Title
ENVIRONMENTS AND SEARCH PATHS FOR THE SOFTWARE TOOLS

Author
Breckon, T.

Publication Date
1983-08-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

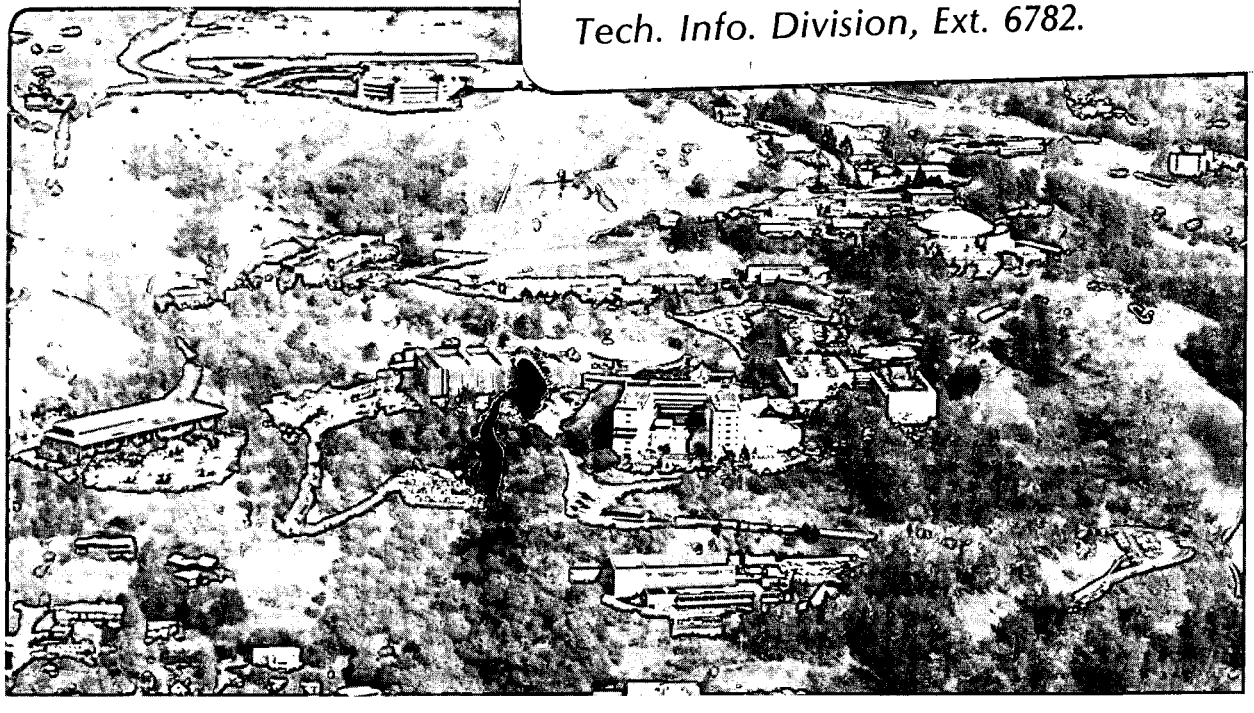
Engineering & Technical Services Division

Presented at the USENIX/Software Tools Joint Conference, Toronto, Canada, July 11-15, 1983

ENVIRONMENTS AND SEARCH PATHS FOR THE SOFTWARE TOOLS

T. Breckon

August 1983



Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

Environments and Search Paths for the Software Tools

Theresa Breckon

Real Time Systems Group
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

August 3, 1983

1. Why Environments?

There are many tools that need to know various bits of information concerning the structure of a computer system. Several tools need to know how files are organized on a system; compilers to expand include files, linkers to locate library object files, shells to locate current working directories, home directories, and executable programs, man tools to locate manual entries, mail tools to locate mail boxes and mailing lists. Visual editors need to locate information about terminal capabilities. Some system dependent primitives may need to rely on concepts such as user ID number, user group number, task number, etc.

Software that requires this information must know how to get this information and what the information will look like when it is retrieved. Generally, software that relies on a particular bit of information can decide on what it should look like, and then the code that extracts the information can massage it to meet this requirement. The code that extracts the information is usually a machine dependent system call. In the past, the Software Tools have provided system information by either specifying a machine dependent primitive which each implementor must supply, or by hard-wiring the information into the software.

These two methods are costly and inconvienent for implementors of the Software Tools. They must either spend time implementing the machine dependent primitive, or they must modify, debug, and test software which has this information hard-wired into it. Environments provide a method for portably describing these various bits of system structure information.

The set of routines to support environments is very small and need only be implemented once. This set need never be expanded to handle new bits of system structure information. New information is simply added to the Tools by specifying a new environment variable. This does not involve adding or modifying code.

In order to maximize on the efficiency and minimize on the cost of environments, the following criteria was followed in developing a model for the Software Tools:

(1) Keep the set of environment routines to a minimum. Try to avoid changes to existing software.

(2) Use existing Tools software whenever possible.

(3) Design a model that can be portably implemented, but don't restrict the design to a single implementation.

2. Environment Model

Using the above design criteria, the environment model described in this paper is based on the concept of a symbol table. In other words, an environment is simple a set of names (or symbols) and their values. This concept allows environments to be portably implemented using existing symbol table routines. The model consists of a very small set of routines because it is based on such a simple concept. This concept is general enough to not restrict environments to a single implementation.

Children inherit environments from their parents. Changes to environments within a child are local to the child's environment and do not effect the parent. Changes are made to an environment with a builtin shell command called *setenv*. *Setenv* is used to set or define environment variables. An environment variable has a

This work supported in part by the United States Department of Energy under Contract Number DE-AC03-76SF00098.

Paper presented at the USENIX/Software Tools Joint Conference, Toronto, Canada, July 11-15, 1983.

single value associated with it. There is no concept of a "legal" environment variable. A user can set any variable name desired. *Setenv* is specified as a builtin shell command so that a user can change the environment of the current shell. The synopsis for *setenv* is:

```
% setenv  name  [value]
```

Both name and value are single character strings, optionally enclosed in quotes. The name and value specified are stored in the environment symbol table. As a convienence to the user, if the value of an environment variable consists of multiple parts, the quotes that would normally enclose the string can be left off, i.e.

```
% setenv  PATH  ~/bin  /usr/bin  /etc/bin
```

Setenv will store the value for PATH as a single string consisting of 3 blank delimited parts. No other attempt is made to interpret the value string. This is left to the software which uses the variable. Interpreting special characters such as escapes at the *setenv* level would only place restrictions on the tools which use particular environment variables. This is analagous to the shell not interpreting command arguments. If *setenv* is called with a name and no value, the variable is set to have no value, i.e. it is removed from the symbol table.

Since *setenv* has been specified as a builtin shell command, code must be added to the shell. This requires two routines; *envset* and *envrm*. The added shell code looks like:

```
if ( command is "setenv" )
   {
   Get name, value from command line
   if no value
      sts = envrm( name )
   else
      sts = envset( name, value )
   }
```

The *envrm* function removes the environment variable from the symbol table. The *envset* function installs the name and value in the symbol table. The portable version of these two routines would call the existing symbol table routines.

The only other two pieces of code to be added to the set of environment software is a *printenv* tool to print out all set environment variables and a *envget* routine to retrieve an environment variable. The portable version of *printenv* would use the symbol table routine sctabl to extract all environment variables and their values from the symbol table. The synopsis for the *envget* routine is:

```
sts = envget( name, value )
```

sts is OK if the value for the specified variable name is retrieved successfully, ERR if the name is not defined. The value is returned as a single ascii string.

3. Search Paths

Since many of the tools in the standard and in extensions to the standard are based on UNIX† models, it stands to reason that we would look to UNIX to model our environment variables. A typical set of environment variables defined by UNIX users is:

PATH	*shell search path*
HOME	*user's login directory*
TERM	*terminal capabilities*

The environment variable PATH is the only search path variable used in UNIX. There are many other tools whose usefulness could be enhanced greatly by the ability to search for specified files using search paths. Search paths are simply lists of places to search for a specified file. For example, the shell search path is a list of directories to look for specified commands. Any tool which searches for a file needs to know where to look for the file. If search paths aren't used, then this knowledge must be hard-wired into the code. This presumes that all system file structures are constructed similiarly and all users want to look in the same places for every file. And, if one system keeps their files elsewhere, the user must either copy the file to the designated spot in order to use it, or must modify the code searching for the file. For example, include files can and do reside in different places on many systems. Compilers need to know where to search for a specified include file. On UNIX, the C compiler looks in two places; the user's current working directory and */usr/incl*. This implies that a user must keep all include files that do not reside in */usr/incl* in the working directory. The user must also keep all library files there in order for the linker to be able to find them. Unless a user keeps all files in one directory, all relevant files must be copied into the current working directory for each compilation. This is very chaotic if more than one person is involved in modifying and creating a large program with multiple include files and libraries.

An include search path variable and a library search path variable allow users to keep their files in seperate directories and access them in an orderly fashion. Typical include and library search path variables would have

† UNIX is a Trademark of Bell Laboratories.

the following values:

~/incl /project/incl /usr/incl
~/lib /project/lib /usr/lib

The compiler would use the include search path variable to search for specified include files. The search would begin in the user's own include directory, then go on to a project's include directory where all of the files for a large software project are stored, and then the standard set of include files for the system would be searched. Other tools which search for a file and could benefit from search paths are a man tool which must search for manual entries (especially if a system has different sets of manual entries pertaining to different projects), a mail tool which searches for mailing list files (so that users may keep their own set of mailing list files), and a roff tool which searches for desired macro files.

The mechanism used by RTSG for implementing search paths is a portable function called pathopen which has the exact same interface as the standard Tools open routine. Search path variable names are incorporated into file names by simple preceding the variable name with a special character. The name of the file to be searched for follows this pattern. The metacharacter used is '+'. For example, an include search path file name would look like:

+INCL/ratdef

The function pathopen would recognize the metacharacter, get the value for the specified search path environment variable INCL, and search each of the directories listed for the include file ratdef. Pathopen would then call open with the full pathname of the file. At sites like RTSG where most of our effort is spent in maintaining and upgrading large project programs, the open routine can be completely replaced by pathopen in order to keep modification procedures orderly and programmer-friendly. This replacement is very simple because of the twin calling conventions of pathopen and open.

4. Implementation Issues

So far the model of environments has been very high-level, with the single concept of retrieving and putting environment variables and their values into a designated symbol table. This was done to facilitate machine dependent implementation of the environment routines. For instance, on a UNIX system, the code for the *envget* routine would probably be replaced with a call to the UNIX getenv routine. Users of the Tools on a VAX/VMS system may want to implement environment variables using VMS symbols.

There is another point in the portable implementation where implementors may choose to supply their own code. This is the point at which environment variables are passed from parent to child. The code to initialize the environment symbol table is kept in a single routine called *envini*. A call is automatically made to *envini* the first time an environment variable is referenced in a process. This was done so that there would be no extra overhead for programs that choose not to access the environment at all. A global flag is used to tell whether the environment has been referenced before. The following code is placed at the beginning of the *envset* and *envget* routines:

```
if ( envflg == NO )
        {
        sts = envini( )
        envflg = YES
        }
```

Envini inherits the environment from the parent and stores the environment variables' names and values in the symbol table. *Envini* first gets a name, value pair from the inherited environment. It then stores this pair in the global environment symbol table.

The portable implementation of *envini* opens a predesignated environment file which contains an environment variable description on each line. The *envset* routine updates the environment file each time it installs a name, value pair in the symbol table.

One problem to beware of in this simple implementation concerns keeping the environment in a predesignated file. If a user is logged in twice and is setting environment variables during both login sessions, variable value conflicts could arise because a single environment file is being updated. A suggested solution to this problem is to have a re-constructable, temporary file name which is unique to each login session.

The *envini* routine can be implemented as seen fit. For instance, an implementor may choose to pass environment variables via the spawn argument facility. *Envini* would simply retrieve the environment variables in the same way that command arguments are retrieved, and then store them in the symbol table. This implementation is a bit more complicated then using files to store the environment, but it removes the problem of conflicting environments from two concurrent login sessions.

The last implementation issue is the lifetime of the environment. UNIX begins each login session with a clean environment and leaves it up to the user to set up the environment with calls to the command *setenv*. At sites which use a login shell and a login startup file, this is recommended. The *setenv* commands are placed in the

login startup file to be loaded by the login shell. Sites which don't have this setup may have to use the predesignated environment file scheme to store initial environment variable values. The first call to *envget* or *envset* will then read the environment from this file into the symbol table.

5. Conclusion

The Software Tools need environments and search paths. RTSG plans to submit the portable environment model described in this paper for distribution in the extensions section of the next Software Tools basic tape. After refining the model based on input from Tools users, we will submit the environment package (along with search path code for the shell), to the standards committee for final approval and distribution.

REFERENCES

1. K.Thompson and D.M.Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, May 1975. See printenv(1), csh(1), exec(2), getenv(3), path(5), environ(5).
2. B.W.Kernighan and P.J.Plauger, *Software Tools*, Addison-Wesley, 1976.
3. R.B.Upshaw and V.Jacobson, *Paths*, LBL internal document, Nov 1981.
4. *RTSG Software Tools Manual*, Lawrence Berkeley Labs, Nov 1981. See paths(0), sh(1), incl(1), man(1), mail(1), splb(2).
5. J.Kunze, *Why the C Shell Has Not Been Universally Accepted*, U.C.Berkeley.

Thanks to D.Scherrer and K.D.Poulton for their mail concerning environments, and to Bob Upshaw and Van Jacobson for their many ideas and suggestions concerning this paper.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720