PTC MKS Toolkit

Using the MKS KornShell

PTC Inc.

PTC MKS Toolkit: Using the MKS KornShell

authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

Important Copyright, Trademark, Patent, and Licensing Information: See the About Box, or copyright notice, of your PTC software.

UNITED STATES GOVERNMENT RIGHTS

PTC software products and software documentation are “commercial items” as that term is defined at 48 C.F.R. 2.101. Pursuant to Federal Acquisition Regulation (FAR) 12.212 (a)-(b) (Computer Software) (MAY 2014) for civilian agencies or the Defense Federal Acquisition Regulation Supplement (DFARS) at 227.7202-1(a) (Policy) and 227.7202-3 (a) (Rights in commercial computer software or commercial computer software documentation) (FEB 2014) for the Department of Defense, PTC software products and software documentation are provided to the U.S. Government under the PTC commercial license agreement. Use, duplication or disclosure by the U.S. Government is subject solely to the terms and conditions set forth in the applicable PTC software license agreement.

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

PTC Inc.
12015 Lee Jackson Memorial Hwy,
Suite 150
Fairfax, Virginia 22033
Phone: +1 703 803-3343
Fax: +1 703 803-3344
E-mail: MKSToolkitInfo@ptc.com

10.3-1000

Technical Support

To request technical support, please contact us on the PTC eSupport Portal below. In your request please include your Service Contract Number (SCN), the name and version number of the product, your serial number, and the operating system and version/patch level that you are using. Contact PTC Technical Support at:

Technical Support: http://support.ptc.com/

When reporting problems, please provide a test case and test procedure, if possible. If you are following up on a previously reported problem, please include the case number in the subject line of your correspondence.

Finally, please give us your e-mail address and telephone number so that we may contact you.

Table of Contents

Introduction 1

What happens when you type a command into your computer? If you are not familiar with the way that computing systems work, you may only have a vague answer to this question: "The computer does the command."

The truth is that the commands you type in are read by a program called a *command interpreter*. On most systems, there is nothing special about the command interpreter: it is just another program. Of course, it is an important program because it's the one that lets you run every other program... but it is not unique. It is possible to write different command interpreters that let you type in commands in different formats.

Once you understand this, you can begin to consider what an ideal command interpreter would do for you. Certainly, the less typing you have to do, the better; so an ideal command interpreter would offer ways to reduce your typing load, especially on repetitive or frequently-used commands. An ideal command interpreter would also be adaptable—it could be customized to be particularly good at the jobs you personally do most often. It should be powerful enough to handle long and complex command lines, but simple enough that quick and easy operations really are quick and easy.

The MKS KornShell is just such a command interpreter. On UNIX systems, all command interpreters are called *shells*. The heart of these operating systems is called the *kernel*; think of a command interpreter as a shell that is wrapped around the central core. When you use a UNIX system, what you see is the shell; the kernel is hidden. MKS admired the capabilities of the KornShell and decided to make it available to users on other systems.

You will find that the MKS KornShell provides many capabilities that the standard Windows command interpreters just do not have; however, standard Windows systems differ from UNIX systems in many respects, and a small number of the original KornShell

capabilities cannot be implemented. In addition, you should be aware that there are occasional conflicts between Windows conventions and UNIX ones. These are discussed later.

This document introduces all of the basic concepts of the shell and gives examples of how to use the shell. It's a good starting point for anyone who is unfamiliar with sophisticated command interpreters. If you are already familiar with shells and command interpreters, you might want to turn directly to the **sh** reference page for a complete technical explanation of the shell.

A Different Shell 2

If you are a new user, you may become so confused by the differences between the Windows and KornShell command interpreters that you might want to give up. That would be a shame, because in the long run, you will probably appreciate these differences and be glad that they are there. Don't give up!

Most potential problems have already been dealt with in the "Getting Started" chapter of the *PTC MKS Toolkit Product Overview and Solutions Guide*; we'll reiterate these command interpretation differences here to alleviate any further misunderstanding. As we go along, if there are any other subtle differences that might cause confusion they will be noted in the particular context.

Slash vs. Backslash

Once you have started the MKS KornShell, you can type in most of the commands that you used before (with **`command.com`** or **`cmd.exe`**); however, there is one important difference to remember.

If you are used to standard Windows syntax, you've probably been using the backslash character (`\`) to separate parts of a path name, as in

```
dir1\dir2\file.ext
```

The MKS KornShell uses slashes instead, as in

```
dir1/dir2/file.ext
```

It is very important to remember to use slashes, because the backslash character has a special meaning to the shell. If you type a backslash by mistake, the shell will probably not understand your command. The rest of this tutorial uses slashes in all path names.

If a command absolutely requires backslashes in file names, the command line that you type into the shell must have *two* backslashes wherever you want to use a single backslash. For example, if you want to read `dir\file` into a word processor named **`wrdproc`**, you may have to type

```
wrdproc dir\\file
```

on the command line. The reason for this is explained in Chapter 4: "Basic Features of the MKS KornShell" on page 9.

Specifying Command Options

PTC MKS Toolkit commands follow the UNIX convention of using a hyphen to indicate a command option. For example, for a case-insensitive search for the string `foo` in all files in the local directory, you would enter

```
ls | grep -i foo
```

The Windows command interpreters **`command.com`** and **`cmd.exe`** normally use a slash (`/`) to indicate command options. Commands in the PTC MKS Toolkit only use the slash in path names.

Missing Commands

As you begin trying out various KornShell features, you are going to discover that many of your favorite PC commands such as **`dir`**, **`del`**, and **`type`** may not work when you are in the shell. These basic commands are used so often that the designers of **`command.com`** and **`cmd.exe`** used a trick to make them faster to run than normal commands. Usually, when you type a command, the command interpreter goes searching for a file that contains the program you want to run, and this search takes time. To avoid this search time, the command interpreter designers built the most basic commands directly into **`command.com`** and **`cmd.exe`**. When you ask to **`del`** a file, for example, **`command.com`** itself deletes the file, without trying to call another program.

The MKS KornShell does not behave this way. Command names like **`del`** and **`dir`** are not special to the shell; the shell goes searching for program files, the same way it searches for any normal command. In this case, however, there *are* no program files, so the operation fails.

There are several alternatives to alleviate this problem:

- The best idea is to make use of PTC MKS Toolkit commands instead of the standard Windows ones: use **`lc`** instead of **`dir`**, **`rm`** instead of **`del`**, **`more`** instead of **`type`**, and so on. The PTC MKS Toolkit commands are designed to be consistent with the shell and to make full use of the shell's abilities. They are also nicer to use than the standard utilities; for example, the output of **`lc`** is more readable than the output of **`dir`**.
- If you still want to use a standard Windows command, you can make a special call to the native command interpreter to run one of its built-in operations. For example,

  ```
  command.com /c del file
  ```

  calls **`command.com`** to run the single command

  ```
  del file
  ```

  You can run any normal **`command.com`** command by typing

  ```
  command.com /c
  ```

  followed by the command you want to run.

For more information, see "Aliases" on page 11.

- Finally, you can set up an *alias* inside the shell, so that when you type **`dir`**, or **`del`**, or **`type`**, the shell does whatever extra work is needed to perform the operation. This requires a little preparation before it works, but once you have done the set-up, you do not have to worry about it again.

  This is the approach taken by the standard `environ.ksh` file provided with PTC MKS Toolkit, which creates such aliases for the following commands:

  ```
  copy    del     dir     erase   move    path
  ren     ver     verify  vol
  ```

Starting the Shell 3

This chapter describes several ways to start up the shell. All of these assume that the MKS KornShell is properly installed on your system.

The Simple Way

The simplest way to start the shell is just to type

```
sh
```

and press ENTER. **sh** is the standard PTC MKS Toolkit name for the KornShell (on some UNIX systems, the standard name is **ksh**).

Once you have entered the **sh** command, the system usually displays

```
$
```

Chapter 7: "Customization" on page 43 shows you how to personalize your shell prompt. You can also customize your prompt from a dialog box using the **gset** command.

This is the shell's standard *prompt*. It means that the shell is ready for you to type in a command. The $ prompt is a simple way to know when you are talking to the shell command interpreter. Other command interpreters use different prompts. For example, the standard **command.com** usually uses

```
C:\>
```

Profile Files

As later chapters show, you will often want to define special names for use while working with the shell (for example, aliases that stand for specific commands). One simple way to do this is to create a *profile* file.

A profile file named `profile.ksh` is usually stored under your home directory. (On UNIX and POSIX-compliant systems, this file is named `.profile`.) This file can contain MKS KornShell commands to set up any options or definitions that you might want to use while working with the shell.

The command

```
sh -L
```

starts the shell and tells it to begin by reading and running your profile file. Note that the letter `L` must be in uppercase. The shell looks for `profile.ksh`, under the directory indicated by the environment variable *`HOME`* (your home directory). If the variable *`HOME`* is not set, the shell looks for the profile file under the current directory; therefore, make sure that you are set to the right directory when you issue the command.

Later sections discuss many kinds of commands that you might want to put into your profile file.

Entering the Shell at Login Time

For more information, see the **`autorun`** reference page in the online *PTC MKS Toolkit Utilities Reference*.

You can start a KornShell window automatically whenever you log in by using **`autorun.ksh`**. If you use the KornShell often, this option will save you having to launch the shell every time you start your computer. **`autorun.ksh`** manipulates the Windows Registry Database to run any program upon bootup or login.

Leaving the Shell

If you have started MKS KornShell by typing **`sh`** or using **`autorun.ksh`**, the shell prompts you to enter a command, runs the command, prompts for another command, and so on. You can keep on going as long as you want. If you want to stop using the shell, just type

```
exit
```

and press `ENTER`. You return to whatever command interpreter you were using before you started the shell. Typing **`exit`** terminates the shell window.

Basic Features of the MKS KornShell

4

This section examines some of the basic features of the MKS KornShell. Once you have a grasp on the following concepts, working efficiently with the KornShell comes much easier.

Wild Card Characters

If you have used other shells or command interpreters you are probably familiar with the concept of wild card characters. A wild card is a special character that can be used to save typing inside path names. For example, a question mark (`?`) inside a path name can stand for any other single character.

```
ls file.?
```

lists any and all files with names that consist of `file.` followed by a single character. This can mean `file.a`, `file.b`, `file.c`, and so on... whichever of the files currently exist.

You might already be familiar with the use of `?` as a wild card character, since it is also recognized by some standard Windows commands. There is, however, one important difference: with standard Windows commands, the question mark stands for any character *except the dot*; with the MKS KornShell and other PTC MKS Toolkit commands, it stands for any character including the dot. This means that `???` matches a name like `a.b` when you are using the MKS KornShell, but not when you are using **`command.com`** or **`cmd.exe`**.

The asterisk (`*`) is another wild card character that is recognized by both the shell and standard Windows software. It stands for any sequence of zero or more characters.

The MKS KornShell has other wild card patterns that cannot be used with standard Windows software. For example, square brackets containing one or more characters stand for any one of the contained characters:

```
[bch]at
```

matches `bat`, `cat`, or `hat`.

```
cp [abc]* a:
```

copies all files, under the current directory, with names that start with `a`, `b`, or `c`, followed by any other sequence of zero or more characters. In other words, it copies all files with names that start with `a`, `b`, or `c`.

You can also specify ranges of characters inside the square brackets, as in

```
[a-m]
```

This matches any character from `a` through `m`. Suppose, for example, that you want to copy the contents of the current directory to floppy disks, but there is too much to fit on one disk. You might say

```
cp [a-m]* a:
```

to copy all files with names beginning with the letters `a` through `m`, then issue the second command

```
cp [n-z]* a:
```

to copy the rest.

If the first character inside a bracket construction is an exclamation mark (`!`), the construction matches any character that ***is not*** inside the brackets. For example,

```
cp [!a-m]* a:
```

copies any file that ***does not*** begin with one of the letters in the range `a` through `m`.

Note PTC MKS Toolkit commands all recognize the bracket wild card constructions, whether you are using the MKS KornShell, **`command.com`**, or **`cmd.exe`**.

Other Special Characters

The wild card characters have special meanings to the shell. There are several other characters that have special meanings

```
|       &       ;       <       >       (       )
space   tab     newline $       `       \       "
'       #       ~       {       }
```

Later sections of this document discuss the special meanings of most of these characters.

If you type in a command that contains any of these characters, the shell often assumes that you are using the character in its special sense. If you do not want to use the special sense of the character, put a backslash (\) in front of the character. For example,

```
echo it\'s me
```

displays

```
it's me
```

If you just try

```
echo it's me
```

without the backslash, the shell interprets the ' with its special meaning. It displays > after you press ENTER and waits for input instead of returning the usual $. An apostrophe (') without a backslash is taken to be the start of a string and the shell assumes that the string keeps going until you type another apostrophe, even if that goes on for several lines.

The moral is that you must remember to put in backslashes in front of any special character, unless you know what its special meaning is and you want that meaning. Since backslash itself is a special character, you must type two of them whenever you want a single backslash.

Aliases

Earlier in this document, we discussed making it easier to use commands like **dir** and **del** under the shell. The answer is to set up an *alias* for any command that you intend to use frequently.

An alias is a personalized name that stands for all or part of a command. It is created by typing

```
alias name="string"
```

in response to the MKS KornShell's usual prompt for input. This is not a normal command; it is an instruction to the shell itself. It does not work properly if you try it with **command.com** or **cmd.exe**.

As an example of a simple alias, try entering

```
alias m="more"
```

(The **more** command displays the contents of a file on your screen.) From this point onward, whenever the shell sees a command that starts with **m** (followed by a space or tab), the **m** is replaced with `more`. Clearly, this saves you some typing; you can say

```
m file
```

to get the effect of

```
more file
```

You can also set and display aliases from a dialog box using the **gvar** command.

Aliases in profile.ksh

To set up aliases for basic **command.com** or **cmd.exe** commands, use the following syntax for each command

```
alias dir="command.com /c dir"
```

or

```
alias dir="cmd.exe /c dir"
```

After you create these aliases, you can use them like the original commands. If you define an alias as

```
alias del="command.com /c del"
```

when the shell sees a command like

```
del file
```

it is translated into

```
command.com /c del file
```

The best way to set up these aliases is to put the **alias** commands in a profile file (`profile.ksh`). When you start the shell with

```
sh -L
```

or when you log in with the sign-on procedure, the shell reads the aliases from the file and sets them up immediately. That way, you do not have to type them in every time you start using the shell.

You can redefine an alias. If you issue the command

```
alias name="string"
```

and *name* is already an alias, the shell forgets the old meaning and uses the new meaning from then on. This means, for example, that you can redefine an alias that was previously defined in your profile file.

When the MKS KornShell replaces an alias, it checks to see if the result is another alias. For example, you might say to yourself, "Why do I need to type **dir** for the **dir** command? I could just set up the alias **d**." You could, therefore, type

```
alias d="dir"
```

When you type something like

```
d mydir
```

the shell replaces **d** with **dir**, which the shell in turn replaces with:

```
"command /c dir"
```

The shell continues to check for and replace aliases until no aliases remain or the replacement would result in an infinite loop of alias expansion.

To display all currently defined and predefined aliases, just type

```
alias
```

at the command prompt.

Double Backslashes

Suppose you have set up an alias for **dir** and are using the MKS KornShell. Suppose also that you want to find out what is under the root directory on your current device. You start by typing

```
dir /
```

because with the MKS KornShell, you use slashes in path names. You find that this does not work; your alias for **dir** is calling **command.com** to do the work, and **command.com** uses backslashes. So you type

```
dir \
```

That doesn't work either; the KornShell prompts with > for another line. Remember that backslash is a special character to the shell. If you want to use the literal meaning of any special character, you have to put a backslash in front. In this case, that means typing two backslashes; therefore, you have to type

```
dir \\
```

to run the **dir** command you want.

This is a general rule when you are using the KornShell: you must type two backslashes wherever you need one, as in

```
dir \\dir1\\dir2\\dir3
```

Command History

The shell records each command that you enter, in a file named `sh_histo` under your *home* directory. This is called the *history file*. (On UNIX and POSIX-compliant systems the history file is called `.sh_history`.) If you enter the command

```
history
```

the shell displays the current contents of your history file. Notice that each command is numbered.

You can re-run any of the commands in your history file by typing **r**, followed by a space, followed by the number of the command you want to run. For example, suppose that you are a programmer and you type in a complicated command to compile part of a program. The program contains a syntax error, so you call a text editor to edit the source code and correct the problem. Now you want to run the same compilation command on the corrected program. You may save yourself a good deal of typing by using

```
history
```

to find out the number of the previous compilation command, then running the command with **r**. For example, if the command was number 14, you type

```
r 14
```

The shell displays the original command 14, then runs it. If you get another error, you can correct it, then compile again with another

```
r 14
```

This way, you can perform the operation many times, but you only have to type the original once.

If you type **r** followed by a space, followed by a string of characters (not beginning with a digit), the shell checks backwards through the history file and runs the most recent command that begins with the given string. For example, let's look at the compilation example. Suppose you are using the Visual Studio **cl** command to compile your program. Then

```
r cl
```

looks back through the history and runs the most recent **cl** command. With this approach, you do not even have to check on the number of the command you want to run. The shell displays the selected command before running it.

This *search backwards* feature of **r** can search for aliases as well as normal commands. It searches for the beginning of the command line as you typed it, not the way that the line looked after the alias was replaced.

If you type **r** without a number after it, the shell repeats the most recent command. Think of **r** as a "**r**edo" command.

Note The **history** and **r** commands are actually just predefined aliases for the **fc** command. See the online **fc** reference page for more details.

You can also display and edit your command history from a scrollable dialog by striking the ESC key then the g key.

Editing Commands from the History File

Suppose that you have a sequence of source files named file1.c, file2.c, file3.c, and so on that you want to compile with similar **cl** commands. This situation is a little different from the one discussed in the last section. You do not want to re-run the ***same*** command for each file; the command has the same form each time, but you have to put in a new file name each time.

You can still do this using the history file. The command

```
r old=new command
```

runs a previous *command* but replaces the first occurrence of the *old* string with the *new*. For example, suppose you compile `file1.c` with

```
cl options file1.c
```

Then the command

```
r file1=file2 cl
```

tells the shell to search back for the most recent **`cl`** command and to change `file1` to `file2`. The shell makes this change, then displays and runs the modified command.

```
r file2=file3 cl
```

performs the same kind of operation, changing `file2` in the previous command to `file3` and then going ahead with the compilation. This saves you the trouble of retyping all the options for the command.

A command of the form

```
r x=y 3
```

replaces *x* with *y* in command 3, and then runs the modified command. If you just say

```
r old=new
```

it replaces *old* with *new* in the most recent command and runs the modified command. Thus the minimum amount of typing in our previous example is:

```
cl options file1.c
r 1=2
r 2=3
r 3=4
...
```

If you are doing this sort of thing, remember that the **`r`** command only replaces the first occurrence of the *old* string. Take a moment to make sure that this is not going to get you into trouble. In the first **`r`** command, 1 is replaced with 2 where it first appears in the command line; if the character appears in the *options* before the file name, the file name stays the same and the *options* change... probably not what you want. To get around this problem, you must give more of the *old* string, as in

```
r file1=file2
```

Combining Commands 5

Now let's look at some simple ways in which you can combine several commands on a single command line. The ability to string and combine multiple commands on the command line is one of the most powerful functions of the KornShell.

Multiple Commands

The MKS KornShell lets you type several commands on the same command line. To do this, just use a semicolon character to separate the commands, as in

```
cd mydir ; ls -l
```

If you have defined the alias

```
alias l="ls -l"
```

you can also say

```
cd mydir ; l
```

since you can use aliases like **`l`** after a semicolon.

Pipes

You can link a sequence of commands into a *pipeline*. A pipeline is written as

command | *command* |...

The commands are entered on the same line and separated by or-bar characters (|).

The idea of a pipeline is that the output from one command is pumped in as input to the next command. As an example of when this is useful, look at the Toolkit command

```
ls -l
```

ls displays a good deal of information about the contents of the current directory. If your directory is large, the information that **ls** displays is more than the screen can display, so information runs off the top of the screen.

You can avoid this problem by piping **ls** through the **more** command. **more** reads input from a file or the standard input and displays a screenful of it at a time. If you type

```
ls -l | more
```

the **ls** command produces its output and this output is piped into **more**. **more** then displays this output one screenful at a time. In general, you can pipe the output of any command into **more** to obtain paginated output:

```
ls | more
cat file1 file2 file3 | more
egrep "pattern" file | more
```

and so on.

The UNIX-inspired commands of PTC MKS Toolkit are particularly well-suited to being used in a pipeline. For example, the **fgrep** command searches for a particular string in a file or from standard input. A command like

```
history | fgrep "cp"
```

displays all the **cp** commands currently recorded in your history file.

```
ls -l | fgrep "Nov"
```

uses **ls** to obtain information on the contents of the current directory, then uses **fgrep** to search through this information and displays only the lines that contain the string `Nov`. This displays the files that were last changed in November.

Command Substitution

When the shell encounters a construct of the form

`$(`*command*`)`

or

```
‘command‘
```

in an input command line, the shell runs the given *command*. It then puts the output of the command, after converting newlines into spaces, back into the command line, replacing *command*, and runs the new line.

You will probably find the `$(...)` syntax easier to use for long command lines; however, the backwards apostrophes (grave accents) are more traditional and accepted on older UNIX shells.

As an example of how command substitution works, consider a file called `srclist`, containing the following list of source code file names:

```
alpha.c
beta.c
gamma.c
```

If you run the command

```
fgrep "printf" $(cat srclist)
```

the MKS KornShell uses **`cat`** to concatenate the contents of `srclist`, rewriting the original command line, so that this line appears as:

```
fgrep "printf" alpha.c beta.c gamma.c
```

This line is then run, with **`fgrep`** searching through the given files, displaying lines that contain the string `printf`. Programmers can use this kind of construction as a quick way to locate all references to a particular variable or function in the source code for a program.

The **`find`** command is often useful in command substitution constructs. **`find`** displays the names of files that have specified characteristics. For example,

```
find dir -name "*.c"
```

finds all files with the `.c` suffix under the `dir` directory.

```
ls -l $(find dir -name "*.c")
```

finds all the `.c` files then uses **`ls`** to display information about these files.

Complicating things further, you could enter

```
ls -l $(find dir -name "*.c") | fgrep "Nov"
```

This sets up a pipeline that only displays information on files that were last changed in November. (To be perfectly accurate, it also displays information on files that have the string `Nov` in their names too.) If there are a lot of these, you can paginate the output with

```
ls -l $(find dir -name "*.c") | fgrep "Nov" | more
```

Another useful **`find`** option has the form

`-ctime` *number*

This says that you want to find files that have changed in the last *number* 24-hour periods. For example,

```
ls -ld $(find dir -ctime 1)
```

displays **`ls`** information on all files that changed in the last 24 hours.

Search Rules

As mentioned earlier, command interpreters usually have to search for a file that contains the command that you want to run. When you are using the shell, you tell the shell where to search. Essentially, you give the shell a list of directories in which commands may be found. This list is called your *search rules*, because it tells the shell where you want to search.

When you are running inside the MKS KornShell, you set up search rules with a command of the form

`PATH='`*dir*`;`*dir*`;...'`

For example, you might say

```
PATH='c:/mks/mksnt;c:/windows;c:/usr/jean/bin;c:/games;c:/'
```

These directories are then searched by the shell, in the following order, when it searches for commands or shell scripts.

1. `c:/mks/mksnt`
2. `c:/windows`
3. `c:/usr/jay/bin`
4. `c:/games`
5. `c:/`

As soon as the shell finds a file with an appropriate name, it runs that file. If the command on the command line has a suffix, the MKS KornShell looks for a file with that name and suffix. If the command does not have a suffix, the shell looks for a file with the same base name and one of the suffixes

.com .exe .bat .sh .ksh

Under **`cmd.exe`**, the MKS KornShell looks for `.cmd` in addition to `.bat`.

Since the shell runs a command as soon as it finds a file with an appropriate name, pay close attention to the order in which directory names are given in your search rules. For example, these search rules specify the `c:/mks/mksnt` directory (where PTC MKS Toolkit commands are stored, assuming ***`ROOTDIR`*** is set to `c:/mks`) before the `c:/windows` directory. With this ***`PATH`***, if you type in a **`sort`** command, the shell uses the PTC MKS Toolkit version of **`sort`** instead of the standard one.

If you set up your ***`PATH`*** incorrectly, you can get the wrong command. You should probably always search the PTC MKS Toolkit commands directory first, as PTC MKS Toolkit commands work in completely different ways compared to standard commands of the same name. As well, some Toolkit commands run other Toolkit commands by name; they expect to get the PTC MKS Toolkit version of those commands, and do not work correctly if the standard PC version is found first.

It is often helpful to have the shell search your current directory for commands (in addition to the standard directories that contain commands). As an example, suppose that a programmer has different directories containing the source code for different programs. Under each directory, the programmer creates a shell script named `compile.ksh` that compiles all the source modules of the program in that directory. To compile a particular program, you just have to use **`cd`** to change to the appropriate directory and then type

```
compile
```

The shell searches the current directory, finds the `compile.ksh` shell script, and runs it. You can add the current directory to your search rules by putting in an entry without a name. For example,

```
PATH='c:/mks/mksnt;;c:/windows'
```

says that the current directory is to be searched after `c:/mks/mksnt` but before `c:/windows`.

```
PATH=';c:/mks/mksnt;c:/windows'
```

says that the current directory is to be searched before anything else.

```
PATH='c:/bin;c:/windows;'
```

ends in a semicolon. This means that the current directory is searched after everything else.

The best way to specify search rules is to put them into your profile file. That way, they are set up every time you log into the shell.

Note If you log into the shell and specify search rules in your profile file, the shell uses the given search rules. If you do not specify search rules, the shell uses the search rules that were active in the previous command interpreter. This means that if you are using both the shell and **`command.com`** or **`cmd.exe`**, set up your search rules in a form that is acceptable to both command interpreters.

`command.com` and **`cmd.exe`** always search the current directory, whether or not it is specified in the search rules. On the other hand, the MKS KornShell only searches the current directory if it is explicitly mentioned in *`PATH`*. If your *`PATH`* specifies the current directory, **`command.com`** and **`cmd.exe`** search the current directory twice.

The whence Command

With aliases and search rules, it can be easy to lose track of what is actually run when you type in a command. The **`whence`** command can reduce the confusion.

`whence` *command*

tells you the file that is run if you type a command line that begins with the given command. For example,

```
whence find
```

tells you what file is run if you type in a command line beginning with **`find`**. This lets you sort out how the search rules work and what effect aliases have.

Shell Scripts 6

So far we have discussed how the shell makes it easier for you to enter single commands. This section discusses how the shell makes it easier for you to perform sequences of commands.

Most people find themselves using some sequences of commands over and over again.

- A programmer may always use the same commands to compile source code, and link the resulting object code.
- A bookkeeper may have to go through the same sequence of commands each week to update the books and produce a report.
- A person who is writing a document may go through the same sequence of commands to combine separate chapters into a single file, to format the file, and finally to print the file.

To simplify such jobs, the shell lets you run a sequence of commands that have been stored in a normal text file. For example, the programmer stores all the appropriate compiling and linking commands in a single file. Once this is done, the programmer can run all the commands by instructing the shell: "Run everything in this file."

A file containing commands in this way is called a shell program or *shell script*. Shell scripts have several advantages over typing the commands individually.

- They reduce the amount of typing you have to do. You only have to type in the shell script once. From that point onward, you can run all the commands in the script using a single command to the shell.
- They reduce the number of errors. If you are typing in ten commands, you have ten chances to make a mistake. With a shell script however, you can take your time, edit the file carefully, and get it right before you try to run it. Once you get the shell script right, you don't have to worry about typing mistakes any more.

- They are easy for other people to use. For example, consider the bookkeeper mentioned earlier. When the bookkeeper goes on vacation, someone else has to fill in. It is much easier for the fill-in to type a single command which does everything correctly than to try to type in the full sequence of commands.

Chapter 10: "Graphical Features of the MKS KornShell" on page 59 describes some of the Windows specific facilities for writing shell scripts in a graphical environment.

For all these reasons, you will probably find that using shell scripts makes your work easier and more productive. This chapter can only scratch the surface, but it should give you an idea of how to write shell scripts and use them profitably.

Running a Shell Script

The command

```
sh file
```

runs a shell script stored in the given file. This works from **`command.com`** or **`cmd.exe`** as well as from the MKS KornShell, provided that your search paths are set up to find the **`sh`** command.

If you are running under the MKS KornShell itself, you can usually run a shell script simply by typing the name of the file that contains the script. For example, suppose you have a file named shell.scp which contains a script; if you type

```
shell.scp
```

at the shell prompt, the shell runs the script.

You can leave off the suffix part of a shell script file name if the suffix is `.ksh`. For example, to run `script.ksh`, you only have to type

```
script
```

if you are already using the shell.

A Sample Shell Script

As an example of a simple shell script, let's look at the situation of a programmer who wants to compile a collection of files written in the C programming language. Assume that the PTC MKS Toolkit **`cc`**

command has been set up to compile any file *file*`.c` and leave the resulting object module in the file *file*`.obj`. Also assume that the hypothetical command **`ld`** links object files together. The shell script

```
cc file1.c
cc file2.c
cc file3.c
 ...
ld file1.obj file2.obj ...
```

compiles everything and links the result.

You can create this script with any text editor. If you store it in the file `compile`, it can be run with the single command

```
compile
```

You only have to type the shell script once; from that point onward, you can run all the commands in the script by typing a single line. This can save you a lot of time and effort if there are a lot of files, or if some command lines have a lot of options. Not only does it reduce the amount of typing you have to do, it reduces errors.

Shell Scripts and Command Interpreters

The `.bat` suffix is typically used for *command files* intended to run under the **`command.com`** command interpreter. Similarly, the `.cmd` suffix is used for command files intended to run under **`cmd.exe`**. Such command files are similar to shell scripts, in that they consist of a sequence of commands to be run by the command interpreter. Indeed, if a file just contains a sequence of commands, without using any of the special features of a command interpreter, you can run the file as either a shell script or a command file.

To make things easier, on Windows 95/98/Me using **`command.com`**, the MKS KornShell looks for al three of

```
file.bat
file.sh
file.ksh
```

when you attempt to run a shell script by typing

```
file
```

Similarly, on Windows NT/2000 using **`cmd.exe`**, the KornShell looks for

```
file.bat
file.cmd
file.sh
file.ksh
```

Using `.bat` or `.cmd` is a good idea if your file is just a straightforward sequence of commands; however, if a shell script uses any of the special features of the shell, give the file the `.ksh` suffix so that you know it is specifically a script for the MKS KornShell.

Shell Variables

You can think of shell scripts as programs made up of commands. To allow more versatile shell scripts, the shell supports many of the features of normal programming languages.

In a conventional programming language, a *variable* is a name that has an associated value. Any time you want to use the value, you can use the variable name instead.

The MKS KornShell also lets you create variables. A shell variable name can consist of uppercase or lowercase letters, plus digits and the underscore character (_). The name can have any length, but the first character may not be a digit. Uppercase letters are distinguished from lowercase ones, so

```
NAME
name
Name
```

are all ***different*** names.

To create a shell variable, just type

name`='`*string*`'`

as a command to the shell. Note that no spaces are allowed around the `=`. For example,

```
HOME='c:/usr/jean'
```

sets up a variable with the name ***`HOME`*** and the value `c:/usr/jean`.

Any command can use the value of a variable with a dollar sign `$` followed by the variable name. For example, if ***`HOME`*** is `c:/usr/jean`,

```
cd $HOME
```

is equivalent to

```
cd c:/usr/jean
```

Similarly,

```
cp $HOME/* /newdir
```

is equivalent to

```
cp c:/usr/jean/* /newdir
```

To change the value of an existing variable, you use a command with the same form. For example,

```
HOME='c:/usr/george'
```

changes the current value of ***HOME***.

If the value on the right hand side of the = sign does not contain any shell special characters, you can leave out the apostrophes. For example, you can just say

```
HOME=c:/usr/george
```

Displaying Shell Variables

You can display your variables and their values by typing

```
set
```

Other parts of this document discuss some of these predefined variables; for complete information, see the **sh** reference page in the online *PTC MKS Toolkit Utilities Reference*.

If you do this, you will probably see many variables that you don't recognize. These are *predefined* variables, set up with default values at the time that you invoke the shell.

You can display the value of a single variable with the **echo** command. For example,

```
echo $HOME
```

displays the current value of the ***HOME*** variable.

You can also display and edit variables in a dialog box by entering the **gvar** command.

Calculations with Variables

Suppose you run the following commands under the shell (either in a shell script or by typing in one command after another).

```
i=1
j=$i+1
echo $j
```

The output of **echo** is `1+1`. The reason is that a normal variable assignment assigns a *string* to a variable. Thus `j` gets the string `1+1`.

If you want to *evaluate* an arithmetic expression, you must use

```
let "variable=expression"
```

This assigns the value of an expression to the given variable. For example, in

```
i=1
let "j=$i+1"
echo $j
```

the **echo** command displays the value `2`. You can also say

```
i=1
let "i=$i+1"
echo $i
```

In this case, the **let** command changes the value of `i`. The new value of `i` is the old value plus 1.

A **let** command can have any of the standard arithmetic expressions:

`-A`	negative A
`A*B`	A times B
`A/B`	A divided by B
`A%B`	remainder of A divided by B
`A+B`	A plus B
`A-B`	A minus B

The standard mathematical order of operations is used, as shown in the way that operations are grouped: all unary minus operations are carried out, then any `*`, `/`, and/or `%` operations (from left to right in the order they appear), then any additions or subtractions (from left to right in the order they appear). Many operators use special shell

characters, so you usually need to put double quotes around the expression to protect these operators from being misinterpreted by the shell. Thus

```
let "i=5+2*3"
```

assigns 11 to `i` since the multiplication is done first. You can use parentheses in the usual way to change the order of operations. For example,

```
let "i=(5+2)*3"
```

assigns 21 to `i`.

Note **`let`** does not work with numbers that have fractional parts. It only works with integers.

Exporting Variables

Up to this point, this document has only talked about defining shell variables and then using them in later command lines. It is also possible to define a shell variable and then call a shell script that makes use of that variable; but you have to do a certain amount of preparation first.

A shell script is run like a separate shell session. By default, it does not share any variables with your current shell session. If you define a variable `VAR` in the current session, it is *local* to the current session; when you call shell scripts, they won't know about `VAR`.

To deal with this situation, you can issue the command

```
export VAR
```

The **`export`** command says that you want the variable *`VAR`* passed on to all the commands and shell scripts that you run in this session. Once you do this, `VAR` becomes *global* and the variable is known to all the commands and shell scripts that you use.

As an example, suppose you issue the commands

```
MYNAME="Robin Hood"
export MYNAME
```

From this point onward, all your commands can use the `MYNAME` variable to obtain the associated name. You may, for example, have shell scripts which write up form letters that contain your name, obtained from the `MYNAME` variable.

When a script begins running, it automatically inherits all the variables currently being exported. However, if the script changes the value of one of those variables, that change is ***not*** reflected to the calling shell.

By default, any variables created within a shell script are *local* to that script. This means that when another program is run, it does not see those variables in its environment. However, the script can use the **`export`** command to turn local variables into global ones.

Inside a shell script,

```
export name
```

indicates that the variable with the given *name* is to be exported. When other programs are run from that script, they inherit the value of all exported variables. However, when the script terminates, all its exported variables are lost to the calling shell.

Some variables are automatically marked for export by the software that creates them. For example, if you log into the shell, the login procedure automatically marks the ***`HOME`*** variable for export so that other commands and shell scripts can use it. Other variables must be explicitly exported. For example,

```
export PATH
```

is commonly used so that search rules and changes to search rules are automatically shared by all shell sessions and scripts.

Variable Attributes

The **`typeset`** command lets you associate attributes with shell variables. This process is analogous to declaring the type of a variable in a conventional programming language. For example,

```
typeset -i8 y
```

says that `y` is an octal integer. In this way, you can make sure that arithmetic with `y` is always performed in base eight rather than the usual base ten.

Other attributes may specify how the variable's value is displayed when the variable is expanded. Attributes of this kind are:

-L *n* Value is always displayed with *n* characters, with the value left-justified within that space.

-R *n* Value is always displayed with *n* characters, with the value right-justified within that space.

-RZ *n* Value is always displayed with *n* characters, with the value right-justified and enough leading zeroes to fill out the rest of the space.

-Z *n* Same as **-RZ***n*.

-LZ *n* Value is always displayed with *n* characters, with the value left-justified and leading zeroes stripped off.

All of these options may lead to truncation of the value if it is longer than the specified length.

Variables with string values may be typeset with the **-u** attribute. This means that whenever such a variable is assigned a new value, all lowercase letters in the value are automatically converted to uppercase. Similarly, the **-l** attribute means that whenever a variable is assigned a new value, all uppercase letters in the value are automatically converted to lowercase.

The read-only attribute **-r** is useful when a variable is marked for export. The command

```
typeset -r name
```

says that the variable *name* cannot be changed from its present value. This makes it impossible for subsequent commands to change this value. You may also use the format

```
typeset -r name=value
```

which sets the variable to the given value, then marks it read-only so that the value cannot be changed. The command

```
typeset
```

with no arguments displays the currently defined variables and their attributes. A useful variation is

```
typeset -x
```

which displays all the variables which are currently defined for export.

Positional Parameters

The script described in "A Sample Shell Script" on page 24 was designed to compile and link a program stored in a collection of source modules. This section discusses a shell script that can compile and link a C program stored in any file.

To create such a script, you need to be familiar with the idea of *positional parameters*. When the shell encounters a construct formed by a `$` followed by a number, the shell replaces the construct with a value taken from the command line that invoked the shell script. `$1` refers to the first string after the name of the script file on the command line, `$2` refers to the second string, and so on.

As a simple example, consider a shell script consisting only of the command

```
echo $1
```

Suppose this script is contained in the file `echoit.ksh` and suppose we run the command

```
echoit hello
```

the shell reads the shell script from `echoit.ksh` and tries to run the command it contains. When the shell sees the `$1` construct in the **`echo`** command, it goes back to the command line and obtains the first string following the name of the shell script on the **`sh`** command line. The shell replaces the `$1` with this string, so the **`echo`** command becomes

```
echo hello
```

The shell then goes on to run this command.

A construct like `$1` is called a *positional parameter*. Parameters in a shell script are replaced with strings from the command line when the script is run. The strings on the command line are called *positional parameter values* or *command line arguments*.

Command line arguments may be enclosed in quotes (single or double). For example,

```
echoit "Hello there"
```

echoes the two words instead of just one. If you say

```
echoit Hello there
```

the string `Hello` is considered parameter value `$1` and `there` is `$2`. Of course, the shell script is only

```
echo $1
```

so the **echo** command displays only the `Hello`.

Going back to the compile and link example, you can write a general shell script as

```
cc $1.c
ld $1.obj
```

As before, assume that **cc** compiles a `.c` source file to produce a `.obj` object file, and that **ld** is a command that links a `.obj` object file to get an executable file. If you store this in the file `clink.ksh`, you can say

```
clink prog
```

to compile and link `prog.exe`. The commands that the shell runs are

```
cc prog.c
ld prog.obj
```

In the same way, the command

```
clink dir/prog2
```

compiles and links `dir/prog2.c`. You can use this shell script to compile and link a C program stored in a single file.

As another example of a shell script containing a parameter, suppose that the file lookup.ksh contains

```
fgrep $1 address
```

(where address is a file containing names, addresses, and other useful information). A command like

```
lookup Smith
```

displays address information on anyone in the file named Smith.

Quotes in Shell Scripts

A $*n* construct (that is, a positional parameter) in a shell script may be enclosed in quotes. When double quotes are used, the parameter is replaced by the appropriate value from the command line. For example, suppose the file `search.ksh` contains

```
fgrep "$1" *
```

If you issue the command

```
search 'two words'
```

the parameter value '`two words`' replaces the construct `$1` in the **fgrep** command, making

```
fgrep "two words" *
```

If the **fgrep** command does not contain the double quotes, the parameter replacement results in

```
fgrep two words *
```

which has an entirely different meaning.

When you use single quotes to enclose a $*n* construct in a shell script, the $*n* is ***not*** replaced by the corresponding parameter value. For example,

```
fgrep '$1' *
```

searches for the string `$1`. The `$1` is not replaced by a value from the command line. In general, single quotes are stronger than double quotes.

Control Structures

We have already mentioned that the shell provides facilities that are similar to those found in programming languages. The sections to come discuss some of the *control structures* of the shell, related to programming control structures like **if** statements and **while** loops.

Testing Conditions

Before discussing the various control structures, it is useful to talk about ways to test for various conditions.

The **test** command tests to see if something is true. The following tables shows the ways it can be used.

For a full description of the test command, see the **test** reference page in the online *PTC MKS Toolkit Utilities Reference*.

Examine the nature of files

test -d *pathname*	is *pathname* a directory?
test -f *pathname*	is *pathname* a file?
test -r *pathname*	is *pathname* readable?
test -w *pathname*	is *pathname* writable?

Compare the age of two files

test *file1* **-ot** *file2*	is *file1* older than *file2*?
test *file1* **-nt** *file2*	is *file1* newer than *file2*?

Compare the values of two numbers *A* and *B*

test *A* **-eq** *B*	is *A* equal to *B*?
test *A* **-ne** *B*	is *A* not equal to *B*?
test *A* **-gt** *B*	is *A* greater than *B*?
test *A* **-lt** *B*	is *A* less than *B*?
test *A* **-ge** *B*	is *A* greater than or equal to *B*?
test *A* **-le** *B*	is *A* less than or equal to *B*?

Compare two strings *s1* and *s2*

test *s1* **=** *s2*	is *s1* equal to *s2*?
test *s1* **!=** *s2*	is *s1* not equal to *s2*?

Test whether strings are empty

test *string*	is *string* not empty?
test -z *string*	is *string* empty?
test -n *string*	is *string* not empty?

In all of these cases, the result of **test** is true or false. (To be precise, **test** returns a status of zero if the test turns out to be true and a status of one if the test turns out to be false.)

You can use **-n** to check if a variable has been defined. For example,

```
test -n "$HOME"
```

is `true` if ***HOME*** exists, and `false` if you have not created a ***HOME*** variable.

If *expression* is one of the expressions recognized by **test**, then

```
test ! expression
```

returns false if *expression* is true, and true if *expression* is false. For example,

```
test ! -d pathname
```

is true if *pathname* is not a directory, and false otherwise.

The if Conditional

An **if** conditional runs a sequence of commands if a particular condition is met. It has the form

```
if condition
then commands
fi
```

The end of the commands is indicated by **fi** (which is **if** backwards). For example,

```
if test -d $1
then lc $1
fi
```

This tests to see if the string associated with the first positional parameter is the name of a directory. If so, it runs an **lc** command to display the contents of the directory.

Any number of commands may come between the **then** and the **fi** that ends the construct. For example, you might have written

```
if
    test -d $1
then
    echo "$1 is a directory"
    lc $1
fi
```

This example also shows that the commands do not have to begin on the same line as `then`, and the condition being tested does not have to begin on the same line as **if**. The condition and the commands are indented to make them stand out more clearly. This is a good way to make your shell scripts easier to read.

Another form of the **if** conditional is

```
if condition
then commands
else commands
fi
```

If the condition is true, the commands after the then are run; otherwise, the commands after the **else** are run. For example, suppose you know that the string associated with the variable pathname is the name of either a directory or a file. Then you can write:

```
if
    test -d $pathname
then
    echo "$pathname is a directory"
    lc $pathname
else
    echo "$pathname is a file"
    more $pathname
fi
```

If the value of `pathname` is the name of a file, this shell script uses **`echo`** to display an appropriate message, then uses **`more`** to display the contents of the file.

The final form of the **`if`** construct is

```
if condition1
then commands1
elif condition2
then commands2
elif condition3
then commands3
      ...
else commands
fi
```

`elif` is short for *else if*. In this example, if *condition1* is true, *commands1* are run; otherwise, the shell goes on to check *condition2*. If that is true, *commands2* are run; otherwise, the shell goes on to check *condition3* and so on. If none of the test conditions are true, the *commands* after the **`else`** are run. The following example shows how this can be used.

```
if test ! "$1"
then
    echo "no positional parameters"
elif test -d $1
then
    echo "$1 is a directory"
    lc $1
elif test -f $1
then
    echo "$1 is a file"
    more $1
else
    echo "$1 is just a string"
fi
```

The **`test`** after the **`if`** determines if the value of the first positional parameter is an empty string. If so, it means that there are no positional parameters, so the shell script uses **`echo`** to display an

appropriate message; otherwise, the script checks if the parameter is a directory name; if so, the directory's contents are listed with **lc**. If that doesn't work, the script checks if the parameter is a file name; if so, the contents of the file are listed with **more**. Finally, if none of the previous tests work, the parameter is assumed to be an arbitrary string, and the script displays a message to this effect.

You can put this script into a file named `listit.ksh` and run commands of the form

```
listit name
```

The shell script does its best to list the contents of *name* in some useful form.

The while Loop

The **while** loop repeats one or more commands while a particular condition is true. The loop has the form

```
while condition
do commands
done
```

The shell first tests to see if *condition* is true. If it is, the shell runs the given *commands*. It then goes back to check the *condition*. If it is still true, the shell runs the *commands* again, and so on, until the *condition* is found to be false.

As an example of how this can be used, suppose a programmer wants to run a program named **prog** 100 times to get an idea of the program's average running speed. The following shell script does the job.

```
i=100
date
while test $i -gt 0
do
    prog
    let i=$i-1
done
date
```

The script begins by setting a variable `i` to 100. It then uses the **date** command to get the current date and time. (Note that this is the **date** command from PTC MKS Toolkit, not the **command.com** or **cmd.exe date** command.)

Next the script performs a **while** loop. The **test** condition says that the loop is to keep on going as long as the value of `i` is greater than zero. The commands of the loop run **prog** and then subtract one from the i variable. In this way, `i` goes down by 1 each time through the

loop, until it is no longer greater than 0. At this point, the loop stops and the final instruction of the script displays the date and time at the end of the loop. The difference between the starting and ending times gives you an idea of how long it took to run the program 100 times.

(Of course, the shell itself takes some time to perform the **`test`** and to do the calculations with `i`. If **`prog`** takes a long time to run, the time spent by the shell is relatively unimportant; if **`prog`** is a quick program, the extra time that the shell takes may be large enough to invalidate the timing.)

You can rewrite this shell script to make it a little more efficient.

```
i=100
date
while let "(i=i-1)>=0"
do
    prog
done
date
```

In this example, the **`let`** command is the condition of the **`while`** loop. It gives `i` a new value and then compares this value to zero. The advantage of writing the program this way is that it does not have to call **`test`** to make the comparison; this speeds up the loop and makes the time more accurate.

The for Loop

The final structure we'll examine is the **`for`** loop. It has the form

```
for name in list
do commands
done
```

The *name* is a variable name; if this variable doesn't exist, it is created. The *list* is a list of strings separated by spaces. The shell begins by assigning the first string in *list* to the variable *name*. It then runs the *commands* once. When the commands have been run, the shell assigns the next string in *list* to *name*, and repeats the *commands*. The shell goes through the *commands* once for each string in *list*.

As a simple example of a shell script that uses **`for`**, consider

```
for file in *.c
do
    cc $file
done
```

When the shell looks at the `for` line, it expands the expression `*.c` to produce a list containing the names of all files (in the current directory) that have the suffix `.c`. The variable `file` is assigned each

of the names in his list, in turn. The result of the **for** loop is to use the **cc** command to compile all .c files under the current directory. If you want, you can write

```
for file in *.c
do
    echo $file
    cc $file
done
```

so that the shell script displays each file name before compiling it. This lets you keep track of what the script is doing.

As you can see, the **for** loop is a powerful construct. The list can also be created with command substitution, as in:

```
for file in $(find . -name "*.c")
do
    echo $file
    cc $file
done
```

This uses the **find** command to find all `.c` files under the current directory, and then compiles these files. This is similar to the previous shell script, but also looks at subdirectories of the current directory.

Combining Control Structures

You can combine control structures by nesting (that is, putting one inside another). For example,

```
for file in $(find . -name "*.c")
do
    if test $file -ot $1
    then
        echo $file
        cc $file
    fi
done
```

This shell script takes one positional parameter, giving the name of a file. The script looks under the current directory and finds the names of all `.c` files. The **if** construct inside the **for** loop tests each file to see if it is older than the file named on the command line. If the `.c` file is older, **echo** displays the name, and the file is compiled. You can think of this as bringing a set of files *up-to-date* with the file name specified on the command line.

Functions

Shell functions are similar to subprograms in standard programming languages: it is a sequence of commands aimed at doing a single job. Typically, functions are used for operations that you tend to do frequently during a session. For example, consider this function:

```
function temphome
{
    curdir=$(pwd)
    cd $HOME
    $1
    cd $curdir
}
```

The first line indicates that this is a function named `temphome()`. The commands that make up this function are enclosed in brace brackets following the heading line. The first command

```
curdir=$(pwd)
```

runs the **`pwd`** command to get the name of the current directory, and assigns this directory name to the variable `curdir`. The next instruction goes to your home directory. After that comes a line that consists only of the positional parameter `$1`, and finally a **`cd`** command that uses curdir to go back to our original directory.

The purpose of `temphome()` is to go to the home directory, run a command, then return to the directory from which it was called. To call this function, type `temphome` followed by the command you want to run. For example,

```
temphome lc
```

runs **`lc`** on your home directory. Notice that **`lc`** is the first positional parameter, so it is run in place of `$1` within the `temphome()` function. To run a longer command, put it in quotes, as in

```
temphome "lc srcdir"
```

What's the point of `temphome()`? There are often situations where you want to run a particular command in your home directory (for example, the command makes use of files in the current directory, and the files you want to use are in your home directory). If you are working in a different directory, it's annoying to have to **`cd`** to your home directory, then **`cd`** back. A function like `temphome()` does the work for you. Of course, with the **`lc`** examples, you can use **`lc`** directly, without going through the trouble of using temphome(); however, with other commands `temphome()` may save you some trouble.

Anytime you find yourself doing the same sequence of commands in a shell session, ask yourself if it is simpler to define a function to do the same thing. You can put a function definition in your profile file and be able to use it whenever you log into the shell.

Returning Values from a Function

Just as a single command sometimes returns an exit status, a function can also return a value. If the statement

```
return expression
```

appears inside a function, the function terminates and the value of the *expression* is returned as the status or *result* of the function. In general, returning a zero means that the function has succeeded in its task; returning a one means that the function has failed. For example, here's a function that is similar to our `temphome()` example in the previous section.

```
function td
{
    if test -d "$1"
    then
        curdir=$(pwd)
        cd $1
        $2
        cd $curdir
        return 0
    else
        echo $1 "is not a directory"
        return 1
    fi
}
```

The `td()` function takes two arguments. The first is supposed to be a directory name. The **`test`** command checks to see if it really is a directory name; if so, `td()` temporarily goes to that directory, runs a single command (the second positional parameter), then returns zero to indicate success. If the first parameter is not an existing directory name, the function displays an appropriate message and returns one to indicate failure. You might call this function with the line

```
td srcdir lc
```

which lists the contents of the directory `srcdir`, if that directory exists.

Customization 7

Customization is the process of changing something to suit your own tastes. The ability to customize the way you use the computer is central to the philosophy of the PTC MKS Toolkit—you should be able to work *your* way, not someone else's.

This tutorial has already examined many of the features that let you customize the MKS KornShell to your own preferences. For example, aliases play a large role in customization.

Customizing Aliases

You have already seen that you can create aliases that let you give your own names to commands. This is certainly one aspect of customization; however, there is another aspect that has not been discussed yet.

Let's take an example. The **egrep** command searches through files and displays lines that contain a requested string. For example,

```
egrep hello file
```

displays all the lines of *file* that contain the string `hello`. Normally, **egrep** distinguishes between uppercase and lowercase letters; this means that this search does ***not*** display lines that contained `HELLO` or Hello. If you want **egrep** to ignore the case of letters as it searches, you must specify the **-i** option, as in

```
egrep -i hello file
```

This finds `hello`, `HELLO`, `Hello`, and so on.

There are probably many users who prefer to use the **-i** version of **egrep** most of the time—it certainly makes sense in many cases. If so, you can define the alias

```
alias egrep="egrep -i"
```

From this point on, if you use the command

```
egrep string file
```

it is automatically converted to

```
egrep -i string file
```

and you get the case-insensitive version of the command **egrep**. It may seem odd to define an alias that has the same name as a command that is used in the alias, but this is so common that the MKS KornShell checks specially for an alias of the same name, and does the correct thing.

As another example, the **rm** command to delete a file has an **-i** option that asks you

```
"Ok to delete?"
```

before it actually removes a file. If you like this extra bit of safety, you might define

```
alias rm="rm -i"
```

After this, when you call **rm**, it automatically checks with you before deleting a file, just to make sure that you really want to delete it.

For more information, see the **gvar** reference page in the online *PTC MKS Toolkit Utilities Reference*.

You can also set and display aliases from a dialog box with the **gvar** command.

If you find yourself using the same option every time you call a command, you might consider creating an appropriate alias so that the shell automatically adds the option every time you call the command. Of course, the best place to define this alias is in your profile file, so that the alias is set up every time you start a session.

Turning Off an Alias

If you have set up an alias like the one for **rm**, you may find that you ***do not*** want the alias to apply in some situations. For example, when you delete a huge number of files, you probably do not want **rm** to ask if it is okay to delete each one. In this situation, you have several options that should help you:

- Get rid of the alias entirely. The command

  ```
  unalias rm
  ```

  gets rid of the **rm** alias. After this, when you type **rm**, you get the real **rm** command.

- Escaping the alias. As you have already seen, putting a backslash in front of special characters tells the shell to use the literal meaning of the character. The same is true of aliases.

  `\rm` *file*

  tells the shell not to replace the **`rm`** with its alias.

- Quoting the alias. This method is similar to escaping the alias. The quotes tell the shell to use the literal meaning of the string contained within them.

  `'rm'` *file*

- Specify the full path name. For example,

  exec_pathname`/rm` *file*

  (where *exec_pathname* is the path name under which the executables are stored, usually *`ROOTDIR`*`/mksnt/`) tells the shell to run the program **`rm`** found in the given path. The shell does not perform alias substitution when a command is specified as a path name in this way.

Prompts

By default, the MKS KornShell prompts for command input with a `$` followed by a space. Many users prefer to change this to something more personal.

To determine the command prompt, the shell looks at the *`PS1`* variable. Thus you can change your prompt by changing the variable's value. For example,

```
PS1="jean% "
```

sets your command prompt to `jean%`. After this, the shell prompts with

```
jean%
```

when it wants you to input a command.

If the string associated with *`PS1`* contains a single `!`, the shell replaces the character with the command number from the history list. For example, if you set up

```
PS1="jean-!% "
```

you get prompts of

```
jean-1%
jean-2%
...
```

The great benefit of this is that it simplifies the job of rerunning a previous command with **r**—you do not have to look up the command's number in the history list, because the number is right there in the prompt.

By default, when the shell prompts for the continuation of a construct (for example, a **for** loop), it displays >. You can change this prompt by assigning a new value to the *PS2* variable. For example,

```
PS2="continue: "
```

displays

```
continue:
```

when prompting for the continuation of a construct. Having a longer prompt sometimes helps to remind you that you are in the middle of something.

For more information, see the **gset** reference page in the online *PTC MKS Toolkit Utilities Reference*.

You can also set your system prompts from a dialog box using the **gset** command.

Of course, the best place to assign different values to the prompt strings is inside your profile file, so that the prompts are set up every time you start a session.

Setting Options

The **set** command lets you set options for your MKS KornShell session. To turn an option on, use

```
set -o name
```

where *name* is the name of the option you want to turn on.

To turn an option off, use

```
set +o name
```

For a complete list of options, see the **set** reference page in the online *PTC MKS Toolkit Utilities Reference*.

Following are some of the options you may find useful.

```
set -o allexport
```

indicates that you want to export every variable which is assigned a value. This exports all variables that currently have values, plus all variables assigned a value in future.

```
set -o noclobber
```

indicates that you do not want the > redirection operator to overwrite existing files. If you specify the construct >*file*, the redirection only works if *file* does not already exist. If you have this option on and you really do want to redirect output into an existing file, you must use >|*file* (with an or-bar after the >) to indicate output redirection.

```
set -o noglob
```

tells the shell not to expand wild card characters in file names. This is occasionally useful if you are typing in command lines that contain a number of characters that are normally expanded.

```
set -o verbose
```

tells the shell to display its input on the screen as the input is read. This lets you keep track of material that comes from a file.

```
set -o
```

displays all current option settings.

The ENV Variable

So far, we have discussed customization inside your profile file, but the shell only reads your profile file when you log on or when you invoke the shell with the **`-L`** option. How can you customize a shell session when you invoke the shell in some other way, for example, from within an editor like **`vi`**?

When you invoke the shell, it looks for an environment variable named *`ENV`*. If this variable exists, its value is assumed to be the name of a file containing shell commands. The shell executes the commands in this file before proceeding with the rest of the session; therefore, you can use the *`ENV`* variable to point to a *set-up* file which sets things up in the same way that the profile file does.

For example, you might put all your alias definitions and other set-up instructions into a file called `/mysetup.ksh`. You want these instructions executed the first time your shell starts executing (after

booting) and whenever you explicitly call the shell during a session (for example, as a subshell to execute a shell script). To make sure *ENV* is set up after booting, put the following into your `autoexec.bat` file:

```
ENV=/mysetup.ksh
ENV=mysetup
```

To make sure *ENV* is set up when you execute a subshell, put the following into your `profile.ksh` file:

```
export ENV=/mysetup.ksh
```

You might find it useful to move all your aliases out of your profile file and into your *ENV* file instead; however, you should keep exported variable assignments in your profile, so that they are only executed once.

Command Editors 8

If you are an experienced computer user, you may already be familiar with the notion of *command editing*. If you have never seen command editing before, the best way to see how it works is to start with an example.

Suppose you want to find a file that you know is in a subdirectory of a directory. There are plenty of clever ways to look for the file (for example, by using the **find** command), but if you are not exactly sure what you are looking for, you might easily go through a sequence of commands like this:

```
lc /dir
lc /dir/subdir
lc /dir/subdir/subsub
lc /dir/subdir/subsub/subsubsb
lc /dir/subdir/subsub/subsubsb/more
```

and so on. (Yes, you can also use the **cd** command to reduce the amount of typing during this search, but often you do not want to leave the directory you are in.)

As you can see, each **lc** command consists of the previous command plus some added material. If you can start with the previous command instead of a blank command line, you do not have nearly as much typing to do.

That is the idea behind command editing. Command editing lets you access commands from your history file, edit them, and run the result. You have already seen this process before, when discussing some of the features of the **r** command; but this chapter looks at a simpler way of doing things.

Asking for the vi Command Editor

You can also select your edit mode from a dialog box with the **gset** command. For more information, see the **gset** reference page in the online *PTC MKS Toolkit Utilities Reference*.

If you run the MKS KornShell command

```
set -o vi
```

it tells the shell that you want the ability to edit commands the way that you normally edit text with **vi**.

Once you have done

```
set -o vi
```

you are set up for **vi** command editing. Whenever the shell prompts you for input, it is as if the shell puts you into **vi**'s insert mode on a new line at the end of the history file. You can type in a new command line just as you normally would.

You can also press ESC to enter a **vi**-like command mode. When you enter command mode, you can use the usual cursor movement commands to move around on the command line, or to move up and down in the history file. For example, if you press the ↑ key (or k), you move to the previous line in the history file (the last command line you entered). Press ↑ again, and you move to the line before that. Press ↓ (or l) and you move forward in the history file.

In this way, it's simple to retrieve recent commands from the history file. You can then edit them using standard **vi** commands. For example, you can use $ to move to the end of the line, and a to begin appending text to the end of the line. When you have edited the line to produce the command that you want to run, simply press ENTER to run that line.

As you would expect, you can use the search commands

/*string*
?*string*

to search backwards and forwards through the history file. You can edit the command line with

For a complete list of available commands, see the **shedit** reference page in the online *PTC MKS Toolkit Utilities Reference*.

- w – move to next word
- b – move to previous word
- d – delete
- c – change
- a – append
- i – insert
- u – undo

and many of the other **vi** commands we have discussed.

Asking for the EMACS Command Editor

If you are familiar with the EMACS editor, you may prefer

```
set -o emacs
```

This allows you to use commands identical to those of EMACS to edit your shell command line.

For full details on using the EMACS command editor, see the **shedit** reference page in the online *PTC MKS Toolkit Utilities Reference*.

Using a Command Editor

If you have never used command editing before, it takes some time to realize when it can be useful. Look for times when you are running the same sequence of commands, or slight variations on the same sequence of commands. Look especially for long commands that normally require a lot of typing; the whole point of command editing is to save yourself the trouble of typing the same thing over and over again. Command editing is also very useful when you have made a mistake in typing a command line and wish to correct it. If you remember that the command editing facilities are there, you will probably soon see opportunities to use them.

Miscellaneous Points 9

This chapter covers additional topics that may contribute to your productive use of the shell.

Tilde Expansion

After all alias substitution has taken place, the shell looks to see if anything on the command line begins with the tilde (~) character. If so, the shell checks on everything from the tilde up to the next slash character; if it is one of a set of recognized sequences, the shell replaces the sequence with another value.

For example, a tilde by itself stands for your home directory (that is, the directory given by your *HOME* environment variable).

```
cp ~/file1 file2
```

copies `file1` under your home directory into `file2`. This works regardless of what your current directory is.

```
cp file1 ~/dir
```

copies `file1` from the current directory into `dir` under your home directory.

A tilde followed by + stands for the variable *PWD* (which contains the name of your current working directory). A tilde followed by - stands for the variable *OLDPWD* (which gives the name of the working directory you were in immediately before the last **cd** command).

A tilde followed by a person's login name stands for that person's home directory; therefore,

```
more ~jsmith/profile.ksh
```

displays the profile file of `jsmith`, from his or her home directory. Notice that you do not have to know what the person's home directory is; the shell looks up that information for you.

These sequences are the only tilde expansions recognized by the MKS KornShell. If a tilde is followed by any other sequence, the entire sequence is left unchanged.

Modified Parameter and Variable Expansion

In shell scripts or functions, a `$` followed by a number stands for a positional parameter to the script or function. For example, if the command

```
echo $1
```

appears in a shell script, it will **echo** the first positional parameter. Similarly, a `$` followed by the name of a shell variable stands for the value of the variable.

These constructs are called *parameter expansions*. In this sense, *parameter* can mean either a positional parameter or a shell variable.

The MKS KornShell also supports more complicated forms of parameter expansions, letting you obtain only part of a parameter value or a modified form of the value. The easiest way to understand this is to look at some examples.

${*parameter*:-*value*}

can be used in any input to the shell. If *parameter* currently has a value and the value is not null (for example, a string without characters), this construct stands for the parameter's value; if the parameter does not have a non-null value, this construct is replaced with the *value* shown in the brace brackets. For example, a shell script might contain

```
SHELL=${SHELL:-bin/sh}
```

If the *SHELL* variable currently has a value, this assignment simply assigns *SHELL* its own current value; however, if *SHELL* does not have a non-null value, this assignment gives it the value of `bin/sh`. The value after :- can be thought of as a back-up value in case the parameter itself does not have a value. As another example, consider the following example that might occur in a shell script:

```
cp $1 ${2:-$HOME}
```

If both positional parameters are present with non-null values, the copy command is just

```
cp $1 $2
```

However, if you call the shell script without specifying a second positional parameter, it uses the back-up value of ***HOME*** and the result is equivalent to

```
cp $1 $HOME
```

The expansion form

${*parameter*:=*value*}

is similar to the previous form; the difference is that if the given *parameter* does not currently have a value, the given *value* is assigned to *parameter*, and then the new value of parameter is used. Thus, the := form actually assigns a value if the *parameter* does not already have one. In this case, the *parameter* must be a variable; it cannot be a positional parameter.

${*parameter*:?*message*}

is related to the previous two forms. If the given *parameter* is not defined or is null, *message* is displayed. If the construct is being used inside a shell script, the script terminates with an error status. For example, you might have

```
cp $1 ${2:?"Must specify a directory name"}
```

In this case, the message following the ? is displayed if there is no second positional parameter. If you omit the *message*, the shell displays a standard message.

For example, you can just say

```
cp $1 ${2:?}
```

to get the standard error message.

The construct

${*parameter*:+*replacement*}

might be thought of as the opposite of the preceding expansions. If *parameter* has not been assigned a value, or has a null value, this construct is just the null string. If *parameter* does have a non-null value, the value is ignored and the *replacement* value is used in its place. Thus if a shell script contains

```
echo ${1:+"There was a parameter"}
```

the **echo** command displays

```
There was a parameter
```

if the script was invoked with a parameter. If no parameter was specified, the **echo** command has nothing to echo.

String Matching Modifiers

Another set of parameter modifiers use wild card expressions to discard part of the parameter value.

`${`*parameter#pattern*`}`

is evaluated by expanding the value of *parameter* and then deleting the smallest left most part of the expansion that matches the given *pattern* of path name wild card characters. For example, suppose that the variable *`NAME`* stands for a file name. You might use

```
${NAME#?:}
```

to remove the device part of the name. If

```
NAME="C:dir/subdir/file.c"
```

then

```
${NAME#?:}
```

expands to

```
dir/subdir/file.c
```

Just as the previous construct removes the smallest left most part of the parameter,

`${`*parameter##pattern*`}`

removes the largest left most part that matches the pattern. For example,

```
${NAME##*/}
```

yields

```
file.c
```

The wild card character `*` stands for any sequence of characters. In this situation, it stands for everything up to the final slash.

The construct

`${`*parameter%pattern*`}`

removes the smallest right most part of the parameter expansion that matches *pattern*. Thus

```
${NAME%.?}
```

stands for

```
C:dir/subdir/file
```

Similarly,

```
${parameter%%pattern}
```

stands for the expansion of *parameter* without the longest rightmost string that matches pattern. For example,

```
${NAME%%/*}
```

stands for

```
C:dir
```

As all these examples suggest, these parameter modifiers are intended to let you break off parts of file names. For example,

```
for $i in *.*
do
    mv $i ${i%.*}.txt
done
```

obtains the names of all the files in the current directory and renames them to have the suffix `.txt`.

Special Parameters

The MKS KornShell has a variety of special parameters that may be used in shell scripts and command lines.

`$@` stands for the complete list of positional parameters, each separated by a single space. For example,

```
echo $@
```

If the positional parameters are all file names,

```
cp $@ dir
```

copies all the files to the given directory *dir*.

`$*` stands for the complete list of positional parameters, each separated by the first character of the value of the shell variable *`IFS`*. For example, with

```
IFS=,$IFS
```

then

```
echo "$*"
```

displays the parameters with separating commas.

`$#` is the number of positional parameters passed to this shell script. This number can be changed by several MKS

KornShell commands (for example, **`set`** or **`shift`**).

`$?` is the exit status value returned by the most recently run command.

`$-` stands for the set of options that have been specified for this shell session. This includes options that were specified on the command line that invoked the shell, plus other options that have been set via the **`set`** command.

Graphical Features of the MKS KornShell 10

Along with all of the standard KornShell functionality, the MKS KornShell provides additional features that allow users to interact with and manipulate the Windows environment. With the power and ease of these features, you can just point and click to perform many of the most common Toolkit functions.

This chapter touches upon many facets of the Windows specific Kornshell commands. It is not, however, a comprehensive guide to these commands. For complete information, see the online *PTC MKS Toolkit Utilities Reference*.

Whether you are a novice or a seasoned shell programmer, you'll find that the graphical capabilities of the MKS KornShell will make your programs easier to write and more user friendly. With commands such as **`msgbox`**, **`filebox`**, **`start`**, **`dlg`**, and **`bindres`**, your KornShell scripts can be used to augment or replace large programs written in far more complex languages such as C or C++.

Managing the KornShell

The graphical nature of Windows makes it easy to determine how your KornShell looks and works. No longer do you need to search for the exact syntax of a command to make the shell function the way you want; most commonly used shell settings and environment variables are now available in easy to use graphical dialogs.

gvar

Under Windows 95/98/Me, you can invoke **`gvar`** from the toolbar by clicking

Use the **`gvar`** command to set environment variables. Click on the property sheets to view and/or edit:

- **Variables**

  View all environment variables, or only those that are read-only or exported. You can also add, delete, or modify any shell environment variable from this dialog box.

- **Positional Parameters**

  Display all currently set positional parameters.

- **Functions**

  Display a list of all currently set functions.

- **Aliases**

  Display the name and expanded value of all current shell system aliases. You can also add, delete, or modify any shell alias.

gset

The **gset** command lets you set many of the common **set** options, as well as define other aspects of how the shell functions. The **KornShell Options** dialog box displayed by **gset** offers the following property sheets where you can view and/or edit settings:

Under Windows 95/98/Me, you can invoke **gset** from the toolbar by clicking

- **User**

  Determine how the shell manages output redirection, end-of-files, and file name generation.

- **Environment**

  Define whether environment variables are automatically exported, whether they can be included on the command line, and whether unset parameters can be used in substitution.

- **Edit**

  Choose between **vi**, **emacs**, or standard windows editing modes.

- **Messages**

  Set Verbose Echo, Execute Trace, and specify whether errors are returned on the command line or in a message box.

- **Prompt**

  Define how your primary and secondary command line prompts appear.

- **History**

  Set your history file and the number of lines it will contain.

- **Persistence**

  Save the current option settings for future use, and set whether the saved option settings are used on shell startup.

Desktop Tools

The MKS KornShell includes graphical utilities that let you display and edit much of the most useful file, user, and system information.

autorun, **gdir**, **ghist**, **gps**, and **ugrep** are shell scripts which incorporate the **dlg** command, and can be examined as examples of advanced **dlg** shell scripting.

autorun Sets the Windows registry database to run any command on startup.

gdir Displays a list of recently visited directories, inserts a directory name into the stack (like the command line **pushd**), or goes to any directory in the stack (like the command line **popd**).

ghist Displays a graphical history of your previous shell command lines. To rerun any command in the history, scroll to the command you want and double click. To edit a command, click it once to display it in the **Current Line** field, make any changes, then click **OK** to run it.

You can invoke **ghist** by hitting `ESC-g` in **vi** or **emacs** command line editing mode, entering **ghist** on the command line, or from the toolbar under Windows 95/98/Me.

gps Displays a scrollable list of all processes currently running on your system.

ugrep Searches files using regular expressions with the capabilities of the command line **grep**, **egrep**, and **fgrep**.

Using Graphical Features in Shell Scripts

With the graphical features of the PTC MKS Toolkit, shell scripts can take on a new level of sophistication. In this section we refine and expand a simple KornShell script by including Windows specific commands.

To show how these commands can improve your shell scripts, we'll start with the `listit.ksh` script described in "The if Conditional" on page 36 and show how it can be improved by adding calls to

msgbox Displays a message box to prompt users and take input.

start Starts a specified program in a new shell window.

filebox Displays a dialog box to search for and open files.

dlg Loads and manipulates Windows dialog boxes.

To refresh your memory, `listit.ksh` is a simple shell script intended to give you as much information as possible about any *name* entered on the command line as an argument. The original script was:

```
if test ! "$1"
then
   echo "no positional parameters"
elif test -d $1
then
   echo "$1 is a directory"
   lc $1
elif test -f $1
then
   echo "$1 is a file"
   more $1
else
   echo "$1 is just a string"
fi
```

msgbox

msgbox informs users of important actions or conditions, and accepts feedback on those conditions. Messages displayed by **msgbox** can include predefined icons and buttons, as well as large amounts of message text. Adding **msgbox** functionality to a KornShell command line script transforms it into an interactive Windows program.

For complete details on the options and syntax of **msgbox**, see the **msgbox** reference page in the online *PTC MKS Toolkit Utilities Reference*.

With the original `listit.ksh`, if you enter the command

```
listit abcdef
```

and `abcdef` is neither a file nor a directory, the shell script returns:

```
abcdef is just a string
```

If we replace the command

```
echo "$1 is just a string"
```

with the **msgbox** command

```
msgbox -fqb ok -i information listit.ksh "$1 is just a
string. Please enter a directory or file name after
listit."
```

the shell script now returns something far more effective:

The following code sample show how you could modify listit.ksh to return a message box wherever it previously returned a command line string.

```
#Define button return values
OK=1
Cancel=2
Yes=6
No=7

#Test to ensure command line parameters are entered
if test ! "$1"
then
   msgbox -fqb ok -i exclamation listit.ksh "You must
enter a positional parameter, otherwise listit.ksh
will fail. Try again."

#Test to check if parameter is a directory
elif test -d $1
then
   msgbox -fqb okcancel -d1 -i information listit.ksh
"$1 is a directory. Do you want a listing printed to
your screen?"
   case $? in
   $OK) lc $1 ;;
   $Cancel) exit ;;
   esac

#Test to check if parameter is a file
elif test -f $1
then
   msgbox -fqb okcancel -d1 -i information listit.ksh
"$1 is a file. Do you want to view it on your screen
now?"
   case $? in
   $OK) more $1 ;;
   $Cancel) exit ;;
   esac
else
```

```
   msgbox -fqb ok -i information listit.ksh "$1 is
just a string. Please enter a directory or file name
after listit."
fi
```

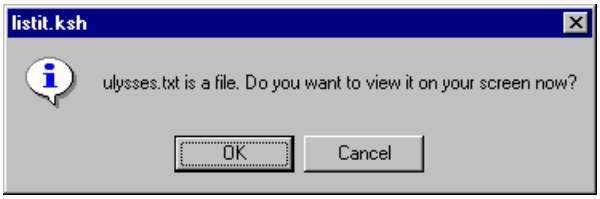
start

It's equally easy to take advantage of Windows' multi-windowing features by including a call to the **start** command in your shell script. **start** creates a new window and performs a specified action in the new window before exiting—see the **start** reference page for a complete description of option and syntax.

Let's look at what happens in the previous example of listit.ksh when the *name* that you enter as an argument is a file. For example, if you enter

```
listit ulysses.txt
```

and `ulysses.txt` resides in the current directory, the script returns



If you click on **OK**, listit.ksh invokes the **more** command to display the file, one page at a time. To let users edit ulysses.txt if they want, instead of simply viewing it, you could replace the call to **more** with a call to **start** a text editor in a new window with `ulysses.txt` loaded and ready to edit. For example, instead of

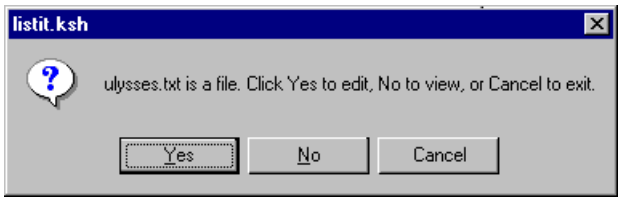
```
then
   msgbox -fqb okcancel -d1 -i question listit.ksh
"$1 is a file. Do you want to view it on your screen
now?"
   case $? in
      $OK) more $1 ;;
      $Cancel) exit ;;
      *) print "You must select either OK or Cancel."
   esac
else
```

you could try something like this:

```
then
   msgbox -fqb yesnocancel -d1 -i question listit.ksh
"$1 is a file. Click Yes to edit, No to View, or
Cancel to exit."
```

```
   case $? in
      $Cancel) break ;;
      $Yes) export EDITOR=${EDITOR:=viw}
   start -x -t "This is a listit.ksh text editing
windows." $EDITOR $1 ;;
      $No) more $1 ;;
   esac
else
```

`listit.ksh` will now return.



filebox

Now that you can pop up graphical messages and start separate windows and programs from your KornShell script, it's time to try giving the user another option if the *name* entered on the command line isn't a file or directory. Currently, if you enter the command

```
listit abcdef
```

you are presented with a message box telling you that `abcdef` is just a string, and that you should enter a directory or file name after `listit`. The program then terminates, leaving you on your own to locate the file name that you want.

For more information, see the **`filebox`** reference page in the online *PTC MKS Toolkit Utilities Reference*.

With **`filebox`** you can offer users a choice. If the name that was entered is not valid, a call to **`filebox`** invokes a file browser offering users the opportunity to select a file from anywhere on their system. For example, you could replace the previous **`else`** construct that prompts for a name to be entered on the command line with a call to **`filebox`**.

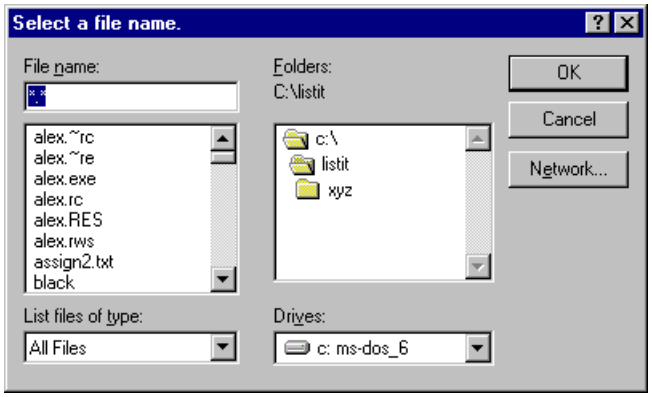
```
else
   msgbox -fqb yesno -d1 -i question listit.ksh “$1
is just a string. Would you like to browse for a file
now?”
   case $? in

   $Yes) filename=$(filebox -amnt “Listit: Select a
file name.”)
      if [ $? -eq $Cancel ]
      then
         exit
```

```
        fi
msgbox -fqb yesno -d1 -i question listit.ksh “Would
you like to edit $filename now?”
case $? in
$Yes) start -x -t “This is a listit.ksh text editing
window.” viw $filename ;;
$No) exit ;;
esac ;;
$No) exit ;;
*) print “You must select a file name.”
esac
fi
```

In this case, when a you enter

```
listit abcdef
```

`listit.ksh` returns a message box telling you that `abcedf` is just a sting and asking if you want to browse for a file. If you select **No** the program terminates as before. If you select **Yes**, however, a file browser appears



After you have selected a file, you are again presented with a message box offering the choice of editing the file or exiting the program.

The next code sample is an example of how an improved `listit.ksh` might incorporate the new calls to **`msgbox`**, **`start`**, and **`filebox`**:

```
#Define Button return values
OK=1
Cancel=2
Yes=6
No=7

#Test to ensure command line parameters are entered
if test ! "$1"
then
   msgbox -fqb ok -i exclamation listit.ksh "You must
enter an argument, otherwise listit.ksh will fail. Try
again."

#Test to check if parameter is a directory
elif test -d $1
then
   msgbox -fqb yesno -d1 -i information listit.ksh
"$1 is a directory. Do you want a listing printed to
your screen?"
   case $? in
   $Yes) dir=$(lc $1)
      msgbox -fq "Your directory contains:" "$dir" ;;
   $No) exit ;;
   *)msgbox -fb ok Listit "You must select either Yes
or No."
   esac

#Test to check if parameter is a file; use a call to
#start to allow editing
elif test -f $1
then
   msgbox -fqb yesno -d1 -i information listit.ksh
"$1 is a file. Would you like to edit it with Vi for
Windows?"
   case $? in
$Yes) start -x -t "This is a listit.ksh text editing
window." viw $1 ;;
   $No) exit ;;
   *) msgbox -fqb ok Listit "You must select either
Yes to edit $1 or No to exit."
   esac
#If parameter is neither a file nor directory, use a
#call to filebox to allow file selection
else
```

```
   msgbox -fqb yesno -d1 -i question listit.ksh "$1
is just a string. Would you like to browse for a file
now?"
   case $? in
   $Yes) filename=$(filebox -amnt "Listit:  Select a
file name.")
   if [ $? -eq $Cancel ]
   then
      exit
   fi
   msgbox -fqb yesno -d1 -i question listit.ksh
"Would you like to edit $filename now?"
      case $? in
      $Yes) start -x -t "This is a listit.ksh text
editing window." viw $filename ;;
      $No) exit ;;
      *) msgbox -fqb ok Listit "You must select a
file name."
      esac ;;
   $No) exit
esac
fi
```

dlg

With the graphical features of the MKS KornShell your shell scripts are no longer limited to the command line. Now you can use commands like **msgbox**, **start**, and **filebox** to can make your shell scripts look and act like programs written in far more complex languages.

dlg takes you to the next level of interactive graphical KornShell scripting. **dlg** (for "dialog") is an MKS KornShell function that lets you read and modify Windows dialog boxes. Whether you are designing new dialogs or manipulating existing ones, **dlg** is an effective alternative to the often complex Windows programming languages.

Basic Structure of dlg Shell Scripts

dlg shell scripts usually follow a structure similar to

```
#Load the dialog
dlg load module dialogid

#Start a loop to read events from the dialog
while dlg event msg ctrl
do

#Read events, manage controls, get or change text
dlg keyword [-d dlg] [-c control] [-i index] [result]
```

```
#Terminate the loop and close the dialog
done
dlg close [-d dlg]
```

While there are many variations, the basic structure can be found in most **dlg** scripts. You need to **load** a dialog box before you can manage it, read the output from **event** to determine user actions, and **close** the dialog box when you're done.

dlg Examples

There are many useful **dlg** commands that this document does not touch upon; see the **dlg** reference page in the online *PTC MKS Toolkit Utilities Reference* for complete details on **dlg** commands.

The examples in this section use some of the basic **dlg** commands to generate a simple Windows program from `listit.ksh`.

list	**settext**	**enabled**	**load**
event	**gettext**	**clear**	**addtext**
getcursel	**winhelp**	**close**	

For examples of more complex shell scripts that employ many other **dlg** commands, examine the code in **autorun.ksh**, **gdir.ksh**, **ghist.ksh**, **gps.ksh**, or **ugrep.ksh** installed in your *ROOTDIR*/mksnt directory.

Since you need access to a dialog editor to create new dialogs (and not everyone has a programs like Microsoft C++ that includes a dialog editor), the following example works under the premise that you will be modifying an existing dialog. Sophisticated programmers, however, will probably want to create their own dialog boxes to meet the explicit needs of each particular **dlg** shell script; if you have created your own dialog you may want to skip the preliminary sections of this tutorial and start with "dlg event" on page 73.

dlg list

To turn `listit.ksh` into a Windows program we'll use one of the sample dialogs located in your *ROOTDIR*/samples/dlg directory. The file `sample1.res` contains four generic dialogs which you can use to get a feel for **dlg**.

To start with, we must determine what `sample1.res` contains. To display the contents of any 32-bit file containing standard dialogs, use the **dlg list** command. For example,

```
dlg list sample1.res
```

will display

```
COMBO_SAMPLE
EDIT_SAMPLE
LIST_SAMPLE
```

```
LISTVIEW_SAMPLE
```

Note Dialogs are often named with a number rather than a text string, but in this case we wanted to make identification as easy as possible.

To display the internal components of a specific dialog, use **`list`** again but this time include the dialog name. The command

```
dlg list sample1.res LIST_SAMPLE
```

will disassemble `LIST_SAMPLE` into a resource compiler format describing all controls, their control number, and their position within the dialog box. Most of the information returned from **`dlg list`** is of little use unless you are coding and linking the dialog yourself, but it does tell you exactly what controls are present. If you create your own dialog with a dialog editor, you won't need to use the information from **`dlg list`**.If you load the sample dialog with the **`dlg load`** command, you will see something similar to

The dialog box contains five button controls (labeled **OK**, **Cancel**, and **Button**), as well as a radio button, a check box and a combo box control.

dlg settext

Obviously, you can't use this dialog "as is"; the generic control labels give no indication of what function each control performs. First we need to determine just what listit.ksh should do. At minimum it should do everything that the command line version of `listit.ksh` does:

- Let users select a file from anywhere on their system
- Display information about the current directory
- Let users edit the file if they want

To make the dialog "user friendly" you will need to alter the text on some of the buttons with the **`dlg settext`** command. **`settext`** changes existing dialog text to the *string* you specify.You could start the new `listit.ksh` with something like

```
#Define your controls
Select=1001
Edit=1002
DirList=1003
Help=106
Cancel=105
List=103

   #Load the dialog and center it on the screen
   dlg load -x -1 -y -1 sample1.res LIST_SAMPLE

   #Change the title text on the dialog
   dlg settext "Listit: A Small Sample of dlg Shell
Scripting"

   #Change the text on buttons
   dlg settext -c $Select "Select File"
   dlg settext -c $Edit "Edit File"
   dlg settext -c $DirList "Directory Listing"
   dlg settext -c $Help "Help"
```

Note Each dialog must be "loaded" before you can use it in any way. If you try to use **`dlg`** commands without first loading the dialog, an error will occur.

Notice how **`dlg settext`** without a **`-c`** *control* option changes the title text of the dialog box, whereas if you specify a *control* **`settext`** operates only on that control.

`dlg settext` works equally well for static text fields and other titles within the loaded dialog. The next step is to change the static text title from `Static Text Box` to something more descriptive:

```
#Change the static text into a title
dlg settext -c 1004 “Selected Files”
```

dlg enabled

In the next phase of our dialog box face-lift, we'll "gray out" the unused check box and radio button. **`dlg enabled`** allows you to remove or reinstate the functionality of any given control in a loaded dialog box. For our purposes, we'll simply gray out the checkbox and radio button and leave them that way.

```
# "Gray out" unnecessary radio button and check box,
# and change their text labels to tell users that
# they are unavailable
dlg settext -c 102 "Not available"
dlg enabled -c 102 0
dlg settext -c 105 "Not available"
dlg enabled -c 105 0
```

If you load the `LIST_SAMPLE` dialog box with the **`settext`** and **`enabled`** changes, it will now look more the way we want.

dlg event

Now that the dialog box appears as it should, we need to associate the proper action with each control. The first thing is to make sure that any user actions are properly read into the shell script. **`dlg event`** returns both the control identifier and the associated message for each user action. For example,

```
while dlg event msg ctrl
do
       echo $msg $ctrl
done
```

would print a message and control number to standard output for each user action. If the user clicks on **Select File**, the shell script would print

```
command 1001
```

For more details on **`while`** loops, see "The while Loop" on page 38.

The return values from **`dlg event`** are what allows your program to interpret user actions. Unless your program is in a **`while`** loop reading the results of **`dlg event`**, user actions will not be recognized.

dlg gettext

One of the fundamental functions of listit.ksh is to let users select a file from anywhere on their system. To accomplish this with **`dlg`**, you could replace the **`while`** loop with a **`gettext`** call to read the contents of the list box control.

In the next example code, **`dlg gettext`** is used to read the contents of the list box control. You can also use **`gettext`** to return the associated text string from any control in a loaded dialog. This example

- Tests the output of **`gettext`** to determine if there is already text in the list box
- Uses **`dlg clear`** to remove the contents
- Presents a file browser to select new files

Once files have been selected, **`dlg addtext`** is used to add the selected file names to the contents of the list box—the `while read filename` construct ensures that file names are added one line at a time rather than strung together.

```
#Start while loop
while dlg event msg ctrl
do
   #Read the ctrl results of dlg event
   case $ctrl in
   $Select)
      dlg gettext -c $List seltext
      if test -n "$seltext"
      then
         dlg clear -c $List
         files_selected=0
         f=$(filebox -amnt "Listit")
         while read filename
         do
            dlg addtext -c $List $filename
            files_selected=1
         done <<EOF
$f
EOF
         if [ files_selected = 0 ]
         then
         msgbox -fq Listit "No files selected"
         fi
      else
         files_selected=0
         f=$(filebox -amnt "Listit")
         while read filename
         do
            dlg addtext -c $List $filename
            files_selected=1
         done <<EOF
$f
EOF
         if [ files_selected = 0 ]
         then
            msgbox -fq Listit "No files selected"
         fi
      fi ;;
   esac
done
```

dlg getcursel

Once users can select files, we need to let them use the **Edit Files** button. The **`dlg getcursel`** command lets you determine which item in the list box control is selected at any given time. **`getcursel`** returns an integer representing the list position of current selection, or -1 if no item is selected.

In the next example, the output from **`getcursel`** is stored in the variable `cursel`, and tested to ensure a selection has been made.

```
$Edit) dlg getcursel -c $List cursel
if test "$cursel" -lt 0
then
    msgbox -fq -i information Listit "No files selected"
else
    dlg gettext -c $List -i $cursel seltext
    export EDITOR=${EDITOR:=viw}
    start -x $EDITOR "$seltext"
fi
```

dlg winhelp

The remaining button controls (**Help**, **Directory Listing**, and **Cancel**) can be activated with relatively simple shell constructs. Use the **`dlg winhelp`** command to make the **Help** button display a Windows Help file. **`winhelp`** is a special **`dlg`** command which allows you to call a help file that operates independently of the current shell, so that you can invoke Windows Help and leave it displayed while you continue to use other commands.

You can specify a **`dlg winhelp -C`** *context-id* option to open the Windows Help file at a predetermined location. For this example, however, we simply open viw.hlp to the default Contents screen.

```
case $ctrl in
$Help) dlg winhelp $ROOTDIR/mksnt/viw.hlp
esac
```

To display a directory listing use the same construct shown in the previous listit.ksh

```
$case $ctrl in
DirList) dir=$(lc)
    msgbox -fq “Your current directory contains:” “$dir”
esac
```

dlg close

Finally, to close the dialog box users can click **Cancel**. You must explicitly terminate the dialog with a **dlg close** command.

```
    case $ctrl in
    $Cancel) break
    esac
done
dlg close
```

The next code sample is an example of what our program looks like with all changes in place. This file is available as `listit.ksh` in the *ROOTDIR*`/samples/dlg` directory.

Note The `ResFile=$(whence $0)`specification is an example of how to call a resource that is contained in the current KornShell script. The encoded resource file, however, has been omitted from this code.

For details of how to bind a resource file into your shell script, see "bindres" on page 78 or the online **bindres** reference page.

```
#Define the controls
Select=1001
Edit=1002
DirList=1003
Help=106
Cancel=107
List=103

#Tell dlg that the .res file is located in the current
#file
ResFile=$(whence $0)

#Load and center the dialog
dlg load -x -1 -y -1 $ResFile LIST_SAMPLE

#Change the title text on the dialog
dlg settext "Listit: A Small Sample of dlg Shell
Scripting"

#Change the text on buttons
dlg settext -c $Select "Select File"
dlg settext -c $Edit "Edit File"
dlg settext -c $DirList "Directory Listing"
dlg settext -c $Help "Help"

#Change the static text above the listbox control into
#a title
dlg settext -c 1004 "Selected Files"
```

```
#"Gray out" the unnecessary controls and change their
#text to tell users they're unavailable
dlg enabled -c 102 0
dlg settext -c 102 "Not available"
dlg enabled -c 105 0
dlg settext -c 105 "Not available"

#Start the "while" loop to read user action
while dlg event msg ctrl
do
case $ctrl in
       $Select)
           dlg gettext -c $List seltext
           if
              test -n "$seltext"
           then
           dlg clear -c $List
              files_selected=0
              f=$(filebox -amnt "Listit")
              while read filename
              do
                 dlg addtext -c $List $filename
                 files_selected=1
              done <<EOF
$f
EOF
              if [ files_selected = 0 ]
              then
                 msgbox -x "No files selected"
              fi
           else
           files_selected=0
           f=$(filebox -amnt "Listit")
           while read filename
           do
              dlg addtext -c $List $filename
              files_selected=1
           done <<EOF
$f
EOF
              if [ files_selected = 0 ]
              then
                 msgbox -fq -i information Listit "No
files selected"
              fi
           fi ;;
       $Edit) dlg getcursel -c $List cursel
```

```
        if test "$cursel" -lt 0
           then
           msgbox -fq -i information Listit "No files
selected"
        else
           dlg gettext -c $List -i $cursel seltext
           export EDITOR=${EDITOR:=viw}
           start -x $EDITOR "$seltext"
           fi ;;
        $DirList) dir=$(lc)
           msgbox -fq "Your directory contains:"
"$dir" ;;
        $Help) dlg winhelp $ROOTDIR/mksnt/toolkit.hlp
;;
        $Cancel) break ; dlg close
esac
done
```

bindres

To convert your shell script to a self contained executable, use the **bindres** program to "bind" the resource from sample1.res into `listit.ksh`. Once your dialog resource is bound into the KornShell file, the executable can be transferred and run on any system with PTC MKS Toolkit installed—without the necessity of any related files.

bindres can be used to incorporate any `.bmp`, `.cur`, `.ico`, or `.res` file into any text file. When a resource file is bound into another file with **bindres**, the file is encoded and appended to the end of the existing code. To bind `sample1.res` into `listit.ksh`, use the command

```
bindres listit.ksh sample1.res
```

Note that **bindres** encodes the entire `sample1.res` file and inserts it at the end of `listit.ksh`, so you end up with all four dialogs from `sample1.res` in your KornShell file. You can either leave the extraneous dialog code in your script (it won't hurt anything, but does tend to make the script appear more complex than necessary), or simply delete it.

If you use **bindres** on a file that already contains a bound resource file, the new output of **bindres** is inserted between the original code and the previous encoded resource file. For example, if you add the `sh.ico` icon into `listit.ksh` so that you can invoke it from the Windows Explorer, the encoded `sh.ico` resource file would be inserted after the shell script code but before the previous encoded bitmap.

To include the encoded KornShell icon in listit.ksh, use the command

```
bindres listit.ksh $ROOTDIR/mksnt/sh.ico
```

toolbar

For more information on adding bitmaps and toolbar buttons from the command line, see the **tb** reference page in the online *PTC MKS Toolkit Utilities Reference*.

Under Windows 95/98/Me, you have the added capability of controlling your KornShell toolbar with the **tb** command.

To add a button to the Windows 95/98/Me KornShell console toolbar, you can either use one of the standard bitmaps already available to you, or create your own bitmap. If you create your own bitmap you must add it to the available pool of bitmaps.

```
tb bitmap 16x16.bmp 24x24.bmp i
```

To use one of the standard Windows toolbar bitmaps for your program, double click the KornShell toolbar. A **Customize Toolbar** dialog box appears displaying the available buttons and the current non-standard toolbar buttons. Double click any available button to invoke the **Customize Button** dialog box, where you can enter the command name and associated tool tip text.

To create your own bitmaps for new buttons on the toolbar, use a drawing program like Windows' **Paint**, and save the file as both a 16x16 bitmap and a 24x24 bitmap.

If you are currently running a program in the shell, any program invoked by clicking a toolbar button will not be run until the current program is finished.

Index

Symbols

A

B

C

D

E

Q

R

S

T

U

V

W