Advanced Programming in Expect:
A Detailed Analysis of Internal Diagnostics

by David L. Fisher

http://dlf.cotse.net/papers/A_Detailed_Analysis_of_Internal_Diagnostics.pdf

The internal pattern-matching attempts of the Expect shell's regular-expression matching engine are critical to properly debugging Expect programs, and especially for finding the more subtle and elusive bugs. This article will cover diagnostics output for various situations and explain exactly what is going on with the pattern-matching program flow, and why seemingly valid approaches to handling these situations fail. Examples will include fast connections, slow connections, unhandled timeouts, and proper programming techniques to make Expect manage the internal buffer properly, and to make your automated systems as reliable as possible.

Behavior of the expect buffer and buffer index:

When the expect clause matches an expression, everything that was received on that channel up to the end of the match, is written to **`expect_out(buffer)`**. Once that is done, the index of the input buffer is moved to the end of the part that matched, so that the next time Expect attempts to match, it considers the data from the previous match to be gone (and in many respects, it is). For example, suppose a remote system sends back the output:

```
abcdefghijklmnopqrstuvwxyz
```

And we have the following **`expect`** statement:

```
expect {
    -i $id
    -ex "abcd"  {exp_continue}
    -ex "lmnop" {exp_continue}
    -ex "efgh"  {exp_continue}
    -ex "wxy"   { }
}
```

This **`expect`** statement is improperly designed and will behave depending on the speed of the connection relative to the spawned process. The simplest solution to this dilemma is to switch the order of the second and third expression so that they match in the desired order:

```
expect {
    -i $id
    -ex "abcd"  {exp_continue}
    -ex "efgh"  {exp_continue}
    -ex "lmnop" {exp_continue}
    -ex "wxy"   { }
}
```

Note: without the **`exp_continue`** command in the action list, the first match in the above statement ("**`abcd`**") would cause program execution to drop out of the **`expect`** clause, to the next line of code in the script. The following examples illustrate both the behavior of the internal buffer, and the flaw with the first **`expect`** statement, on fast and slow connections.

Example of matching attempts on a slow connection:

This is an example of how Expect would try to match the alphabet string with the first **`expect`** clause (where "**`lmnop`**" precedes "**`efgh`**") on a slow connection, where it receives one character at a time:

```
expect: does "a" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? no

expect: does "ab" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? no

expect: does "abc" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? no

expect: does "abcd" (spawn id 123) match exact string "abcd"? yes
expect: set expect_out(0,string) "abcd"
expect: set expect_out(buffer) "abcd"
expect: set expect_out(spawn_id) 123
expect: continuing expect
```

Once the match is made, the index is advanced to the end of the match, effectively removing the matched part of the input from the buffer. Since the **`expect`** clause is still in effect and more data is coming, the diagnostics output would continue:

```
expect: does "e" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? no

expect: does "ef" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? no

expect: does "efg" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? no

expect: does "efgh" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? yes
expect: set expect_out(0,string) "efgh"
expect: set expect_out(buffer) "efgh"
expect: set expect_out(spawn_id) 123
expect: continuing expect
```

Obviously, if the Expect script is receiving input at this rate, the expressions will match regardless of the order they are listed in the expect statement. However, if the connection is fast (see next example), this logic breaks down.

Example of matching attempts on a fast connection:

This is an example of the way Expect would try to match the alphabet string on a fast connection, where the entire string was received at once (or more accurately, within a single cycle of Expect's polling loop):

```
expect: does "abcdefghijklmnopqrstuvwxyz" (spawn id 123) match exact string "abcd"?
yes
expect: set expect_out(0,string) "abcd"
expect: set expect_out(buffer) "abcd"
expect: set expect_out(spawn_id) 123
expect: continuing expect
```

```
expect: does "efghijklmnopqrstuvwxyz" (spawn id 123) match exact string "abcd"? no
"lmnop"? yes
expect: set expect_out(0,string) "efghijklmnop"
expect: set expect_out(buffer) "efghijklmnop"
expect: set expect_out(spawn_id) 123
expect: continuing expect
```

Again, the index of the input buffer is advanced to the end of the match, only this time we lose more data than we intended to. Since the expressions in the **expect** statement were improperly ordered, we skipped over part of the string we were hoping to match with a different expression. Since there is more data in the input buffer, the diagnostics would continue to show the matching attempts, but it should become clear that "**efgh**" will never be matched, and its associated action block will never be executed.

```
expect: does "qrstuvwxyz" (spawn id 123) match exact string "abcd"? no
"lmnop"? no
"efgh"? no
"wxy"? yes

expect: set expect_out(0,string) "qrstuvwxyz"
expect: set expect_out(buffer) "qrstuvwxyz"
expect: set expect_out(spawn_id) 123
```

At this point in execution, the buffer contains the "**z**" only (assuming no further input was received), and since the actions list for matching "**wxy**" was empty, we drop out of the expect clause and never match "**efgh**".

Note: Expect works asynchronously - an **expect** statement will execute whenever there is any data in the input buffer, and you can not make any assumptions about how much data is received before execution starts. This is why it is important to fully specify an expression so it defines only what you intend to match.

Example of the effect of unhandled timeouts:

The following example looks to be correct Expect code, except that without a timeout handler defined (ie, in the **expect_after** statement), this code only works when it works. For this example, diagnostics output are provided so you can see what went wrong: (Note: "**% **" is the prompt we expect to see from the remote system once we're logged in; also, the carriage-return is translated to a carriage-return/newline sequence in raw mode – in cooked mode it shows up as a **control-M**, or **^M**):

```
spawn telnet myhost;
expect "Login:"
exp_send –i $spawn_id "loginname\r";
expect "Password:"
exp_send –i $spawn_id "mypassword\r";
expect -i $id "% ";
```

This code sequence, which closely mirrors an example in the book, will work when everything works. The problem is when something doesn't work, this code will not detect the error and continue to execute as if there is no error. When code like this breaks, usually it results in the rest of the program sending its entire interaction sequence to the **telnet** prompt.

Now, if we look at the diagnostics for this sequence, we might see:

```
spawn telnet myhost;
parent: waiting for sync byte
parent: telling child to go ahead

expect: does "Unknown host myhost\r\ntelnet> " (spawn id 123) match glob expression
"Login:"?  no
expect: timed out
```

```
send: sending "loginname\r" to 123
expect: does "Unknown host myhost\r\ntelnet> loginname\r\nUnknown command
"loginname"\r\ntelnet> " (spawn id 123) match glob expression "Password: "? no
expect: timed out

send: sending "mypassword\r" to 123
expect: does "Unknown host myhost\r\ntelnet> loginname\r\nUnknown command
"loginname"\r\ntelnet> mypassword\r\nUnknown command "mypassword"\r\ntelnet> "
(spawn id 123) match glob expression "% "? no

expect: timed out
```

At this point, the Expect script thinks it has successfully logged in, since the timeouts went unhandled and the logic of the code allowed it to advance regardless of errors. In other words, we specified explicitly what we were looking for (the sequence of prompts to login, and then the command prompt indicating we are logged in), and we explicitly or implicitly specified how long we were willing to wait (the global **`timeout`** variable, which defaults to 10 seconds if not modified), but we didn't specify what to do if our expectations are not met.

The rest of this script will fail on timeouts, since the only replies will come from the telnet client program.

Expecting echo from a remote process

Often times, it is useful to expect the echo a string that was just sent, to ensure that we don't inadvertently match the echo of the string against our expected response. Many validation methods, including the **`send_expect`** interface detailed in a different article, depend on remote system echo. Simply sending a series of commands without checking at each step of the way is akin to typing in commands at a terminal without looking. Usually, you would look at the command you are about to execute just before hitting the enter key (which, in most cases in Expect, is the carriage-return at the end of the line), to make sure you spelled everything right, and to make sure everything was received by the remote system. The exception to this rule is for listening loops which only want a single character, such as getc and getchar (from various languages).

Using timeouts as a benefit:

Timeouts are exceptional conditions in Expect, and should be treated as such. In a properly designed program, a timeout almost always indicates a situation that was not anticipated by the developer. Although it may seem sufficient to specify a single expression representing what you are looking for, and allowing timeouts to indicate that your expectations were not met, you wind up introducing a number of problems with this approach.

Most importantly, timeouts mean that the predefined timeout period has elapsed. If you are running tests and the device under test is failing, it means that its output is not matching your **`expect`** expressions – what you do next is critical to how your system performs. Assuming your expressions are correct, and your heuristics for determining success are correct, this is an actual failure in either communications with a remote system or failure of the commands on the remote system to behave as anticipated.

Since you don't match any expressions that indicate things you think might happen, obviously something you weren't expecting has happened. The **`expect`** statement times out, and the associated action is executed, and this is where you declare the failure. You're probably wondering what's wrong with this approach. You're probably also wondering why your program takes so long to execute. This approach has a built-in requirement that failure detection introduces a delay, for each failure, of the current value of the timeout used by the **`expect`** statement that detects each failure. In other words, if you left the **`timeout`** at its default value of ten seconds, and detected 60 failures throughout the course of a test using timeouts as indicators, you've wasted ten minutes during execution on timeouts.

Additionally, this may or may not be because the remote system is failing to behave as expected. There are times when an **`expect`** statement can only look for error messages (such as when a UNIX-standard program exits quietly with a zero return value) and when no errors are found, an assumption can be made

that no error messages were issued. In cases such as these, it is beneficial to examine the return status as indicated by the shell's built-in variable (in csh, it is called **status**, in PERL it is called **$_**, etc).

There are a number of things that may happen that are neither failures of the remote system or errors in your program, but can adversely affect the outcome.

Error messages and prompts

Let's say you want to run a command which only outputs messages if there are errors, otherwise it runs quietly.

```
% rm *.o
%
```

If there's an error, the exchange might look more like:

```
% rm *.o
rm: No match
%
```

In both cases, the prompt is re-issued, so simply looking for the next prompt is weak criteria. If our **expect** statement looks like:

```
exp_send -i $id "rm \*\.o\r";
expect {
    "% " { }
    timeout {
        ## timeout handler
    }
}
```

An **expect** clause like this will never catch error conditions, because the weak criteria for determining success is easily met. The internal diagnostics of the above clause with an error condition would look like:

```
send: sending "rm *.o\r" to spawn_id 123
expect: does "rm *.o\r\nrm: No match\r\n% " (spawn_id 123) match glob expression "% "?
yes
expect: set expect_out(0,string) "% "
expect: set expect_out(buffer) "rm *.o\r\nrm: No match\r\n% "
expect: set expect_out(spawn_id) 123
```

A better approach would be to set up an expression to match the possible error conditions first:

```
exp_send -i $id "rm \*\.o\r";
expect {
    -re "\r\n(.+)\r\n% " {
        ## expect_out(1,string) has the message
        ## issued before the prompt - check here
        ## if it's an error
    }
    "% " {
        ## this will now only match if there
        ## was no error message
    }
    timeout {
        ## timeout handler
    }
}
```

In the above example, the first carriage-return in the regular expression matches the carriage-return of the command. The newline is appended by the terminal handler (translation to carriage-return/newline sequence), then we specify one or more lines (at least one character), finally followed by another carriage-return/newline sequence and the prompt. Since we check this expression before we check for just the prompt, we won't accidentally match the second expression to an error message.

Note the order of the expressions: the expression to match an error condition comes before the expression to match a success condition. If the expressions were reversed, the same problem would exist as in the previous example, which did not check for any error condition.

Interpreting diagnostics for multiple processes

When dealing with multiple processes, the diagnostics output can be copious. The biggest reason for this is that, whenever there is any input from any channel, the diagnostics will report on the matching attempts made on all channels. Essentially, this amounts to quite a bit of output that only says that nothing matched, and a block of output that shows the match. For example, consider the following **`expect`** statement, executing without **`expect_before`** or **`expect_after`**:

```
expect {
    -i $id1
    "expression1" { action1 }
    "expression2" { action2 }
    -i $id2
    "expression3" { action3 }
    "expression4" { action4 }
    -i id3
    "expression5" { action5 }
    -i any_spawn_id
    "expression6" { action6 }
}
```

Let's say we get some input from the process on **`id2`**, and that it matches none of the expressions. The diagnostics would look like (Note: for simplicity here and throughout this book, wherever spawn ids are listed as **`id1`**, **`id2`**, and so on, their actual value will exactly match their names; this is not the way Expect assigns spawn ids, but should serve to help understand these examples).

```
expect: does "" (spawn id id1) match glob expression "expression1"? no
"expression2"? no
"expression6"? no

expect: does "bla bla bla" (spawn id id2) match glob expression "expression3"? no
"expression4"? no
"expression6"? no

expect: does "" (spawn id id3) match glob expression "expression5"? no
"expression6"? no
```

Notice the order of both the expression-matching attempts, and the expressions themselves, for each of the spawn ids. Expect has detected input in its buffer and has triggered the matching engine – since there was no input on **`id1`**, it shows an empty string. This is because the input buffer for that spawned session exists, but is empty. When Expect triggers the expression matching, it checks all spawned channels for the contents of their input buffers. This is independent of the status of null handling in Expect, and is a byproduct of way the input buffers are handled. Since the input was on spawn id **`id2`**, that is the only channel we are interested in.

Notice also the expressions that are listed for each channel. Since the **`expect`** statement specified which spawn id the expressions were meant for, they are only checked for that spawn id. The last expression is listed for any_spawn_id, an Expect global which refers to any spawn id listed thus far in the current **`expect`** statement (which, in this case, would be **`id1`**, **`id2`** and **`id3`**).

With a carelessly designed regular expression listed for any spawn id, the expression matching engine may go awry. Consider the following example:

```
expect {
    -i $id1
    "\[\r\n]*\[^\r\n]*\[\r\n]*" { action1 }
    -i $id2
    "data we want to match"     { action2 }
    -i id3
    "not really significant"    { action3 }
    -i any_spawn_id
    eof                         { eof_action }
}
```

In the above poorly-designed but well-formatted **expect** statement, the expression for spawn id **id1** is intended to match a single line of output, but due to its weak design, it will match much more or much less, depending on what is received. The expression for spawn id **id2** is the one I'm hoping will cause this expect statement to execute the action, but it will never happen. If we look at the diagnostics to see what went wrong, we will see when Expect checks its input buffer for spawn id **id1**, which is empty, the empty string is matched against the expression:

```
expect: does "" (spawn id id1) match glob expression "\[\r\n]*\[^\r\n]*\[\r\n]*"? yes
```

This is basically the same as the first line of the previous example, where the expression matching engine checks each input buffer in the order in which is listed in the **expect** statement, but now a match is made immediately. If you remember your regular expressions, this expression will match an empty string due to the careless use of the "*****" modifier on every atom. Zero or more matches are specified for each character class, and the empty string in the **id2** input buffer matches, sure enough, zero occurrences of each character class.

The same thing can happen in a far more subtle manner, also. If we change the order of the expressions in the previous **expect** statement, we introduce a bug that might not appear for quite some time:

```
expect {
    -i $id1 "not really significant"       { action1 }
    -i $id2 "data we want to match"        { action2 }
    -i $id3 -re "\[\r\n]*\[^\r\n]*\[\r\n]*" { action3 }
    -i any_spawn_id "something else"       { eof_action }
}
```

This example combines the careless regular expression with the timing problems discussed earlier in this chapter. When things are running smoothly, and the connection is fast (or, more accurately, when data appears in the input buffer in a single polling round), the above **expect** statement will work properly.

```
expect: does "" (spawn id id1) match glob expression "not really significant"? no
"something else"? no

expect: does "data we want to match" (spawn id id2) match glob expression "data we
want to match"? yes
```

Once again, we see the unfortunate example of "things work when they work". If this **expect** statement is used for a spawned process that has a slow connection, or simply the output is delayed long enough for the empty expression to match first, such as our earlier example of output being received one character at a time, the diagnostics would show something completely different

```
expect: does "" (spawn id id1) match glob expression "not really significant"? no
"something else"? no

expect: does "d" (spawn id id2) match glob expression "data we want to match"? no
"something else"? no

expect: does "" (spawn id id1) match regular expression ""\[\r\n]*\[^\r\n]*\[\r\n]*"?
yes
```

The same thing would happen if the expression had been listed for spawn id **id2** either before or after the expression we wanted to match.

While the data you meant to match is not yet lost at this point, it is almost inevitable that it will be lost on the next **expect** statement that matches on that channel. Let's say that the next **expect** statement is a simple match on that channel:

```
expect -i $id2 "more data" action
```

Let's assume that the process on **id2** sends back the line "Here is some more data", which will match the above expression. Since the earlier data we wanted to match is still in the buffer, the internal diagnostics for this expression would look like:

```
expect: does "data we want to match\r\nHere is some more data\r\n% " (spawn id id2)
match glob expression "more data"? yes
expect: set expect_out(0,string) "more data"
expect: set expect_out(buffer) "data we want to match\r\nHere is some more data"
```

Line matching and the match_max variable

The **match_max** variable defaults to 2000 bytes, which is coincidentally a full 80x25 terminal screen (and this is really just a coincidence – Curses, I say!) Often times this isn't enough to check an entire output buffer for what we expect to see, and developers decide to increase this value to allow for the capture of all of the output. For example, if we want to parse the message of the day (**motd**) before our initial prompt after logging in, we would need to allow for an arbitrarily sized buffer for the message. The drawbacks to this approach:

- the input buffer grows to unreasonable size and makes debugging very difficult as the diagnostics output continues to reflect the entire contents of the input buffer
- the program slows down as expression matching is applied against an ever-growing string
- if the **match_max** variable is still too small and more output is received, the first part of the output is lost in a first-in-first-out (FIFO) buffer

I'm not going to provide examples of this as it would take up a lot of space in this article, but those of you who have inspected the diagnostics have likely already seen this effect.

The easiest way to handle large output strings where the significant data is relatively small and either embedded or appended to the output is to process the output one at a time. For example:

```
expect {
    -i $id
    -re "^% $" {
        ## just the prompt, with nothing before or after
        ## we already know we captured everything including the
        ## carriage return/newline sequences, so this indicates
        ## that a command prompt is waiting
    }
    -re "(\[^\r\n]*)\[\r\n]+" {
        ## this is a line of output and not a prompt (unless the
        ## system issues prompts terminated by newlines)
        ## handle the output (if required) and stay in the clause)
        exp_continue;
    }
    timeout {
        ## no output from the remote system - handle here
    }
}
```

In this example, the input buffer is kept small as each line is captured and processed one at a time. For an arbitrarily long message, insignificant lines of data can be checked and discarded on the fly, and the regular expression engine is only matching small chunks of data. The first expression defines a prompt with nothing following it, and is usually sufficient (except for slow systems where the output may include a prompt and subsequent command string, which hasn't yet appeared). The second expression defines a line of data, possibly empty, terminated by a newline sequence.

Note: it's a good idea to parametrize the prompt for reasons beyond the scope of this article, and in most cases it's a simple application of the spawn id as an array index to the expected prompt for that channel, so instead of literally using "`%` " in each **`expect`** statement you may prefer to use something like **`$prompts($id)`**, and defining the array at runtime.

Once a line of output is captured, expressions can be applied against it for matching expectation. If there is something that spans multiple lines, this approach would have to be modified to allow for the receipt of multiple lines before handling each line by itself. While it is possible to craft very complex expressions, there comes a point where simplicity is preferable and actual code in the procedure is more appropriate.