The BNR Standard Login (A Login Configuration Manager)

Christopher Rath – Bell-Northern Research Ltd.

ABSTRACT

Several years ago, Bell-Northern Research wrote a login configuration system for their UNIX workstation user community. While it performed adequately at its release, it was very quickly overcome by the user community's needs. This configuration system, called the BNR Standard Login, was recently re-written. This paper describes its past and present states; making a case for the continued existence of a login configuration manager in BNR's ever-expanding multi-vendor UNIX workstation network.

Introduction

Early in 1989, Bell-Northern Research (BNR) began to widely deploy UNIX workstations on users' desks. As more workstations were purchased, system administration became more difficult; of particular concern was the distribution and installation of software updates.

As a result of complaints from users, the UNIX release team became concerned with the configuration burden being placed upon users by new releases of the software: when a new version of a tool was released, the user was often required to change their configuration files in order to use the new software. Feedback from users indicated that they did not like having to make the changes, and disseminating these changes was difficult.

In order to overcome these problems, the BNR Standard Login System (hereafter referred to as *the standard login*) was devised. The standard login consisted of a shell script sourced from the user's `.login` or `.profile` file; this shell script configured the user's environment for all supported applications and allowed safe, automatic deployment of configuration changes to the user's environment; since everyone sourced the same central script.

Standard Login v1.x Components

Version 1.x of the standard login consisted of several files (and a set of files):

`bnrsetup.[c]sh` – the script which served as the **main** procedure of the standard login. It was sourced from either the user's `.login` or `.profile` file and was responsible for executing each setup script and **eval**ing its output.

`macros` – a list of setup scripts which *might* be resident on the system. Each entry in the file was the absolute path of a script.

`.bnrnosetup` – a list of the setup scripts which the user did not want the standard login to execute.

`.bnrsetuplog` – the output log created by the standard login whenever an error occurred.

`.bnrrc` – a mini name space where applications could store user specific application defaults. It was also where users would enter parameters to the setup scripts.

`os.setup` – the first setup script to be executed by the standard login; always the first entry in the `macros` file.

`X11.setup` – the last setup script to be executed; always the last entry in the `macros` file.

`wga.setup`, `pub.setup`, ... – each of the other setup scripts had a corresponding entry in the `macros` file.[1] When run, they emitted the shell code used to configure a user's environment.

Standard Login v1.x Processing

The standard login, prior to the re-write completed for this project (i.e., version 1.x), did its work as shown in Figure 1, *Standard Login Version 1.x Process Flow.*

As originally implemented, the standard login was run by placing a source command at the bottom of your `.profile` or `.login` file. This would set up the process environment space and then start X windows. Users could add their own customizations through a parameter list stored in their `.bnrrc` file.

The processing was quite straight-forward: each setup script listed in the `macros` file was examined in turn. If the user had requested that the script be skipped, then the standard login went on to the next entry in the `macros` file; otherwise the setup script would be run and any output from the script **eval**ed.

To run the setup script, a parameter list was retrieved from the user's `.bnrrc` file and appended to the command line used to invoke the script. The setup scripts did everything from validating the platform they were running on, to starting daemons; they ranged in size from 28 to 304 executable-lines in length.

[1] The setup scripts were called *setup macros* when the standard login was originally written.

The `.bnrrc` Name Space

As part of the background to understanding both the standard login and the problems associated with it, it is necessary to discuss the the `.bnrrc` name space.

As originally conceived, the `.bnrrc` file was simply a place to put options which would be passed to the setup scripts as command line parameters. These `.bnrrc` file entries had the following format:

<SetupScriptName> *<VariableName>*=*<String>*

The standard login developers' initial intent was to provide a facility for users to set variables used by the setup scripts. Thus, allowing the user to customize a setup script's output. However, as the reader has probably noticed, these entries bear a striking resemblance to UNIX *environ* (7) entries.

About six months after the standard login had been released, it occurred to the standard login developers that it might be useful to have a C function which would emulate the **getenv(3)** UNIX library entry; that is, an **ITgetenv()** function which would be used by applications to retrieve their `.bnrrc` entries directly. So, a library was created; providing functions to add, delete, and retrieve entries from the `.bnrrc` file. Thus, the `.bnrrc` became a name space available to any application using the **ITgetenv()** library.

The function prototype for the **ITgetenv()** function is:

```
char *ITgetenv(const char *VariableName,
               const char *ProductTag);
```

The **ITgetenv()** function works as follows:

1. If the requested *<VariableName>* is set in the process environment space, return that value;
2. Then check the `.bnrrc` file for the requested *<ProductTag>*-*<VariableName>* pair, return that value if found;
3. If no variable was located, return **(void *)NULL**.

The Problem

As more application providers began to utilize the standard login to configure their applications, users began to experience a deterioration in the length of time it took to login. In some extreme cases, login times of over 10 minutes were being reported.

A secondary problem was experienced by the application providers: few of them knew how to write robust shell scripts. As a result, a non-trivial amount of their project time was spent writing, testing and maintaining their piece of the standard login. One common programming error by neophyte shell programmers involved the generation of error messages: a setup script would encounter an error and so output a message; which would then be passed along to the **eval** as though it were a command – causing the standard login to terminate.

These two problems, excessive execution time and non-trivial development, provided a business case for spending a substantial amount of time either re-writing or eliminating the standard login.

Specific User Concerns

While the aforementioned problems were the most serious, they were by no means the only ones. The issues uncovered during our initial investigation of the standard login included, in order of assigned priority:

1. too slow to execute
2. too easily broken (i.e. not robust enough)
3. not enough user documentation
4. **PATH** environment variable becoming excessively long

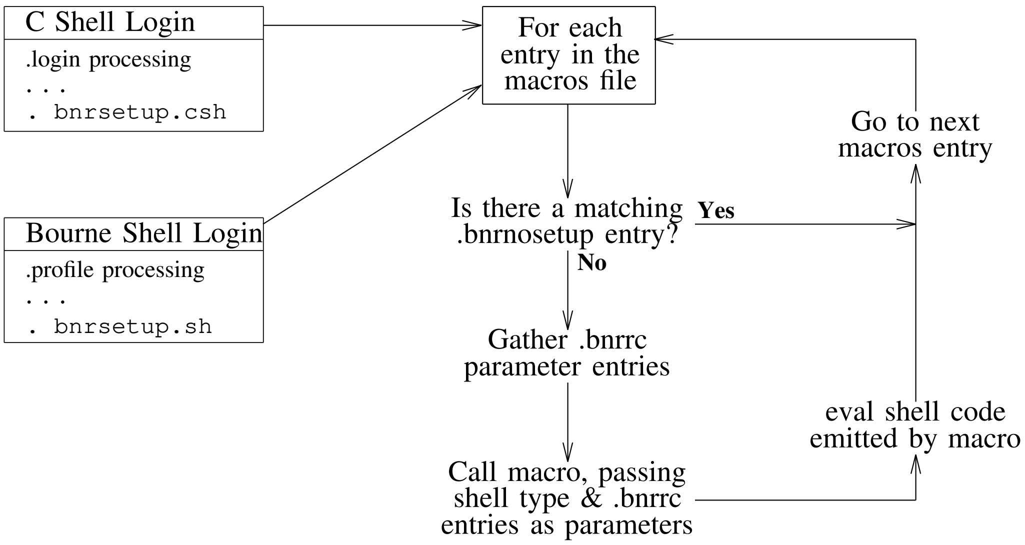

Figure 1: The Standard Login version 1.x process flow

5. no supported location for users' local environmental changes

Execution Time

As mentioned above, users on older, and/or heavily loaded hardware, were reporting login times in excess of 10 minutes. With HP VUE becoming the recommended window manager, and as users began to avail themselves of VUE's session save and restore functionality, login times were going to become even longer – even if the standard login portion remained static.

Some timing analysis was performed on the standard login. The results of that timing is shown in Table 1: *Standard Login Version 1.0 Timings.*

As can be seen from Table 1, the standard login spent time performing three basic functions: running the setup scripts, **eval**ing the output from the scripts, and setting up the user's X11 desktop.

The most significant amount of time was spent configuring the user's X11 desktop. However, we are unable to change the amount of time X11 will take to start the user's X-clients. In addition, the user can further degrade this portion of the login by adding to their X11 (VUE or Motif) configuration.

The other significant step was the execution of the `bnrsetup.sh` script and the setup scripts themselves. The user had no direct control over the execution time of these scripts. The 20 seconds spent running the scripts (in this test) was not a large amount of time; but, some of the setup scripts which were not tested took significant amounts of time on their own. One of these, `pde.setup`, is listed in the table.

The `bnrsetup.sh` script itself was a high consumer of time. However, we decided that it wouldn't be sufficient to only optimize that one script. It would be necessary to enhance the execution of the setup scripts themselves too.

It should also be noted that the time to **eval** the setup scripts' output was also not something we could significantly change. However, we could change the time it would take to generate the command stream being sent to the **eval**. We could also attempt to optimize the command stream itself.

Not Robust Enough

For a variety of reasons, few of them the fault of the setup scripts' authors, the standard login was very fragile. If an error occurred in either a setup script, or in a user's local configuration files, it would often cause the entire login process to halt prematurely. This meant that the user was either not able to log in, or was left with a partially configured environment.

The user-induced problems we saw included:

- *Some `.bnrrc` file Syntax errors would cause the standard login to fail;* one example of this was due to a quirk in the standard login's implementation. That quirk allowed C Shell users to use C Shell specific **path** syntax in the `.bnrrc` **PATH** entries. However, as soon as that entry was used by some login shell other than the C Shell (usually because of people copying each others `.bnrrc` files) the standard login would fail.
- *User `.bnrrc` **PATH** entries were easily broken;* users making changes to their **PATH**

Elapsed Time†	Portion of Time	Processes Executed
78s	100%	Total time‡ to login.
20s	26%	Time to source the `bnrsetup.sh` script, including execution of the `os`, `wga`, `ingres`, `pub`, `frame`, `probs` and `X11` setup scripts.
14s	18%	Time to source only the `bnrsetup.sh` script, without executing any of the setup scripts themselves.
3s	4%	Time to **eval** the code output from the setup scripts.
55s	71%	Time to configure the X11 desktop (with mwm, XClock, three XTerms, XBiff, and Emacs) and any other login tasks.§
~120s	–	Time to run the `pde.setup` script∗ (a script not included in our timing tests).

† Unless otherwise noted, all times were generated using the **time** and **date** UNIX commands on an HP360 workstation. Each value is the average of 4 consecutive measurements.

‡ Measured with a stopwatch.

§ This number was obtained by subtracting the setup script execution time (20s) and **eval** time (3s) from the total login time (78s).

∗ This time was reported to us by a user as an average time of execution.

Table 1: Standard Login Version 1.0 Timings

variable, via their `.bnrrc` file, often ended up truncating their **PATH**.

No User Documentation

Very little documentation existed to help users customize their local environments. Some **man** pages were eventually released, but during the course of my interviews I did not encounter a single individual who had seen them. The **man** pages were stereotypically UNIX too: they contained both user and developer information, presented in a minimalist fashion. While this was perfect for the experienced UNIX user, new-comers were lost.[2] Clearly, entry level documentation, maybe in the form of a reference card, was required.

As well, the default `.bnrrc` file installed in a user's home directory at account creation time didn't contain any useful comments. This further frustrated users attempting to make entries in their `.bnrrc` files.

Another interesting documentation problem involved conflicting information. The **man** pages for some of BNR's internal tools told users to make entries in their `.cshrc` or `.profile` files; however, if the user ran the tool without first making the entries, then the tool would prompt the user for the information and write it into the `.bnrrc` file. This eventually resulted in users making conflicting entries in their UNIX configuration files.

Long PATH Variable

Almost every setup script added a component to the **PATH**. Many of the scripts added more than one new component. In addition, the method recommended to users for re-ordering their **PATH** caused a duplication of components. Thus, making the **PATH** longer than necessary.

User's Environmental Changes

No supported location, beyond entries in the `.bnrrc` file, was given to users for making changes to their environment. This was compounded by comments placed in the default login files (`.login`, etc.): comments warning users *not* to make any changes to the files. The problem with this is that only a very few environmental changes were possible via the `.bnrrc` file. The reality was that users sometimes needed to make changes to files other than the `.bnrrc` file; C Shell and Korn shell users especially. So, any attempt to make non-`.bnrrc` file changes unsupported was pointless.

Embedded comments to the contrary, no one was actually attempting to mandate that only `.bnrrc` file changes were supported. The comments had been placed in the default files to discourage unnecessary fiddling. In retrospect, the comments were too strongly worded.

Specific Developer Issues

While most problems directly impacted end users, some problems also affected software developers and support personnel. Those specific items identified during our initial inquiries were, in order of assigned priority:

1. insufficient documentation
2. lack of shell script writing skills in software development groups
3. default settings didn't provide enough feedback
4. the standard login scripts were not portable between versions of UNIX (HP-UX, SunOS, AIX, DomainOS)

Insufficient Documentation

Little documentation existed for setup script writers. The only real documentation consisted of the existing setup scripts themselves. This began to cause serious problems because there was no published programming interface (API). So, setup scripts began to use shell variables that had been intended as private `bnrsetup.[c]sh` variables.

Shell Programming Skills

There is a learning curve associated with every programming language. The UNIX shells are certainly no exception to this rule. However, because UNIX users use the shell languages at the UNIX prompt, new shell programmers typically believe they are already on top of the learning curve. However, experience shows that nothing could be further from the truth.

In the standard login setup scripts a variety of common mistakes were made time and time again. The mistakes were further compounded as new script writers cribbed code from existing scripts. The most common problems experienced were:

- *Reliance upon an unknown **PATH**;* for example, this resulted in commands such as **grep** running some program on the **PATH** other than `/bin/grep`.
- *Non-specific references to program names;* for example, C Shell users with an **rm** alias[3] would often find themselves being prompted about the removal of the old `.bnrsetup-log` file, when the `bnrsetup.csh` script attempted to remove it.
- *Error messages written to the **stdout** file handle;* as a result, error message often ended up in the **eval** command stream, causing the standard login to abort.
- *Indiscriminate use of unquoted variables;* when an unquoted variable was not defined, execution would sometimes fail because a

[2] It was interesting to observe that even when **man** pages were available, the experienced UNIX users went about discovering how the standard login worked by reading the source code.

[3] Where **rm** was aliased to '**rm -i**'.

parameter to a command was now missing.

- *Inconsistent use of **set** and **setenv**;* some of the setup scripts used **set** to change variables instead of **setenv**. This was not a problem when the **PATH** and **path** variables were being manipulated. However, one of the standard login's internal variables[4] was often mis-set in this fashion, sometimes resulting in a setup script's changes to that variable being lost.

Execution Feedback

The lack of user feedback during the execution of the standard login was of more concern to software support than to the script developers. However, it has been included here since this was a design decision that had been made by the standard login developers.

When users experienced difficulty logging into their systems, they called their local software support team. Invariably, since the standard login's default mode was to output no messages, the user was unable to give software support any information. This increased the time it took software support personnel to solve problems with the standard login; and subsequently reduced users' satisfaction with the standard login.

Script Portability

Stated simply, complex UNIX shell scripts are *not* portable; that is, a script written for HP-UX will invariably break when run under SunOS. While this has recently begun to change, it is still a real problem and it will be with us for a few years yet.

It is beyond the scope of this paper to discuss this is much detail. However, a couple of examples of non-portable behaviour are given here (to bring a little wisdom to the uninitiated):

- *Determination of platform type;* every vendor provides a different mechanism for identifying their hardware and OS release level to a program.
- *Many UNIX commands are implemented differently;* the list of commands that don't work identically on different platforms ranges from **awk** to **ps**, and **echo** to **grep**. While each of these commands performs the same basic functions on all platforms, the specific command line options available differ.

The non-portable nature of shell scripts meant either that different versions of the setup scripts existed on different platforms, or that each script had been constructed using multiple **if-then-elif-...-else** clauses to isolate platform dependent code. Both of these situations caused maintenance headaches.[5]

Possible Solutions

A number of solutions were examined as alternatives to the original standard login. Each of these possible alternatives was considered as a building block in the final solution; we believed that a number of small or medium sized changes was most likely to result in success. The list of alternatives included, in no particular order:

- re-writing the existing setup scripts in C (or some other compiled language)
- delaying the configuration of an application until its execution-time
- a table (configuration file) driven compiled program
- a make-environment program which would only be run when necessary
- optimizing the existing scripts
- remove the standard login and use the windowing system's configuration system

All of the options considered had one criterion in common: no changes were required to the user's personal configuration files in order to implement the option. We did not consider it generally helpful to make automatic and arbitrary changes to files in a user's home directory.

Re-write Scripts In C

This option was typically the first suggestion everyone made when we talked about the standard login's speed problems. However, it is very costly: every setup script would have to be re-written to gain any substantial benefit. While none of the scripts do anything very complex, the effort required would still be high. We did not want any of our solutions to make more work for the application developers, so this potential solution was not seriously considered.

Other High Level Languages

Also considered at this time was re-writing the setup scripts in Perl or TCL. These languages were rejected because they presented the same problem as shell scripts: the developer community we were supporting would have to learn a new, seldom used (for them), language. It must also be pointed out that the standard login, as it existed at the time, did not preclude the use of Perl or TCL scripts. However, none of the setup script providers chose any language other than **sh** or **csh**.

[4] Script writers often tried to change a variable named **PATH_START** using a **set path_start=** command. Amazingly enough, this sometimes produced a correct result due to an undocumented shell variable with this same name. However, it failed to work as soon as more than one script used this private variable name.

[5] An effort to overcome this problem was undertaken within BNR in 1991–92 (see [And92]). The core standard login scripts were re-written to run on all of the platforms then in use: Apollo HP, IBM, and Sun. This resulted in a doubling of the length of the scripts due to differences in the various vendor's UNIX systems.

One benefit this method had over the others was that it could be integrated into the system over time: developers didn't have to do the re-write by any specific date. This was because the setup scripts would all still be called as stand-alone programs and asked to emit shell code. Each setup script could be updated as the developers had time.

Delayed Configuration

It would be possible to delay the configuration penalty until execution time by placing the configuration script in a central directory. The script would configure a run-time environment and then call the actual application.

This method has the advantage that any changes to the environment take place within the scripts execution space, and not within the user's. Thus **PATH**s don't grow and grow, and environment variables only required by a specific application are hidden from all other applications.

ITgetenv Library

As mentioned above, a BNR provided library, called the *ITgetenv library,* was available to application developers. It provided transparent access to the `.bnrrc` file via a call modeled after **getenv(3)**. Applications could use the ITgetenv library calls to retrieve user's application defaults at execution time, rather than relying upon the standard login to do it for them.

Two problems existed here: there was no shell script access to the `.bnrrc` name space, and applications were not making judicious use of the ITgetenv library. However, both of these deficiencies were correctable.

Table-Driven C Program

Another option was a table-driven C program[6] which would produce shell code to be **eval**ed. This C program would replace the setup scripts with *setup tables* which would, potentially, be more maintainable (by application developers) than either the existing scripts or C programs would be.

This method would require all of the existing scripts to be re-written. However, since each script would be reduced to a simpler configuration file, less effort would be required than to re-write the scripts as C programs.

Make Environment Program

A make environment program could be written which would check the standard login configuration files and scripts, and generate a command stream and save it in a file if the time-stamp on the command stream was older than any of the standard login's files. The saved command stream would then be **eval**ed. If none of the standard login's files had changed since the command stream had last been generated, the command stream would simply be **eval**ed at future logins.

This method would have been the cleanest for the setup script writers, since all of the existing scripts could be left as is. However, we would still have been left with the problem of the setup scripts themselves: they were difficult to debug, were easily broken by changes in the environment, and required expertise in shell programming to be properly written.

In cases where no command stream needs to be generated, this solution would probably have run the fastest. However, problems arise:

- *How would different kinds of logins be handled?* Would multiple saved command streams be required; one for X11 logins, one for telnet, etc. While you could easily handle this by treating a change in login type as a change in the standard login's files, this would have meant slower logins for users who login using more than one method on a regular basis.
- *How would logins work when a home directory is NSF mounted on a different platform?* Do we have still more saved command streams, or do we simply keep re-generating one? See the previous point for further discussion.
- *What if a some application goes away?* How does the make-environment program detect that pieces are now missing? Does it matter that a piece is missing? The pre-processing required to decide whether to regenerate the **eval** stream is now going beyond the capability of the average **make** program.

Too many problems exist with this option to seriously attempt to implement it. The heavily networked BNR environment is quite dynamic. That is why the standard login was written in the first place. Writing a program to maintain a static piece of code doesn't appear to make sense.

More Lightweight Scripts

The setup scripts were spending a considerable amount of their execution time checking parameters, platform type and OS types. Some of the execution overhead could be eliminated by re-writing the scripts so that they assume more about the environment they run in; and common information could be inherited from the central `bnrsetup.sh` script. This would make them less robust, but that might not be a problem as they are programmatically invoked.

An example of this process duplication was the determination of CPU-type. Almost every one of the scripts spent time (from 15 to 40 lines) determining what kind of CPU they were executing on. It would be much more efficient for the `bnrsetup.sh`

[6]As will be seen later in this paper, the term *table-driven* turned out to be a misnomer.

script to determine this up front and then pass it along to the setup scripts themselves.

While this option would considerably enhance the execution of the smaller, shorter setup scripts. The longer, more problematic scripts would only see a small fraction of their execution time improved by this change.

Stop Using the Standard Login

One simple way of speeding up logins would be to stop using the standard login altogether. Some user's we spoke with did advocate this solution; especially those who had already stopped using the standard login.

We believed that it was possible to consider this request because HP VUE was about to be deployed. VUE did have facilities within it to configure the user environment. In addition, it offered a session save and restore facility that meant users only had to configure their environment once.

There were two serious difficulties with this solution: it was a vendor specific solution that we would have to find replacements for on non-HP platforms, and it didn't address the needs of non-X11 logins. The latter problem was most serious, since many non-X11 based UNIX sessions were initiated by users each day; **telnet** sessions between machines, dial-in from home, and batch (**cron**) jobs are common examples of these non-X11 logins.

The Final Solution

We approached the construction of our solution by asking two questions:

1. How can we make the login faster?
2. How can we provide extra functionality required by users and developers?

We chose these questions because they summarized the issues which had initially set us upon this trek. The questions ensured that we would not lose our focus as we went on with the project. At a more fundamental level that focus was *serving our customers,* since customer input is what had initially spurred development of the standard login.

Enhancing Login Speed

In order to speed up the standard login we proposed two things: replacing the script based system with a configuration file (table) driven C program, and working with developers to delay the configuration of some applications until execution-time.

Table Driven C Program

After examining the existing setup scripts, we determined that a small configuration language could be defined which would provide most of the functionality required by the setup script writers. It was also our belief that the functionality that might be missing from this small language would not have any significant impact on the developers of the existing setup scripts.

We designed a small finite choice grammar (see [McG80]) which provided commands to manipulate environment variables, determine a file's type, validate a terminal type, and execute arbitrary UNIX command-lines. In all, nineteen commands were created.

In deciding to re-write the standard login in this manner, we were making a decision to create our own small language instead of using one of the plethora of languages already in existence. The primary factors which influenced our decision were:

- tight integration with the `.bnrrc` name space
- fast parsing and execution of the setup tables

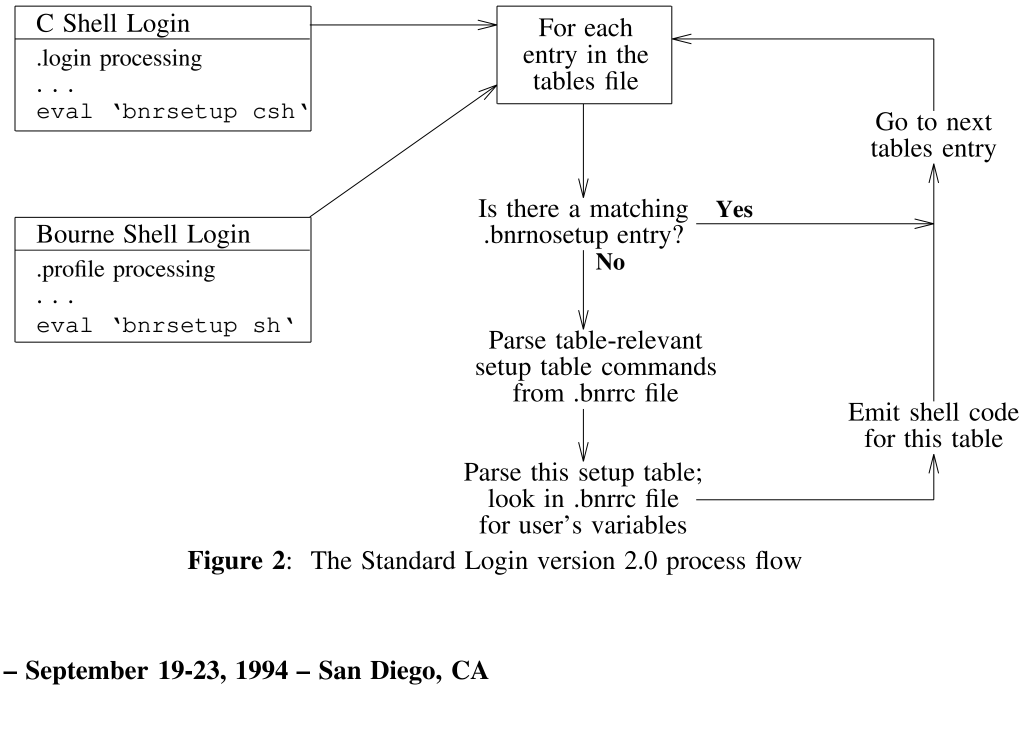

Figure 2: The Standard Login version 2.0 process flow

- to place some limits on what configuration could be done by the standard login
- unrestrained use of all source code and executables used in the standard login, since the standard login is distributed to our development partners around the world

Delayed Configuration

The intent of this proposal was twofold: to identify specific tools which would aid developers in delaying configuration, and then to convince developers to modify their applications to support delayed configuration. What this really meant was providing tools for, and convincing developers to use, the `.bnrrc` name space.

Our discussions with developers resulted in the specification of three programs: **rcget**, **rcset**, **rcunset**. These three programs would be used from shell scripts to add, remove and retrieve entries from/to the `.bnrrc` file. A copy of the **rcget**, **rcset**, **rcunset** UNIX **man** page has been included as an appendix to this paper.

The three programs were designed to be as customizable as possible and to return as much information as possible. As a result of our developer interviews, two application groups shipped early versions of the utilities and their applications in order to make better use of the `.bnrrc` name space before wide distribution of the re-written standard login.

Measured Performance

Timing of version 2.0 of the standard login was performed and is shown in Table 1: *Standard Login Version 2.0 Performance Gain.* It is interesting to note that although version 2.0 spent 30% longer **eval**ing the output code, there is actually 3 times as much code being generated by the standard login (measured in bytes). The particular set of setup tables used in this timing test now generates 3892 bytes (129 commands) of shell commands, compared to version 1.1's 1202 bytes (34 commands).

Developer Services

As we discussed above in subsection *Other High Level Languages,* our rejection of that option did place a time-constraint upon the script-providers. To minimize the impact of our re-write, we committed project resources to translate existing setup scripts into new setup tables for *any* script-provider requesting the service.

Over the life of the project, about 15 scripts were re-written on behalf of script-providers; only one of the re-writes took a full day, 3 of the scripts were re-written in under an hour, and the remaining scripts were completed in less than half a day. Once a script had been re-written, the script-provider assumed ownership of, and maintenance responsibility for, the table.

Standard Login v2.x Components

The standard login now consists of these files:

`bnrsetup` – the main program which performs all the processing. It is run, and its output is **eval**ed, from either the user's `.login` or `.profile` file. `bnrsetup` parses each setup table and emits shell code to a containing **eval** statement.

`tables` – a list of setup tables which *might* be resident on the system. Each entry in the file was the absolute path of a table.

`.bnrnosetup` – a list of the setup tables which the user does not want the standard login to process.

`.bnrsetuplog` – the output log created by the standard login. This file always exists following execution of `bnrsetup`.

Ver. 1.1 Elapsed Time†	**Ver. 2.0 Elapsed Time**	**Version 2.0 Performance**	**Processes Executed**
30.0s	3.0s	10 times faster	Time to run the **bnrsetup** program, including processing of the `os`, `wga`, `ingres`, `frame`, `probs` and `X11` setup scripts or tables.
8.5s	1.5s	5 times faster	Time to run only the **bnrsetup** program, without processing any of the setup scripts or tables themselves.
3.0s	4.5s	30% slower	Time to **eval** the code output from the setup scripts.
33.0s	7.5s	>4 times faster	Total time to **eval** the output from the **bnrsetup** program.‡

† All times were generated using the **time** and **date** UNIX commands on an HP360 workstation. Each value is the average of 4 consecutive measurements.

‡ This number was obtained by adding the **bnrsetup** execution time (30.0s and 3.0s, respectively) to the **eval** time (3.0s and 4.5s, respectively).

Table 2: Standard Login Version 2.0 Performance Gain

`.bnrrc` – a mini name space where applications store user specific application defaults. Users may also place their own setup table commands here; to be executed just prior to the processing of the corresponding setup table.

`os.setup.table` – the first setup table to be processed by the standard login; always the first entry in the `tables` file.

`X11.setup` – the last setup table to be processed; always the last entry in the `tables` file.

`wga.table`, `pub.setup.table`, ... – each of the other setup tables has a corresponding entry in the `tables` file. When processed, they cause shell code to be emitted to configure the user's environment.

Setup Table Commands

This section contains a description of the setup table commands we created. This list of table commands was generated by examining the setup scripts in use with version 1.x of the standard login. Commands were created to allow all of the functionality required for the setup script writers to re-write their scripts as setup tables.

It is worth noting that one of the commands was incorrectly constructed, **exitIfEqual**. This command should have been implemented as **exitIfNil**. As can be seen in the FrameMaker setup table below, the current version of the **exit** command leaves an environment variable set upon exiting from the table. As well, the logic required to use the existing **exit** command is reversed, adding an extra line to the setup table in order to detect the exit condition.

The commands are presented in logical rather than alphabetical order. Note that while setup table commands are not case-sensitive, the command parameters are. Each entry (a.k.a setup table command) in a setup table is terminated with a new line.

*<comment>* – Begin a comment[7] that continues until a newline is encountered. Note that if the newline is escaped with a backslash ('\') then the comment continues onto the next line (this is not a recommended practice).

set *<varName>* *<varValue>* – Set and export the *<varName>*[8] variable.[9] In the C Shell, any shell variable with the same name will be **unset** to prevent confusion when *<varName>* is used later in the setup table processing.

setIfEqual *<cmpName1>* *<cmpName2>* *<varName>* *<varValue>* – If the *<cmpName1>*[10] and *<cmpName2>* variables are equal, then assign the new specified value to the *<varName>* variable. See the **set** command for further details. Any of the variable names may refer to the same environment variable. Thus the command, '**setIfEqual test1 test1 test1 val**', is the same as saying, '**set test1 val**'.

setIfNil *<cmpName>* *<varName>* *<varValue>* – Set and export the *<varName>* environment variable only if the *<cmpName>* variable has not yet been set in the environment or the `$HOME/.bnrrc` file. See the **set** command for further details. Note that *<cmpName>* and *<varName>* may both refer to the same variable.

setLocal *<varName>* *<varValue>* – In the Bourne and Korn shells, set but do not export an environment variable called *<varName>*. In the C shell, set a shell variable called *<varName>*.

unset *<varName>* – Remove an environment variable that was previously set. In the C Shell any shell variable will be **unset** along with the like-named environment variable.

realize *<varName>* – Take the `$HOME/.bnrrc` file variable, *<varName>*, and move it into the setup table's environment. This function is provided to allow the variables found in the `$HOME/.bnrrc` file to be easily modified within the setup table. If *<varName>* is already defined in the environment, or if no such variable exists in the `$HOME/.bnrrc` file for this table, then no action will be taken by this command.

prefixPath *<varName>* *<varValue>* – Prefix *<varValue>* to the named path environment variable, *<varName>*. See the **setPath** command for further details.

appendPath *<varName>* *<varValue>* – Append *<varValue>* to the named path environment variable, *<varName>*. See the **setPath** command for further details.

setPath *<varName>* *<varValue>* – Assign *<varValue>* to the *<varName>* path variable. Before the new path variable is actually sent to the shell, the path contained in *<varName>* is scanned and any duplicate components are removed.

setPathIfNil *<cmpName>* *<varName>* *<varValue>* – Set the *<varName>* path environment variable only if the *<cmpName>* variable has not yet

[7] *<comment>* is any string terminated by a newline.

[8] *<varName>* is a UNIX environment variable to be assigned a value by the setup table command.

[9] *<varValue>* is a string to be assigned to a variable named elsewhere in the setup table command. One level of expansion is performed on any variables referenced within the *<varValue>* string. Note that C shell colon commands may not be used with variables. Also, empty strings must be doubly quoted to prevent shell syntax errors when the table is processed; for example, '("")'. When using variable names within path strings, always enclose the variable name in braces; for example, '**(/some/path:${HOME})**'.

[10] *<cmpName>, <cmpName1>* and *<cmpName2>* are variables whose values are to be used in a comparison by the setup table command.

been set in the environment or the `$HOME/.bnrrc` file. See the **setPath** command for more details.

chkFile *<filename>* *<varName>* – Determine the file type[11] and place it in the named environment variable. One of **directory**, **error**, **executable**, **nonexistent** and **regular** will be returned in *<varName>* by **chkFile**. This command is similar to the UNIX *test*(1) command. Note that any *<filename>* which is not a directory, an executable file, or a regular file, and which does not cause an error, will be identified as **nonexistent**.

[11] *<filename>* is any valid UNIX filename. One level of expansion is performed on any variables referenced within the *<filename>* string.

exitIfEqual *<cmpName1>* *<cmpName2>* – Discontinue processing of the table if the two environment variables have the same value. When this command is used in a `$HOME/.bnrrc` file, it only causes processing of the `$HOME/.bnrrc` file to stop for that table, not the processing of the entire setup table.

exec *<command-line>* – Run a program in the background.

exec-fg *<command-line>* – Run a program in the foreground.

execIfNil *<cmpName>* *<command-line>* – Examine the *<cmpName>* variable; if it has not yet been set in either the environment or the `$HOME/.bnrrc` file, then execute the command line in the background.

```
## BNR/NT Setup Table
#####
## /bnr/etc/frame.setup.table - Table to configure FrameMaker.
#####
# Unset the _NoPkg and _OK variables, since we're going to need to
# detect their presence, or lack thereof, later.
unset   _NoPkg
unset   _OK

###
# Set up to see if we're on a supported platform.
setIfEqual      CPU     _HP     _OK     (YES)
setIfEqual      CPU     _HPPA   _OK     (YES)
setIfEqual      CPU     _SUN4   _OK     (YES)
###
# If _OK didn't get set then we're not on a supported platform.  So,
# output a message to the user saying that we couldn't configure
# frame, and then exit.
execIfNil-fg    _OK     (echo 'FrameMaker is not supported on this platform.')
execIfNil-fg    _OK     (echo 'FrameMaker setup failed.')

setIfNil        _OK     _NoPkg  (YES)
exitIfEqual     _NoPkg  _YES

###
# We're on a supported platform, so do configuration.  Set the
# DIR_FRAME variable; always deferring to what the user may have
# entered.
realize         DIR_FRAME
setIfNil        DIR_FRAME       DIR_FRAME       (/bnr/3rdparty/frame)
###
# Add the DIR_FRAME directory to the path.
prefixPath      PATH_TAIL       (${DIR_FRAME}/bin)

###
 # Now clean up by getting rid of the DIR_FRAME environment variable;
# and the junk left over from the platform check.
unset   DIR_FRAME
unset   _OK
unset   _NoPkg
###
# End of the table.
```

Figure 3: FrameMaker Setup Table

execIfNil-fg *<cmpName>* *<command-line>* – Examine the *<cmpName>* variable; if it has not yet been set in either the environment or the `$HOME/.bnrrc` file, then execute the command line in the foreground.

execIfEqual *<cmpName1>* *<cmpName2>* *<command-line>* – Compare the two environment variables; if they are equal, then execute the command line in the background.

execIfEqual-fg *<cmpName1>* *<cmpName2>* *<command-line>* – Compare the two environment variables; if they are equal, then execute the command line[12] in the foreground.

configTerm – Examine the **TERM** variable and verify that the terminal type is found in the terminfo database. If it is not found then prompt the user for a value. If the user simply presses the `[return]` key at the prompt then use the value set by the `$HOME/.bnrrc` file's **DEF_TERM** variable. If no **DEF_TERM** variable exists, then assign a default, platform dependent, value.

If the user enters a terminal type that is not found in the terminfo database, re-prompt the user, warning of the error. The user may simply press `[return]` once again to force the invalid terminal type. **configTerm** will re-prompt two times beyond the initial prompt (at most) if invalid types are entered by the user.

The FrameMaker Setup Table

A setup table which configures a user's environment to run FrameMaker, as supplied with version 2.0 of the standard login, is shown in Figure 3. Although this is taken from the actual standard login release, it has been shortened slightly for inclusion in this paper.

Note the reverse logic preceding the **exitIfEqual** command, as discussed above.

Project Success Or Failure

At the outset of this project, we chose as our goal an execution time of less than 15 seconds for an average user. From the results of our own timing tests (shown in table 2), and from actual user reports, we can see that we have reached our 15 second goal. As well, setup scripts did prove to be as easy to re-write as we had expected.

Conclusions

When I began this project I had never used, nor even seen, the standard login. As an outsider, I viewed its existence with a skeptical eye. However, after speaking with many users, and a dozen-odd application development groups within the corporation, my skepticism has vanished.

The standard login is widely used within Bell-Northern Research, and with good reason:

- it provides developers with a reliable mechanism for disseminating changes to their application's environment
- it gives users a platform and shell independent method of customizing their desktop environment
- it provides one configuration interface for all of the core applications; especially important since third party (purchased) applications are often configured by an in-house developer using the standard login

These reasons assure the standard login's continued existence. As we begin to provide corporate-wide support for non-UNIX desktop computing, exactly what the standard login will look like five years from now is anyone's guess. However, we do know that any change will be upwardly compatible; and I believe, that compatibility will attract more users and developers to the standard login.

Software Availability

The standard login and its components are not available in any form.

Author Information

Christopher Rath has been working in the industry for a little over 10 years. His first exposure to UNIX was as a Xenix Sysadmin in 1985, and over the years he has done programming, analysis, and sysadmin work on a wide variety of OS's and platforms. He is presently employed by Bell-Northern Research, managing the group responsible for packaging and distributing the corporation's desktop UNIX environment. Christopher's mailing address and telephone number are *Bell-Northern Research Ltd., P.O. Box 3511, Station C, Ottawa, ON, Canada, K1Y 4Y7,* and *(613) 765-3141,* respectively. He may be reached via email at crath@bnr.ca .

References

[And92] Doug Anderson. *Development of The Standard Login Scripts.* Co-op Term Report submitted to manager at end of work term, April 1992.

[Arn92] Bob Arnold, ASK/Ingres Product Division. *If You've Seen One UNIX, You've Seen Them All.* In *LISA V Conference Proceedings.* USENIX Association, September 1991.

[jpl93] jpl@allegra.att.com. Finding Commands in a Multi-Vendor Universe. Unpublished email message to the author's manager, July 1993.

[McC92] Tom McConnell (tmcconne@sedona.intel.com). SUMMARY: POLL: What is in X/shell

[12] *<command-line>* is a valid UNIX, shell independent command line. Also, empty strings must be quoted to prevent shell syntax errors when the table is processed; for example, '(" ")'.

files? Unpublished USENET posting to comp.windows.x.apps., August 1992.

[McG80] Andrew D. McGettrick. *The Definition of Programming Languages,* volume 11 of *Cambridge Computer Science Texts.* Cambridge University Press, The Pitt Building, Trumpington St., Cambridge, England, CB2 1RP, first edition, 1980.

[Par93] John Parr. Conversations during the standard login re-write. Principle author of the original standard login system, 1993.

[Sko94] David F. Skoll (dfs@doe.carleton.ca). envv – manipulate environment variables in a shell-independent manner. UNIX man page and source code for Version 1.1 of the application, March 1994.