to me how Unix could have existed for years without Expect."

—Erik Basilier, Motorola