subject: **Introduction to KSH-I (Issue 3)**
Charge Case 311531-0101
File Case 49059-6

date: **October 4, 2017**

from: **David G. Korn**
MH 59554
5D-112 x7975
(ulysses!dgk)

TM

ABSTRACT

Ksh-i is a command language (shell) for the UNIX* operating system. It is essentially compatible with the System V version of the Bourne shell[1] , has many additional features, such as those found in *Csh*[2], and executes faster than either of these shells. This memo introduces many of the additional features and explains some of the reasons for the better performance. This memo assumes that the reader is already familiar with the Bourne shell. The Appendix contains a sample script written in Ksh-i. The manual page for the current version is also included.

MEMORANDUM FOR FILE

1. Introduction

Over the past several years several papers have been written describing new command interpreters for the UNIX system. These papers can be divided into two categories: Those that improve the shell as a programming language, and those that improve the shell as a command interpreter. Most of the papers fall into the latter category. In particular, *vicmd*[3] preserves the friendly environment of *vi* (from which this memo was entered), and adds a facility for convenient command entry. An*emacs* oriented shell has also been written by Veach[4]. The*2dsh* [5] shell allows the setup of more complicated networks of processes than just pipelines. The never developed *See-shell*[6] proposes a Small-Talk like interface[7] suitable for bit-mapped terminals such as the BLIT[8] and the Apollo[9]. Perhaps the most widely used shell, other than the Bourne shell, is *Csh*, which runs under the Berkeley UNIX operating system. Csh has many attractive command interpreter features not currently in the Bourne shell; most notably, job control, history, arithmetic, and command name aliasing. On the other hand, many people (including this author), think that the Bourne shell is superior as a programming language. The history mechanism of Csh has recently been added as a local modification to the Bourne shell by J. L. Steffen[10].

The use of the shell as a programming language has been described by Dolotta and Mashey[11] and has been used by many people here at Bell Laboratories. Kolettis[12] presented extensions to the Bourne shell to provide message passing facilities and other inter-process communication and synchronization features. The Form shell[13] added form entry/edit capabilities to the Bourne shell. A proposal for a more programming language oriented shell has been proposed by Sturzenbecker[14].

This memo describes *Ksh-i*, aka Korn shell-international. This memo is not a tutorial, only an introduction. A description of the *Korn*shell can be found in Kochan and Wood[15]. Ksh-i is a direct descendant of the Form shell with most of the form entry/edit features removed and with many new features added. The primary focus of this work has been to provide an enhanced programming environment in addition to the

---

* UNIX is a trademark of Bell Laboratories.

major command entry features of *Csh*. Improved performance has been a major objective. Many of the additions have been provided so that medium sized programming tasks can be written at the shell level without a serious performance penalty. A concerted effort has been made to achieve System V Bourne shell compatibility so that scripts written for the Bourne shell can run without modification with Ksh-i. The description of features in this memo assumes that the reader is already familiar with the Bourne shell.

A version of Ksh-i has been run on several machines including but not limited to VAXEN , PDP-11's, IBM-370's, AT&T 3B's, UNIX-PC, PC-6300+, Suns, Alliant, CCI, Sequent, Pyramid, and Apollo Domain. It has been run on top of several versions of the UNIX operating system including Venix, Xenix, System III, System V, UTS, BSD 4.1, 4.2, 4.3, 8th. Edition, and DOMAIN/IX. The shell is in use in several centers at AT&T Bell Laboratories, and has been installed as */bin/sh* on VAXEN running System V, BSD 4.1., BSD 4.2, 3B's, PC-6300+, and UNIX-PC's running System V.

2. Shell Variables

The ability to define and use variables to store and retrieve values is an important feature in most programming languages. Ksh-i has variables with *identifier* names that follow the same rules as the Bourne shell. Since all variables have string representations, there is no need to specify the *type* of each variable in the shell. In Ksh-i, each variable can have one or more *attributes* that control the internal representation of the variable, the way the variable is printed, and its access or scope. Two of the attributes, *readonly* and *export*, are available in the Bourne shell. The **typeset** built-in command of Ksh-i assigns attributes to variables. The complete list of attributes, some of which are discussed here, appears in the manual page. The **unset** built-in of the Ksh-i removes values and attributes of parameters.

Whenever a value is assigned to a variable, the value is transformed according to the attributes of the variable. Changing the attribute of a variable can change its value. There are three attributes for field justification, as might be needed for formatting a report. For each of these attributes, a width can be defined explicitly or else it is defined the first time an assignment is made to the variable it Each assignment causes justification of the field, truncating if necessary. Assignment to fixed sized variables provides a simple way to generate a substring consisting of a fixed number of characters from the beginning or end of a string.

The attributes **−u** and **−l**, are used for upper case and lower case formatting respectively. Since it makes no sense to have both attributes on simultaneously, turning on either of these attributes turns the other off. The following script provides an example of the use of shell variables with attributes. This script reads a file of lines each consisting of five fields separated by **:** and prints fields 4 and 2 in upper case in columns 1-15, left justified, and columns 20-25 right-justified respectively.

```
typeset −L15u f4                    # 15 character left justified
typeset −R6u f2                     # 6 character right justified
IFS=:
set −f                              # skip file name generation
while   read −r f1 f2 f3 f4 f5      # read line, split into fields
do      print −r "$f4    $f2"       # print fields 4 and 2
done
```

The integer attribute, **−i**, causes the variable to be internally represented as an integer. The**i** can be followed by a number representing the numeric base for printing, otherwise the first assignment to an *integer* variable defines the output *base* (see below). This base will be used whenever the variable is printed. Assignment to*inte ger* typed variables result in arithmetic evaluation, as described below, of the right hand side.

Ksh-i allows one-dimensional *arrays* in addition to simple variables. Any variable can become an array by referring to it with a *subscript*. All elements of an array need not exist. Subscripts for arrays must evaluate to an integer between 0 and 511, otherwise an error results. Evaluation of subscripts is described in the next section. Attributes apply to the whole array.

Assignments to array variables can be made with parameter assignment statements or with the **typeset**

built-in. Referencing of subscripted variables requires the character **$**, but also requires braces around the array element name. The braces are needed to avoid conflicts with the file name generation mechanism. The form of any array element reference is:

${*name***[***subscript***]}**.

A subscript value of ***** or **@** can be used to generate all elements of an array, as they are used for expansion of positional parameters.

A few additional operations are available on shell variables. **${#***name***}** will be the length in bytes of **$***name*. For an array variable **${#***name***[*]}** gives the number of elements in the array.

There are four parameter substitution modifiers that have been added to strip off leading and trailing substrings during parameter substitution. The modifier **#**(**##**) strips off the smallest (largest) matching pattern from the left and the modifier **%**(**%%**) strips off the smallest (largest) matching pattern from the right. For example, if the shell variable **i** has value **file.c**, then the expression **${i%.c}.o** has value **file.o**.

3. Arithmetic Evaluation

The built-in command, **let**, provides the ability to do integer arithmetic. All arithmetic evaluations are performed using *long* arithmetic. Arithmetic constants are written as

*base***#***number*

where *base* is a decimal integer between two and thirty-six and *number* is any non-negative number. Anything after a decimal point is truncated. Base ten is used if no base is specified.

Arithmetic expressions are made from constants, variables, and one or more of the fourteen operators listed in the manual page. Operators are evaluated in order of precedence. Parentheses may be used for grouping. A variable does not have to have an integer attribute to be used within an arithmetic expression. The name of the variable is replaced by its value within an arithmetic expression. The statement

let x=x+1

can be used to increment a variable **x**. Note that there is no space before or after the operators**+** and **=**. This is because each argument to **let** is an expression to evaluate. The last expression determines the value returned by **let**. Let returns true if the last expression evaluates to a non-zero value. Otherwise,**let** returns false.

Many of the arithmetic operators have special meaning to the shell and must be quoted. Since this can be burdensome, an alternate form of arithmetic evaluation syntax has been provided. For any command that begins with **((**, all the characters until the matching **))** are treated as a quoted arithmetic expression. The double parentheses usually avoids incompatibility with the Bourne shell's use of parentheses for grouping a set of commands to be run in a sub-shell. Expressions inside double parentheses can contain blanks and special characters without quoting. More precisely,

((...))

is equivalent to

let " ... "

The following script prints the first *n* lines of its standard input onto its standard output, where *n* can be supplied as an optional argument whose default value is 20.

```
typeset −i n=${1-20}                                    # set n
while   read −r line && (( (n=n−1)>=0 ))                # at most n lines
do      print −r − "$line"
done
```

4. Functions and Command Aliasing

Two new mechanisms have been provided for creating pseudo-commands, i. e., things that look like commands, but do not always create a process. The first technique is called command name *aliasing*.

As a command is being read, the command name is checked against a list of *alias* names. If it is found, the

name is replaced by the text associated with the *alias* and then rescanned. The text of an alias is not checked for aliases so recursive definitions are not allowed. However, if the value of an alias ends in a space, then the word following the alias is also checked for alias substitution.

Aliases are defined with the **alias** built-in. The form of an**alias** command definition is:

alias *name=value*

The first character of an *alias* name can be any non-special printable character, while all remaining characters must be alpha-numeric. The replacement text, *value*, can contain any valid shell script, including meta-characters such as pipe symbols and i/o-redirection. Unlike **csh**, aliases in **Ksh-i** cannot take arguments. Aliases can be used to redefine built-in commands so that the alias

alias test=./test

can be used to look for *test* in your current working directory rather than using the built-in **test** command. Keywords such as **for** and **while** cannot be changed by aliasing. The command **alias**, without arguments, generates a list of aliases and corresponding texts. The**unalias** command remo ves the name and text of an alias.

Aliases are used to save typing and to improve readability of scripts. For example, the alias **alias integer='typeset −i'** allows integer the variables **i** and **j** to be declared and initialized with the command **integer i=0 j=1**.

Aliases can be used to bind program names to the full path-name of the program. This eliminates the path search but requires knowledge of where that program will be stored. *T racked* aliases make this use for aliasing automatic. A tracked alias is not given a value. Its value is defined at the first reference by a path-search as the full path-name equivalent of the name, and remains defined until the **PATH** variable is changed. Programs found in directories that do not begin with / that occur earlier in the path-search than the value of the tracked alias, take precedence over tracked aliases.

Tracked aliases provide an alternative to the *Csh* command hashing facility. Tracked aliases do not require time for initialization and allow for new commands to be introduced without the need for re-hashing. The **−h** option to the shell allows all command names that are valid alias names to become tracked aliases. This option is automatically turned on for non-interactive shells.

Functions are more general than aliases but also more costly. Functions definitions are of the form

```
function name
{
any shell script
}
```

The function is invoked by writing *name* and optionally following it with arguments. Positional parameters are saved before each function call and restored when completed. Functions are executed in the current shell environment and can share named variables with the calling program. Options, other than execution trace **−x**, set by the calling program are passed down to a function. The option flags are not shared with the function so that any options set within a function are restored when the function exits. All traps other than **EXIT** and **ERR** (described later) are also inherited. A trap on **EXIT** within a function will execute after the function completes but before the caller resumes. Therefore, any variable assignments and any options set as part of a trap action will be effective after the caller resumes. The **return** built-in can be used to cause the function to return to the statement following the point of invocation.

By default, variables are inherited by the function and shared by the calling program. However, environment substitutions preceding the function call apply only to the scope of the function call. Also, variables defined with the **typeset**, built-in command are local to the function that they are declared in. Thus, for the function defined

```
function name
{
   typeset −i x=10
```

```
    let z=x+y
    print $z
}
```

invoked as **y=13 name**, **x** and **y** are local variables with respect to the function **name** while **z** is global.

Alias and function names are never directly carried across separate invocations of Ksh-i, but can be passed down to sub-shells. Ordinarily, shell scripts invoked by name are executed in a sub-shell while scripts invoked as **ksh** *script* and shell escapes from other programs are carried out by a separate shell invocation. The **−x** flag is used with **alias** to carry aliases to sub-shells while the **−fx** flags of **typeset** are used to do the same for functions.

Each user can create a startup file for aliases and functions or any other commands. Aliases and functions that are to be available for all shell invocations should be put into this file. Aliases and functions which should apply to scripts, as well as interactive use, should be set with the **−x** flag. Setting this flag to redefine the semantics of a command can have undesired side effects. For example, **alias −x ls='ls −l'** will cause shell procedures which use the **ls** command within a pipeline to break. By setting and exporting the environment variable, **ENV**, to the name of this file, the aliases and functions will be defined each time Ksh-i is invoked. The value of the **ENV** variable undergoes parameter substitution prior to its use.

Several of the UNIX commands can be aliased to Ksh-i built-ins. Some of these are automatically set each time the shell is invoked. In addition, about twenty frequently used UNIX commands are set as tracked aliases.

The location of an alias command can be important since aliases are only processed when a command is read. A**.** procedure is read all at once (unlik e *profiles* which are read a command at a time) so that any aliases defined there will not effect any commands within this script.

A name is checked to see if it is a built-in command before checking to see if it is a function. To write a function to replace a built-in command you must define a function with a different name and alias the built-in name to this function. For example to write a **cd** function which changes the directory and prints out the directory name, you can write,

```
alias cd=_cd
function  _cd
{
   if     'cd'  "$@"
   then   echo  $PWD
   fi
}
```

The single quotes around **cd** within the function prevents alias substitution. The **PWD** variable is described below.

The combination of aliases and functions can be used to do things that can't be done with either of these separately. For example, the function and aliases defined as

```
function _from # i=start to finish [ by incr]
{
     typeset var=${1%%=*}
    integer incr=${5−1} $1
    while   (( $var <= $3 ))
    do      _repeat
          let $var=$var+incr
    done
}
alias repeat='function _repeat {'  from='}; _from'
```

allow you to write loops such as

```
repeat
    any script command
from i=1 to 13 by 3
```

with the expected behavior.

5. Input and Output

An extended I/O capability has been added to enhance the use of the shell as a programming language. The Bourne shell has a built-in **read** for reading lines from file descriptor 0, but does not have any internal output mechanism. As a result, the **echo(1)** command has been used to produce output for a shell procedure. This is inefficient and also restrictive. For example, there is no way to read in a line from a terminal and to *echo* the line exactly as is. In the Bourne shell, the **read** built-in cannot be used to read lines that end in **'\'**, and the **echo** command will treat certain sequences as control sequences. In addition, there is no way to have more than one file open at any time for reading.

Ksh-i has options on the **read** command to specify the file descriptor for the input. The **exec** built-in can be used to open, close, and duplicate file streams. The **−r** option allows a **'\'** at the end of an input line to be treated as a regular character rather than the line continuation character. The first argument of the **read** command can be followed by a **?** and a prompt to produce a prompt at the terminal before the read. If the input is not from a terminal device then the prompt is not issued.

The Ksh-i built-in, **print**, is used to output characters to the terminal or to a file. Again, it is possible to specify the file descriptor number as an option to the command. Ordinarily, the arguments to this command are processed the same as for **echo(1)**. However, the **−r** flag can be used to output the arguments without any special meaning. The **−n** flag can be used here to suppress the trailing new-line that is ordinarily appended.

To improve performance of existing shell programs, the **echo** command is built into Ksh-i. For the System V version of Ksh-i, the built-in **echo** is equivalent

print −,

where the − signifies that there are no more options permitted. On the Berkeley UNIX version the value of the **PATH** variable determines the behavior of the built-in **echo** command. If **echo** would resolve to **/bin/echo** with a path search, then **echo** is equivalent to

print −R.

The **−R** option allows only the **−n** flag to be recognized as the next argument. Otherwise,**echo** beha ves like the System V **echo** command.

The shell is frequently used as a programming language for interactive dialogues. The **select** statement has been added to the language to make it easier to present menu selection alternatives to the user and evaluate the reply. The list of alternatives is numbered and put in columns. A user settable prompt, **PS3**, is issued and if the answer is a number corresponding to one of the alternatives, the select loop variable is set to this value. In any case, the **REPLY** variable is used to store the user entered reply. The shell variables **LINES** and **COLUMNS** are used to control the layout of select lists.

6. Command Re-entry

An interactive shell saves the commands you type at a terminal in a file. If the variable **HISTFILE** is set to the name of a file to which the user has write access, then the commands are stored in this *history* file. Otherwise the file **$HOME/.sh_history** is checked for write access and if this fails an unnamed file is used to hold the history lines. This file may be truncated if this is a top level shell. The number of commands accessible to the user, is determined by the value of the **HISTSIZE** variable at the time the shell is invoked. The default value is 128. A command may consist of one or more lines since a compound command is considered one command. If the character **!** is placed within the *primary prompt* string, **PS1**, then it is replaced by the command number each time the prompt is given. Whenever the history file is named, all

shells which use this file share access to the same history.

A built-in command **fc** (fix command) is used to list and/or edit any of these saved commands. The command can always be specified with a range of one or more commands. The range can be specified by giving the command number, relative or absolute, or by giving the first character or characters of the command. The option **−l** is used to specify listing of pre vious commands. When given without specifying the range, the last 16 commands are listed, each preceded by the command number.

If the listing option is not selected, then the range of commands specified, or the last command if no range is given, is passed to an editor program before being re-executed by Ksh-i. The editor to be used may be specified with the option **−e** and following it with the editor name. If this option is not specified, the value of the shell variable **FCEDIT** is used as the name of the editor, providing that this variable has non-null value. If this variable is not set, or is null, and the **−e** option has not been selected, then **/bin/ed** is used. When editing has been complete, the edited text automatically becomes the input for Ksh-i. As this text is read by Ksh-i, it is echoed onto the terminal.

An editor name of − is used to bypass the editing and just re-execute the command. In this case only a single command can be specified as the range and an optional argument of the form *old=new* may be added which requests a simple string substitution prior to evaluation. A convenient alias,

alias r='fc −e −'

has been pre-defined so that the single key-stroke **r** can be used to re-execute the previous command and the key-stroke sequence, **r abc=def c** can be used to re-execute the last command that starts with the letter **c** with the first occurrence of the string **abc** replaced with the string **def**. Typing **r c > file** re-executes the most recent command starting with the letter **c**, with standard output redirected to *file*.

7. In-line editing

Lines typed from a terminal frequently need changes made before entering them. With the Bourne shell the only method to fix up commands is by backspacing or killing the whole line. Ksh-i offers options that allow the user to edit parts of the current command line before submitting the command. The in-line edit options make the command line into a single line screen edit window. When the command is longer than the width of the terminal, only a portion of the command is visible. Moving within the line automatically makes that portion visible. Editing can be performed on this window until the *return* key is pressed. The editing modes have commands that access the history file in which previous commands are saved. A user can copy any of the most recent **HISTSIZE** commands from this file into the input edit window. You can locate commands by searching or by position.

The in-line editing options do not use the *termcap* database. They work on most standard terminals. They only require that the backspace character moves the cursor left and the space character overwrites the current character on the screen and moves the cursor to the right.

There is a choice of editor options. The *emacs*, *gmacs*, or *vi* option is selected by turning on the corresponding option of the **set** command. If the value of the **EDITOR** or **VISUAL** ends any of these suffixes the corresponding options is turned on. A large subset of each of each of these editors' features are available within the shell. Additional functions, such as file name completion, have also been added.

The code for the *emacs* and *gmacs* editing option was supplied by Mike Veach. In the*emacs* or *gmacs* mode the user positions the cursor to the point needing correction and inserts, deletes, or replaces characters as needed. The only difference between these two modes is the meaning of the command **^T**. Control keys and escape sequences are used for cursor positioning and control functions. The available editing functions are listed in the manual page.

The code for the *vi* editing option was supplied by Pat Sullivan. The*vi* editing mode starts in insert mode and enters control mode when the user types **ESC**(**033**). The*r eturn* key, which submits the current command for processing, can be entered from either mode. The cursor can be anywhere on the line. A subset of commonly used *vi* commands are available. The**k** and **j** command that normally mo ve up and down by one *line*, move up and down one *command* in the history file, copying the command into the input edit window. For reasons of efficiency, the terminal is kept in canonical mode until an **ESC** is typed. On

some terminals, and on earlier versions of the UNIX operating system, this doesn't work correctly. The **viraw** option of the **set** command, which always uses **raw** or **cbreak** mode, must be used in this case.

Most of the code for the editing options does not rely on the Ksh-i code and be used in a stand-alone mode with most any command to add in-line edit capability. However, all versions of the in-line editors have some features that use some shell specific code. For example, **ESC-=** in all edit modes prints the names of files that match the current word and **ESC-*** adds the expanded list of matching files to the command line. A trailing ***** is added to the word if it doesn't contain any file pattern matching characters before the expansion.

8. Job Control

The job control mechanism is almost identical to the version found in *Csh* of the Berkeley UNIX operating system, version 4.1. The job control feature allows the user to stop and restart programs, and to move programs to and from the foreground and the background. It will only work on systems that provide support for these features. However, even systems without job control have a **monitor** option which when enabled will report the progress of background jobs and enable the user to **kill** jobs by job number or job name.

An interactive shell associates a *job* with each pipeline typed in from the terminal and assigns them a small integer number called the job number. If the job is run asynchronously, the job number is printed at the terminal. At any given time, only one job owns the terminal, i. e., keyboard signals are only sent to the processes in one job. When Ksh-i creates a foreground job, it gives it ownership of the terminal. If you are running a job and wish to stop it you hit the key **^Z** (control-Z) which sends a **STOP** signal to all processes in the current job. The shell receives notification that the processes have stopped and takes back control of the terminal.

There are commands to continue programs in the foreground and background. There are several ways to refer to jobs. The character **%** introduces a job name. You can refer to jobs by name or number as described in the manual page. The built-in command **bg** allows you to continue a job in the background, while the built-in command **fg** allows you to continue a job in the foreground even though you may have started it in the background.

A job being run in the background will stop if it tries to read from the terminal. It is also possible to stop background jobs that try to write on the terminal by setting the terminal options appropriately.

There is a built-in command **jobs** that lists the status of all running and stopped jobs. In addition, you are notified of the change of state of any background jobs just before each prompt. When you try to leave the shell while jobs are stopped or running, you will receive a message from Ksh-i. If you ignore this message and try to leave again, all stopped processes will be terminated.

A built-in version of **kill** makes it possible to use *job* numbers as targets for signals. Signals can be selected by number or name. The name of the signal is the name found in the *include* file **/usr/include/signal.h** with the prefix **SIG** removed. The**−l** flag of **kill** generates list of v alid signal numbers and names.

9. Security

There are several documented problems associated with the security of shell procedures[16]. These security holes occur primarily because a user can manipulate the *environment* to subvert the intent of a *setuid* shell procedure. Frequently, shell procedures are initiated from binary programs, without the author's awareness, by library routines which invoke shells to carry out their tasks. When the binary program is run *setuid* then the shell procedure runs with the permissions afforded to the owner of the binary file.

In the Bourne shell, the **IFS** parameter is used to split each word into separate command arguments. If a user knows that some *setuid* program will run **sh −c /bin/pwd** (or any other command in **/bin**) then the user sets and exports **IFS=/**. Instead of running**/bin/pwd** the shell will run **bin** with **pwd** as an ar gument. The user puts his or her own **bin** program into the current directory. This program can create a copy of the shell, make this shell *setuid*, and then run the **/bin/pwd** program so that the original program continues to run successfully. This kind of penetration is not possible with*Ksh-i* since the **IFS** parameter only splits

arguments that result from command or parameter substitution.

Some *setuid* programs run programs using *system()* without giving the full path name. If the user sets the **PATH** variable so that the desired command will be found in his or her local bin, then the same technique described above can be employed to compromise the security of the system. To close up this and other security holes, *Ksh-i* goes into a *protected* mode whenever the real and effective user or group id are not the same. In this mode, the**P ATH** variable is reset to a default value and the **.profile** and **ENV** files are not processed. Instead, the file**/etc/suid_pr ofile** is read and executed. This gives an administrator control over the environment to set the **PATH** variable or to log setuid shell invocations. Clearly security of the system is compromised if **/etc** or this file is publicly writable.

In BSD UNIX the operating system looks for the characters **#!** as the first two characters of an executable file. If these characters are found, then the next word on this line is taken as the interpreter to *exec* for this command and the interpreter is *exec*ed with the name of the script as argument zero and argument one. If the *setuid* or *setgid* bits are on for this file, then the interpreter is run with the effective uid and/or gid set accordingly. This scheme has two major drawbacks. First of all, using the**#!** notation forces an **exec** of the interpreter even when the call is invoked from the interpreter which it must exec. This is inefficient since the interpreter can handle a failed exec much faster than starting up again. More importantly, *setuid* and *setgid* procedures provide an easy target for intrusion. By linking a *setuid* or *setgid* procedure to a name beginning with a − the interpreter is fooled into thinking that is being invoked with a command line option rather than the name of a file. When the interpreter is the shell, the user gets a privileged interactive shell. There is code in*Ksh-i* to guard ag ainst this simple form of intrusion.

A more reliable way to handle *setuid* and *setgid* procedures is provided with *Ksh-i*. The technique does not require any changes to the operating system and provides better security. Another advantage to this method is that it also allows scripts which have execute premission but no read permission to run. Taking away read permission makes scripts more secure.

The method relies on a setuid **root** program to authenticate the request and exec the shell with the correct mode bits to carry out the task. This shell is invoked with the requested file already open for reading. A script which cannot be opened for reading or which has its suid and/or setgid bits turned on causes this setuid **root** program to get execed. For security reasons, this program is given the full pathname **/etc/suid_exec**. A description of the implementation of the**suid_exec** program can be found in a separate paper[17].

10. Miscellaneous

Ksh-i has several additional features to enhance functionality and performance. This section lists most of these features.

10.1 Tilde substitution

The character ~ at the beginning of a word has special meaning to Ksh-i. If the characters after the ~ up to a / match a user login name in the **/etc/passwd** file, then the ~ and the name are replaced by that user's login directory. If no match is found, the original word is unchanged. A ~ by itself, or in front of a /, is replaced by the value of the **HOME** parameter. A~ follo wed by a **+** or − is replaced by the value of the parameter **PWD** and **OLDPWD** respectively. Tilde substitution takes place when the script is read, not while it is executed.

10.2 Built-in I/O Redirection

All built-in commands can be redirected. Compound commands which are redirected are not carried out in a separate process.

10.3 Added options

Several options have been added to the shell and all options have names that can be used in place of flags for setting and resetting options. The command **set −o** will list the current option settings.

The option, **−f** or **noglob**, is used to disable file name generation.

The option **ignoreeof** can be used to prevent **^D** from exiting the shell and possibly logging you out. You must type **exit** to log out.

The **−h** or **trackall** option will cause all commands whose name is a valid alias name to become a *tracked* alias. This option is automatically turned on for non-interactive shells.

The job **monitor** option will cause a report to be printed before issuing the next prompt when each background job completes. It is automatically enabled for systems that have job control.

If the **bgnice** option is set, background jobs are run at a lower priority.

The option **markdirs** causes a trailing **/** to be appended on every directory name resulting from a pattern match.

The **protected** or **−p** options provides additional security by disabling the **ENV** from being executed and by resetting the **PATH** variable to the default value. Whenever a shell is run with the effective uid (gid) not equal to the real uid (gid) then this option is implicitly enabled. Instead of the **ENV** file, the file **/etc/suid_profile** is read so that administrators can have control over setuid scripts.

10.4 Built-in pwd

The **pwd** command is built-into Ksh-i and therefore much faster.

10.5 Logical naming

The **cd** command will take you where you expect to go even if you cross symbolic links. Thus, **cd ..** will move you up one level closer to the root even if your current directory is a symbolic link.

10.6 Previous Directory

Ksh-i remembers your last directory in the variable **OLDPWD**. The**cd** b uilt-in can be given with argument **−** to return to the previous directory and prints the name of the directory. Note that**cd −** done twice returns you to the starting directory, not the second previous directory. A directory*stac k* manager has been written as shell *functions* to *push* and *pop* directories from the stack.

10.7 Additional Variables and Parameters

Several new parameters have special meaning to Ksh-i. The variable **PWD** is used to hold the current working directory of the shell. The variable **OLDPWD** is used to hold the previous working directory of the shell.

The variable **FCEDIT** is used by the **fc** built-in described above. The variables **VISUAL** and **EDITOR** are used for determining the edit modes as described above.

The variable **ENV** is used to define the startup file for non-login Ksh-i invocations.

The variables **HISTSIZE** and **HISTFILE** control the size and location of the file containing commands entered at a terminal.

The parameter **MAILPATH** is a colon (**:**) separated list of file names to be checked for changes periodically. The user is notified before the next prompt. Each of the names in this list can be followed by a **?** and a prompt to be given when a change has been detected in the file. The prompt will be evaluated for parameter substitution. The parameter **$_** within a mail message will evaluate to the name of the file that has changed. The parameter **MAILCHECK** is used to specify the minimal interval in seconds before new mail is checked for.

The variable **RANDOM** produces a random number each time it is referenced. Assignment to this variable sets the seed for the random number generator.

The variable **SECONDS** is incremented every second. In a roundabout way, this variable can be used to generate a time stamp into the **PS1** prompt. The following code explains how you can do this on System V. On BSD you need another command to initialize the **SECONDS** variable.

 # If you . this script then you can use $TIME as part of your PS1 string to get

```
#   the time of day in your prompt
typeset −RZ2  _x1 _x2 _x3
let SECONDS=$(date  '+3600*%H+60*%M+%S')
_s='(_x1=(SECONDS/3600)%24)==(_x2=(SECONDS/60)%60)==(_x3=SECONDS%60)'
TIME='"${_d[_s]}$_x1:$_x2:$_x3"'
# PS1=${TIME}whatever
```

The parameter **PPID** is used to generate the process id of the process which invoked this shell.

The value of the parameter **_** is the last argument of the previous foreground command. Before execing each command this parameter is set to the file name of the command and placed in the environment.

The parameter **TMOUT** can be set to be the number of seconds that the shell will wait for input before terminating. A 60 second warning message is printed before terminating.

The **COLUMNS** variable can be used to adjust the width of the edit window for the in-line edit modes. It is also used by the **select** command to present menu choices.

The **LINES** variable controls how many rows a select list will take up on the screen. Select lists will try to occupy no more then two-thirds of **LINES** lines on the screen.

10.8 Modified variables

The input field separator parameter, **IFS**, is only used to split words that have undergone parameter or command substitution. In addition, adjacent non-blank delimiters separate null fields in Ksh-i.

The **PS1** parameter is evaluated for parameter substitution and a **!** is replaced by the current command number.

10.9 Timing Commands

A keyword **time** has been added to replace the **time** command. Any function, command or pipeline can be preceded by this keyword to obtain information about the elapsed, user and system times. Since I/O redirection bind to the command, not to **time**, parenthesis should be used to redirect the timing information which is normally printed on file descriptor 2.

10.10 Co-process

Ksh-i can spawn a *co-process* by adding a **|&** after a command. This process will be run with its standard input and its standard output connected to the shell. The built-in command **print** with the **−p** option will write into the standard input of this process and the built-in command **read** with the **−p** option will read from the output of this process. Only one such process can exist at any time.

10.11 Process Substitution

This feature is only available on versions of the UNIX operating system which support the **/dev/fd** directory for naming open files. Each command argument of the form **(***list***)**, **<(***list***)**, or **>(***list***)** will run process *list* asynchronously connected to some file in the **/dev/fd** directory. The name of this file will become the argument to the command. If the form with > is selected then writing on this file will provide input for *list*. If < is used or omitted, then the file passed as an argument will contain the output of the *list* process. For example,

paste (cut −f1*file1***) (cut −f3***file2***) | tee >(***pr ocess1***) >(***process2***)**

cuts fields 1 and 3 from the files *file1* and *file2* respectively, *pastes* the results together, and sends it to the processes *process1* and *process2*, as well as putting it onto the standard output. Note that the file which is passed as an argument to the command is a UNIX *pipe*(2) so that the programs that expect to *lseek*(2) on the file will not work.

10.12 Command Substitution

Command substitution (``) in the Bourne shell suffers from some complicated quoting rules. It is hard to write a **sed** pattern which contains back slashes within command substitution. Putting the pattern is single quotes doesn't help much. Ksh-i leaves the Bourne shell command substitution alone and adds a newer and easier to use command substitution syntax. All the characters between a **$(** and a matching **)** are evaluated as a command the output is substituted just as with ``. The**$** means *value of* and the **()** denotes a command. The command itself can contain quoted strings even if the substitution occurs within double quotes. Nesting is legal. You can use unbalanced parenthesis within the command providing that they are quoted.

The special command substitution of the form **$(cat file)** can be replaced by **$(< file)**, which is faster because no separate process is created.

10.13 Whence

The addition of *aliases*, *functions*, and more built-ins has made it substantially more difficult to know what a given command word really means. A built-in command, **whence** when used with the **−v** option has been provided to answer this question. A line is printed for each argument to **whence** telling what would happen if this argument were used as a command name. It reports on keywords, aliases, built-ins, and functions. If the command is none of the above, it follows the path search rules and prints the full path-name, if any, otherwise it prints an error message.

10.14 Additional test operators

The binary operators **−ot** and **−nt** can be used to compare the modification times of two files to see which is file is *older than* or *newer than* the other. The binary operator**−ef** is used to see if tw o files have the same device and i-node number, i. e., a link to the same file.

The unary operator **−L** returns true for a symbolic link.

10.15 Added Trap

All traps can be given by name in Ksh-i. The names of traps corresponding to signals are the same as the signal name with the **SIG** prefix removed. The trap**0** is named **EXIT** and a ne w trap named **ERR** has been added. This trap is invoked whenever the shell would exit if the **−e** flag were set. This trap is used by Fourth Generation Make[18] which runs Ksh-i as a co-process.

10.16 Shell Accounting

There is a compile time option to the shell to generate an accounting message for each shell script. The changes needed to provide this feature were supplied by Foregger[19] and have been adopted as described in his memo.

10.17 Coded in Standard C

Early versions of Bourne shell were coded in an ALGOL-68 like dialect of C. Ksh-i is coded in standard C. It tries to adapt itself to the environment when it is compiled taking advantages of the features of the host environment when possible. There are far fewer *lint* messages from Ksh-i then for the Bourne shell. Ksh-i does not catch the segmentation violation signal, SIGSEGV, so that it can run on machines that can't recover from these traps.

10.18 Internationization

Ksh-i treats eight bit characters transparently without stripping off the leading bit. There is also a compile time switch to enable handling multi-byte and multi-width characters sets.

10.19 No special meaning for ^

The Bourne shell uses ^ as an archaic synonym for |. The^ is not a special character to Ksh-i.

10.20 Added conveniences

You can refer to multi-digit positional parameters in Ksh-i by putting the number in braces. Thus, **${12}** is legal in Ksh-i but illegal in the Bourne shell.

Ksh-i will perform file name expansion of file name arguments if the expansion is unique. Thus, **cat < file*** will expand the file name if the expansion is unique.

If you invoke the shell as **ksh script** then Ksh-i will do a path search on script.

Unbalanced quotes will cause the shell to print an error message giving the type of quote and the line number on which the opening quote occurs.

Run time error messages detected by the shell will print the line number within a function or script where the error was detected.

11. Example

An example of a Ksh-i script is included in the Appendix. This one page program is a variant of the UNIX *grep(1)* program. Pattern matching for this version of *grep* means shell patterns consisting of **?**, *****, and **[]**.

The first half examines option flags. Note that all options except **−b** have been implemented. The second half goes through each line of each file to look for a pattern match.

This program is not intended to serve as a replacement for **grep**; just as an illustration of the programming power of Ksh-i. Note that no auxiliary processes are spawned by this script. It was written and debugged in under two hours. While performance is acceptable for small programs, this program runs at only one tenth the speed of **grep** for large files.

12. Performance

Ksh-i executes many scripts faster than the System V Bourne shell. One major reason is that many of the functions provided by *echo(1)* and *expr(1)* are built-in. The time to execute a built-in function is one or two orders of magnitude faster than performing a fork and execute of the shell. Command substitution of built-ins is performed without creating another process, and often without even creating a temporary file.

Another reason for improved performance is that all I/O is buffered. Output buffers are flushed only when required. Several of the internal algorithms have been changed so that the number of subroutine calls has been substantially reduced. Ksh-i uses hash tables for variables. Scripts that rely heavily on referencing variables execute faster. More processing is performed while reading the script so that execution time is saved while running loops.

Scripts that do little internal processing and create many processes may run a little slower on System V because the time to *fork* Ksh-i is slightly slower than for the Bourne shell. On BSD Unix, Ksh-i can be compiled with a **VFORK** option which uses *vfork* whenever possible. In this case, binary programs startup somewhat faster but shell script files start a little slower since a separate invocation of the Ksh-i in required.

The **ENV** file can have an undiserable effect on performance. Even if this file is small, the shell must perform an open of this file. If large functions are placed in the **ENV** file they must be read in and compiled even if they are never referenced. If you only need the startup file for interactive shells only, then set your **ENV** variable to a value which evaluates to a file name for interactive shells and to the null string otherwise. If you export the startup file name in the variable **START**, then setting

ENV='${START[(_$−=1)+(_=0)−(_$−!=_${−%%*i*})]}'

will only invoke the startup file for interactive shells since the subscript evaluates to 0 only if the shell is interactive.

If you need a startup **ENV** file for all shells then use a *case* statement on the **$−** parameter to distinguish which actions only apply to interactive shells. The **ENV** file should look like

```
# options aliases and functions for all shell invocations
case   $- in
*i*)
     # options aliases and functions for interactive only
     ;;
esac
```

If there are functions which are only occasionally referenced, put them into a separate file **$HOME/functions** or any name you prefer and put aliases in the **ENV** file for each function name of the form

alias *function_name*=**'. $HOME/functions;***function_name***'**

In the beginning of the **$HOME/functions** file you must unalias each of the function names defined in the file. The first reference to any *function_name* in the function file causes the function file to get read in and the functions compiled.

13. Conclusion

Ksh-i has several thousand regular users. Ksh-i is a suitable replacement for the Bourne shell. It offers new features, better performance, and is essentially upward compatible with the Bourne shell.

MH-59554-DGK-dgk David G. Korn

Copy to
Members of Center 5954
Laboratory 4542 Supervision
J. W. Gross
N. J. Kolettis
J. L. Steffen
P. D. Sullivan
M. T. Veach

APPENDIX

```
#
#SHELL VERSION OF GREP
#
vflag= xflag= cflag= lflag= nflag=
set −f
while((1))# look for grep options
docase"$1" in
−v*)vflag=1;;
−x*)xflag=1;;
−c*)cflag=1;;
−l*)lflag=1;;
−n*)nflag=1;;
−b*)print ′b option not supported′;;
−e*)shift;expr="$1";;
−f*)shift;expr=$( < $1 );;
−*)print $0: ′unknown flag′;exit 2;;
*)iftest "$expr" = ′′
thenexpr="$1";shift
fi
test "$xflag" || expr="*${expr}*"
break;;
esac
shift# next argument
done
noprint=$vflag$cflag$lflag# don’t print if these flags set
integer n=0 c=0 tc=0 nargs=$## initialize counters
for i in "$@"# go through the files
doif((nargs<=1))
thenfname=′′
elsefname="$i":
fi
test "$i"  &&  exec 0< $i# open file if necessary
whileread −r line# read in a line
dolet n=n+1
case"$line" in
$expr)# line matches pattern
iftest "$noprint" = ""
thenprint −r "$fname${nflag:+$n:}$line"
fi
let c=c+1 ;;
*)# not a match
iftest "$vflag"
thenprint −r "$fname${nflag:+$n:}$line"
fi;;
esac
done
iftest "$lflag" && ((c))
thenprint − $i
fi
let tc=tc+c n=0 c=0
done
test "$cflag" && print $tc#  print count if cflag is set
let tc#  set the exit value
```

REFERENCES

2. S. R. Bourne, *An Introduction to the UNIX* Shell," BSTJ - Vol. 57, No. 6 part 2, pages 1947-1972.

2. W. Joy, *An Introduction to the C Shell*, University of California, Berkeley, 1980.

3. S. L. Arnold, *Vicmd a Visual Shell for Video Terminals*, TM-81-54533-12, 1981.

4. J. L. Steffen and M. T. Veach, *The Edit Shell - Connecting Screen Editing with the History List*, USENIX Association Toronto Proceedings, 1983.

5. M. J. Rochkind, *2dsh - An Experimental Shell for Connecting Processes With Multiple Data Streams*, TM-80-9323-3.

6. Wayne T. Wilner, *See-Shell: a Graphical User-Interface for UNIX Systems*, Bell Laboratories internal memorandum, 1982.

7. D. C. Smith, C. Irby, R. Kimball, and B. Verplank, *Designing the Star User Interface*, BYTE, April, 1982, pp. 242-282.

8. R. Pike, *The Blit Programmer's manual*, Bell Labs, 1982.

9. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumfp, *The Architecture of an Integrated Local Network*, IEEE Journal of Selected Areas in Communications, Local Area Networks Special Issue, November 1983.

10. J. L. Steffen, *An Input History for the Bourne Shell*, TM-82-55426-3, 1982.

11. T. A. Dolotta and J. R. Mashey, *Using the shell as a Primary Programming Tool*, Proc. 2nd. Int. Conf. on Software Engineering, 1976, pages 169-176.

12. N. J. Kolettis. *Extended Shell - A Potential Real Time Interpreter*, TM-77-4145-01, 1977.

13. D. G. Korn and D. A. Lambeth, *Form Shell*, TM-80-9224-3, 1980.

14. M. C. Sturzenbecker, *A New Command Language for UNIX and related systems*, TM-82-45192-3.

15. S. G. Kochan and P. H. Wood, "Unix Shell Programming," Hayden Book Company, 1985.

16. F. T. Grampp and R. H. Morris, *UNIX Operating System Security*, AT&T Bell Labs Tech. Journal, Vol. 63, No. 8, Part 2, pp.1649-1671, 1984.

17. D. G Korn *Parlez-vous Kanji?* TM-59554-860602-03, 1986.

18. G. S. Fowler, "The Fourth Generation Make," Proceedings of the Portland USENIX meeting, pp. 159-174, 1985.

19. T. H. Foregger, *Shell Accounting*, Case 40094-21, July 1982.