Advanced Programming in Expect: Prompt Your Prompts

by David L. Fisher

http://linuxgazette.net/2013/01/advanced-programming-in-expect-prompt-your-prompts
http://dlf.cotse.net/papers/Pompt_Your_Prompts.pdf

The critical aspect of the command prompt is to ensure that an automated system is in sync with the remote process it is controlling. A command prompt is meant to indicate that there is a process waiting for commands or other inputs. Duh. Accurate prompt detection becomes very important to the reliability of an automated system because without knowing whether or not you have captured the most recent prompt and the remote process is actually waiting for input, your system might wind up sending data to a system in an unknown state, a condition likely to cause failures and errors and possible even exceptional exits. Typically the cause of losing sync with the current prompt is that the prompt captured is the one before the command was issued – this results in an automated system thinking it has received the output of a completed command and the remote session is ready for the next command.

Maintaining Synchronization With Remote Session Prompts

This article will thoroughly address the issue of prompt detection regardless of the factors that may confound ordinary prompt detection and verification, and allow automation developers to focus on other tasks instead of debugging what can be an intermittent error. For those of you who intend on working with the code, I can't emphasize this enough: always wear eye protection.

Can You Control The Prompt?

Once common mistake developers make is to use a generic regular expression using common characters used as the last non-space character of the prompt, and anchoring the expression. For example, consider this prompt (and note the lack of escape characters within braces:

```
{[\r\n]?[^\r\n]+[%#>$] $}
```

This is the same expression in quotes, with which we'll need to be familiar, since the braces will prevent shell evaluation of the variables we want to use within the expression, so shown below are the escape characters in place:

```
"\[\r\n]?\[^\r\n]+\[%#>\$] \$"
```

This defines a string where there may be a line break, then a string which terminates in a common prompt character and a space, and that space must be the last character. This is a reasonable definition and works in most cases, until it stops working.

Besides the fact that prompts may be set to something that does not adhere to the standard prompt expression, there are also instances in which you may thing you got the prompt, but instead matched something within the incoming data. Even on very fast connections, if you enable diagnostics and watch Expect's internal matching attempts, you may see that it is processing one character at a time, and you can watch the input buffer grow until a match is made. When dealing with cell modems and other slow connections, it is possible (I know from personal experience) that the data may appear to stop and the last part if it may match the above expression. Your expect clause will exit, thinking it has captured the next command prompt, when in fact your system should still be iterating the expect clause and waiting for the real prompt.

The above condition is even possible no matter how much you can control the prompt, and you may also have to deal with systems that don't allow you to set the prompt.

In situations where the prompt itself is part of the returned data (such as if you **`printenv`** on the remote system), and you are able to control the prompt output, there is one trick that will help many of the subtle

mistakes that prompt expressions cause. By setting the command prompt to include the command number of the command you just executed, you can expect a prompt with a number one higher than that number. For example, in **csh**, adding the parameter "**%h**" will output the command **history** number. You can then set your expected clause to include a number two higher (see below) as the next command prompt, and avoid the confusion of seeing the last prompt. The actual arguments you would use in your prompt settings, and how you would choose to get this number (either by regular expression to capture it or using the history command) is up to you.

The following section applies to handling basic command shells like bash, sh, csh, etc. The section after that applies to any prompting at all.

A Solution for Spawned Shells

For sessions which invoke something beyond just shell commands, we can't rely on this solution unless we want to specify the prompt we expect. For example, the history number would not show if you were to invoke any program that prompts (like telnet).

I use **csh** when I spawn sessions, so the following example will only work for **csh/tcsh** users, and I've added other interesting data (the **%h** in the prompt setting is the command number in the shell's **history**, and it increments every time a command is executed but not when blank lines or carriage returns are sent):

```
set prompt = "\[%h\]$USER@${HOST}:`pwd`> +"
```

This will result in a prompt that looks something like this:

```
[223]root@cyborg:/usr/local/etc>
```

Before we issue the command from whice we're expecting some output, we should know what the **history** number will be of the next prompt. A simple shell command will suffice:

```
set number = `history | tail -1 | awk '{print $1}'`
```

This is the command number of the command we are going to send, and incrementing it by one will give us the command number that we expect to see in the next command prompt:

```
"\[\r\n]?\\[$number\\]\[^\r\n]\+[%#>\$] $"
```

The above expression assumes that "number" is the calculated value of the next command prompt to expect. This solution should be elegant enough to stand the test of time, but only when dealing with systems where you can set the prompt to include history numbers. An alternative would be to use unique numbers and set the prompt every time before sending the command, and then using that unique number to determine whether it is the same as the one you sent the command to – if so, your **expect** clause should continue executing. In TCL8.4, you could use "**clock clicks -milliseconds**" to get the number, but this command has been deprecated and in TCL8.5 beta the command would be "**clock -milliseconds**".

Following is an example in TCL to send a command and make sure your expect clause doesn't exit earlier than it should.

Is This Really The Prompt I've Been Expecting?

Suppose you are unable to create a unique expression for each time you are waiting for the next command prompt. This is where it gets annoying and when automated systems get derailed.

If we're reasonably sure we've captured the most recent prompt and the remote process is actually prompting for more commands, we may want to make sure. If we couldn't do enough with the prompt to be reasonably sure it's the most recent, then we really want to make sure. If we don't even know what the prompt is, then we have to find out and verify it. Like most philosophies about handling multiple remote command sessions, we often hit the return key once or twice to make sure the system is still responsive – after we've determined that the system isn't waiting for an answer to a question. If it's waiting for an answer, and especially if it provides a default if you simply hit the enter key, then this strategy won't work. Therefore,

besides just detecting command prompts, we must have something to determine whether we're being asked a question instead.

The following example is for optionally discovering and definitely determining what the current prompt is. It is a re-entrant procedure for discovery, and only runs once for verification.

```
proc check_prompt { id {prompt ""} {wait 0} {timeout 1} } {
     ## we have to determine the prompt, and we may have to wait first
     after $wait;
     ## find out if the session is responsive
     exp_send -i $id "\n";
     ## process what we get back as a response
     if { "$prompt" != "" } {
        ## we are looking for a particular prompt to verify
        ## this can be changed to allow flexibility if needed
        expect {
            -i $id
            -t $timeout
            -exact "($prompt)$" {
                ## partial or complete prompt matched with anchors
                return $expect_out(1,string);
            }
            timeout {
                error "Timeout checking prompt against $prompt";
            }
        }
     } else {
        ## use a fairly well-crafted regular expression to find a prompt, and
        ## then re-enter this procedure with a verification request
        expect {
            -i $id
            -t $timeout
            -re {your-(well-crafted-expression)} {
                ## you may not feel the need to verify this one
                return $expect_out(1,string);
            }
             -re "\[\r\n](\[^\r\n\]+\[%#>>\$] ?)$" {
                 ## verify the suspected prompt
                 return [check_prompt $id $n $expect_out(1,string)];
             }
             timeout {
                 error "Timeout checking prompt";
                 return "UNKNOWN";
          }
     }
}
```

Note the first argument is the **spawn_id** – Expect programmers should get used to passing this variable if they are handling multiple spawned sessions in the same procedures.

If this procedure is called with just the **spawn_id**, it will attempt to determine the current prompt and then call itself with the prompt on the command line to verify it, and then return the prompt or throw an exception. If this procedure is called with both the **spawn_id** and the n parameter (the next history command number), the we can more precisely define an expression that should exactly match the command prompt we're expecting. If this procedure is called with a **prompt** argument, it will either return the very same prompt or throw an exception.

But Wait – There's More

So you don't forget, order before midnight tonight! The solutions presented above have proven to be invaluable in keeping an automated system in sync with a remote command prompt, and ultimately preserve a sense of high confidence in the results produced by any automated system. There are still some instances where even more is required, but this article just covers the basics – hopefully you'll be able to create your own more esoteric solutions. Sometimes a threaded program will return to the command prompt while one

or more of the threads have not yet completed, and when they do, they will report to the same **`stdout/stderr`** that the original program had.