"The Software Tools"
Unix Capabilities on Non-Unix Systems

Deborah K. Scherrer, Philip H. Scherrer, Thomas H. Strong, & Samuel J. Penny

BYTE Magazine, November 1983, p. 430

(Retyped by Emmanuel ROCHE.)

1 The Software Tools package

The Software Tools package is a set of programs and subroutines that provides the power and elegance of Bell Laboratories' Unix on non-Unix computer systems. The tools offer Unix-like program development features that complement systems ranging from microcomputers to mainframes.

Available in various forms from several sources, the Software Tools package includes more than 60 utility programs, a command interpreter (*shell*), and a large programming library.

Code sharing, coupled with early feedback from users, has allowed developers to build on each other's work and has produced a dynamic environment in which new ideas are rapidly tried and proven. The natural selection process that results produces high-quality, useful utilities that have been tried, improved, tested, and accepted by many users with varying needs and a variety of systems.

The Tools

The Software Tools utilities provide a framework for executing most common computing tasks. Each tool is a powerful but general software module designed to do one thing well.

The tools are easy to learn and use. They perform functions such as organizing and manipulating files, creating, editing, and rearranging text, examining files, preparing documents, and transforming language and data. Frequently used tools are:

- diff Determines the differences between 2 files
- ls Lists the file names in a directory
- ar Maintains multiple small files nested inside a larger one

- sort   Sorts lines of a text file in several ways
- find   Locates text patterns in a file using a flexible expression syntax
- field   Rearranges data columns in a file
- sedit   Performs serial editing functions on a file
- format Formats a document for publication or distribution

The complete set of Software Tools provides most of the functional capabilities of the Unix tools. Table 1 is a list of the tools and their Unix equivalents.

Table 1: The Software Tools and their Unix equivalents.

Text Manipulation

Software Tool	Unix Utility	Description
e, edin	ed	Editor
sedit	sed	Stream editor
ch	gres	Change text patterns
tr	tr	Transliterate characters
find	grep	Locate text patterns
fb		Find text patterns in blocks of lines
isam		Build index sequential access list
xref		Cross-reference of symbols
field		Manipulate fields of data
mcol	pr -n	Produce multicolumn output
sort	sort	Sort lines
lam		Laminate lines of files together
uniq	uniq	Strip duplicate lines
rev	rev	Reverse order of characters
number		Number lines
detab		Convert tabs to spaces
entab		Convert spaces to tabs
crypt	crypt	Crypt and decrypt files
cpress		Compress files
expand		Expand compressed files
os		Convert backspaces for printing
	col	Convert reverse line feeds for printing
pl		Print specific lines in file
	awk	Pattern scanning and processing language
	join	Join lines with identical fields
	prep	Put words on single lines

Manipulating Files

cat	cat	Concatenate/copy files
crt		Paginate files to terminal
cp	cp	Copy files
pr	pr	Paginate files for printing
show		Show all characters (control too)
tail	tail	Print last lines of files
tee	tee	Copy input to output and named files
includ		Include files within files
split	split	Split up file
cmp	cmp	Simple file compare
diff	diff	Differential file compare
	diff3	3-way differential file compare
comm	comm	Print lines common to 2 files
ll		Print longest, shortest line lengths
wc	wc	Count words, characters, lines
	dd	Convert and copy a file

Managing Files and Directories

ls	ls	List files
cd	cd	Change directory
pwd	pwd	Print working directory name
mv	mv	Move/rename file
rm	rm	Remove files
ar	ar	Archive files
n.a.	chown, chgrp	Change owner/group of files
n.a.	chmod	Change mode of file
	find	Search for files
	ln	Link files
	mkdir	Make a directory
	rmdir	Remove a directory
	sum	Validate a file (checksum)
	tar, tp	Tape archiver
	touch	Update last-change-date
	file	Determine file type

Document Preparation

format	roff, nroff	Text formatter

	troff	Text formatter for typesetter
form		Form letter generator
spell	spell	Spelling checker
lookup	look	Look up words in dictionary
kwic, unrot	ptx	Generate permuted index
	deroff	Remove nroff commands
	eqn	Generate equations for nroff
	tbl	Generate tables for nroff
	refer	Find and insert literature references
	pubindex	Make index for "refer"
	tc	Translate troff output for Tektronix 4015

Process Control

sh	sh	Command-line interpreter (shell)
run		Run a tool (without shell)
which		Print full pathname of command
reset		Reset system after media change
logout	logout	Log out of shell
n.a.	at	Run process at specific time
n.a.	login	Log into system
n.a.	kill	Kill (background) process
n.a.	nice	Run process at low priority
n.a.	ps	Process status
n.a.	sleep	Suspend termination for specified period
n.a.	wait	Wait for completion of a process
	time	Time a process
	prof	Display profile data

User Support/Information Retrieval

dc	dc	Desk calculator
date	date	Print/set time and date
echo	echo	Print command-line arguments
man	man	Print manual entry
n.a.	passwd	Set/change password
n.a.	tty	Get terminal name
n.a.	who	List users on system
	true, false	Commands which return true or false
	basename	Print basename of file
	cal	Print calendar
	calendar	Remind user of appointments

	expr	Evaluate arguments as an expression
	factor	Factor a number
	test	Condition command
	units	Quantity conversions

Language Translation/Program Development

macro	m4	Macro processor
ratfor	ratfor	RATFOR preprocessor
fsort		Sort FORTRAN declarations
rc	rc	RATFOR, FORTRAN, link, load
fc	fc	FORTRAN, link, load
ld	ld	Load
tsort	tsort	Topological sort
yacc	yacc	Compiler-compiler
lex	lex	Lexical analyzer
	adb	Debugger
	as	Assembler
	bas	BASIC interpreter
	bc	Arbitrary-precision arithmetic language
	cc, pcc	C compile
	lint	C syntax check
	F77	FORTRAN compile
	struct	Convert FORTRAN-66 to RATFOR
	lorder	Find ordering relation for library
	nm	Print name list of object files
	od	Octal dump
	size	Print size of object file
	strip	Remove symbols and relocation bits
	ranlib	Convert archives to random libraries

Miscellaneous

	graph	Draw a graph
	plot	Graphics filter
	spline	Interpolate smooth curve
	tk	Paginate for the Tektronix 4014
n.a.	write	Send message to another user
n.a.	mesg	Permit or deny messages
tcs	sccs	Test maintenance system
msg	mail	Send/receive mail
	learn	Computer-aided instruction about Unix

lpr	Print spooler
make	Maintain program groups
cu	Call another Unix machine
uucp	Unix-to-Unix copy
uux	Unix-to-Unix command execution
stty	Set terminal options
tabs	Set terminal tabs

Key:
n.a: -- not applicable to single user/single process systems like CP/M.
The capabilities of a Software Tool and a Unix utility may not always be exactly the same.

The Shell

The Software Tools shell is a command interpreter that reads lines from the user terminal or a file and interprets them as requests to execute programs. The shell includes mechanisms to redirect the input and output of the tools to the user terminal, files, or other programs. It also enables the user to group commands together to make up new commands. The ease of generating and executing complex user-tailored commands from simple ones distinguishes Unix and the Software Tools from other systems in which utilities are often clumsy. The Section "2 Software Tools Shell" describes the shell in greater detail.

The Library

The Software Tools library provides a framework for accessing system services by both the tools and user programs. The library includes basic system operations as well as groups of functions satisfying common programming needs. These include:

- Unix-type I/O (input/output) functions
- file and directory manipulation
- dynamic memory allocation
- string manipulation
- linked-list handling
- symbol-table creation
- text-pattern matching
- data-type conversion and manipulation
- date and time formatting
- command-line argument handling
- process control

Table 2 describes the library functions in detail.

Table 2: The functions of the Software Tools library.

Symbol Definitions (ratdef)
----

definitions	Standard RATFOR definitions

File Manipulation
----

*amove	Move (rename) a file
*close	Close (detach) a file
*create	Create a new file (or overwrite an existing one)
*gettyp	Get type of file (character or binary)
*isatty	Determine if a file is a terminal
*mkuniq	Generate unique file name
*open	Open an existing file for reading, writing, or both
*remove	Remove a file from the file system

I/O
---

fcopy	Copy one file to another
*flush	Flush output buffer for file
getc	Read character from standard input
*getch	Read character from file
*getlin	Read next line from file
*note	Determine current file position
*prompt	Prompt user for input
putc	Write character to standard output
*putch	Write character to file
putdec	Write integer in field
putint	Write integer in field on file
*putlin	Output a line onto file
putstr	Write string in field on file
*readf	Binary read from a file
*remark	Print single-line message
*seek	Move read/write pointer
*setmod	Set character device mode
*writef	Binary write to a file

Process Control

----

*endst	Close all open files and terminate program execution
*exec	Execute task
*initst	Initialize all standard files and common variables

Directory Manipulation
----

*closdr	Close directory
*cwdir	Change working directory
*gdraux	Get auxiliary directory information
*gdrprm	Get next directory entry
*gwdir	Get name of current working directory
*opendr	Open directory for reading

String Manipulation
----

addset	Add character to array if it fits, increment pointer
addstr	Add string to array if it fits, increment pointer
concat	Concatenate 2 strings together
ctoc	Copy string-to-string
equal	Compare str1 to str2
gettok	Parse tokens
getwrd	Get non-blank word from array, increment pointer
index	Find character in string
length	Compute length of string
scopy	Copy string from one array to another
sdrop	Drop characters from a string
skipbl	Skip blanks and tabs in array
sktok	Skip over tokens
slstr	Slice (take) a substring from a string
stake	Take characters from a string
stcopy	Copy string, increment pointer
stncmp	Compare first n characters of strings
stncpy	Copy n characters from one array to another
strcmp	Compare 2 strings
strim	Trim trailing blanks and tabs from a string
type	Determine type of character

Character Conversion
----

clower	Convert character to lower case
ctoi	Convert string to integer, increment pointer
ctomn	Translate ASCII control character to mnemonic
cupper	Convert character to upper case
esc	Check for escaped character
fold	Convert string to lower case
gctoi	Generalized character-to-integer conversion
gitoc	Generalized integer-to-character conversion
itoc	Convert integer to character string
lower	Convert string to lower case
mntoc	Convert ASCII mnemonic to character
upper	Convert string to upper case

Pattern Matching
----

amatch	Look for pattern matching regular expression
getpat	Encode regular expression for pattern matching
makpat	Encode regular expression for pattern matching
match	Match pattern anywhere on line

Command Line Handling
----

*delarg	Delete a command-line argument
*getarg	Get command-line arguments
gfnarg	Get next filename argument
query	Print command usage information

Dynamic Storage Allocation
----

*dsfree	Free a block of dynamic storage
*dsget	Obtain a block of dynamic storage
*dsinit	Initialize dynamic storage

Symbol Table Manipulation
----

delete	Remove a symbol from symbol table
enter	Place symbol in symbol table

lookup	Get string associated with symbol from hash table
mktabl	Make a symbol table
rmtabl	Remove a symbol table
sctabl	Scan all symbols in a symbol table

Linked List / Stack Handling

maklst	Create and initialize linked list
frelst	Remove a linked list and free allocated memory
push	Push an item onto the top of the list/stack
pop	Pop an item from the top of the list/stack
inject	Inject a new item into a linked list
xtract	Read an item from a linked list
prvnod	Get previous node pointer
nxtnod	Get next node pointer
remod	Remove a node from a linked list

Date Manipulation

atodat	Convert ASCII characters to integer date
fmtdat	Convert date to character string
*getnow	Get current date and time
wkday	Get day-of-week corresponding to month-day-year

Error Handling

cant	Print "name: can't open" and terminate execution
error	Print single-line message and terminate execution

(* indicates that the routine is system-dependent and has been implemented by Carousel Microtools for CP/M and MS-DOS.)

The Tools or Unix?

Although the Software Tools provide many of the features of Unix, they are not an exact copy of Unix. They exist alongside the local operating system and provide many of the desirable aspects of Unix in situations where using Unix is impossible or inappropriate. For instance, if you do not want to pay Unix's high price, if you want to use software packages that are not available in Unix versions, or if a Unix implementation is not available for your hardware, the Software Tools can provide the power and elegance of the Unix interface.

Let us look at the Software tools movement and considerations that have made the tools successful.

The Software Tools Movement

In 1976, Kernighan and Plauger wrote *Software Tools* (see Reference 3). Their goal was to teach good programming style based on their experiences with Unix at Bell Laboratories. They used pared-down versions of Unix Utilities rewritten in RATFOR (Rational FORTRAN), a C-like preprocessor language (see Section "3 What Is RATFOR?"). The programs and the RATFOR preprocessor were made available on magnetic tape. The book and tape were the seeds from which the tools movement developed. The movement arose independently at several major research laboratories and universities.

The tools were of immediate interest to researchers and users, and the programs were implemented on numerous computers. As users began to experiment with and enhance the programs, they began to realize that the tools offered more than a useful set of utility programs. Researchers, primarily at Lawrence Berkeley Laboratory (LBL), expanded the original package to include a powerful subroutine library, a Unix-like shell, and many more of the Unix utilities. By providing all 3 levels (shell, utilities, and library), the tools now offered a portable, uniform interface with the functionality of Unix. The package was implemented on the diverse assortment of LBL machines and on many machines to which the researchers had network access. The result was Unix functionality on non-Unix systems and a consistent user interface across many different systems (see Reference 1).

One reason the Software Tools have been so widely accepted is their portability. The tools can be implemented on virtually any machine. This portability was achieved by using a programming language that was available on all machines and by isolating system dependencies into "primitive" function calls that must be implemented separately for each different system.

With certain data-type manipulation conventions and other programming details, this portability has enabled the package to be implemented on more than 50 operating systems. Table 3 provides a partial list of manufacturers offering computers on which the tools have been implemented.

Table 3: A partial list of manufacturers on whose machines the Software Tools package has been implemented to varying degrees of sophistication.

ACOS
Amdahl

Apollo
AN/UYK
Burroughs
CDC
Cray
Data General
DEC
FACOM
GEC
HP
HITAC
Honeywell
IBM
Intel
Interdata
Modcomp
Multics
NCR
Perkin-Elmer
Prime
Rolm
SEL
Tandem
Univac
Wang
Xerox
Machines running CP/M
Machines running MS-DOS
Machines running Unix

Which Language Is Best?

Computer languages are judged on their ability to solve specific problems; therefore, the best language for the Software Tools package was the one that could most adequately fill the following requirements:

- Availability - The language had to be available on almost every machine.

- Suitability - The language had to be appropriate for textual (as opposed to numerical) applications; it had to be powerful enough to handle the support libraries that provide the necessary file access, I/O process control, and other system-support services.

- Quality - The language had to be high-level, easy to read and understand, easy to learn, and powerful enough to solve applications problems.

FORTRAN filled the first requirement, fell down a bit on the second, and provided little of the third. C met the second and third requirements but was not usually available on both microcomputers and larger machines. Pascal met the third requirement but was no more commonly available than C and was not appropriate to the support of large libraries and moderately complex bodies of code (see Reference 2). Several other state-of-the-art languages were appealing but not generally available. Thus, no single language met all the requirements, and a compromise was necessary. The RATFOR language preprocessor was chosen because it provided the control structures, readability, and elegance of C and was translatable into FORTRAN (the language available on most systems). A C-like support library was developed to supplant FORTRAN'S incomplete textual, file manipulation, and I/O capabilities. Even Though FORTRAN is used at the RATTOR base level, the user is insulated from FORTRAN just as the user of any high-level language is insulated from the machine language.

The choice of language was not critical to the approach. In fact, for the person using the tools, the implementation language is unimportant. Only the tools implementer and people developing new tools with the library ever need to use the language. Had the tools been designed solely for the microcomputer environment, C might have been a more appropriate choice. With the computer industry rapidly developing new machines and more elegant languages, the Software Tools community is now re-evaluating the original choice of language and considering mechanisms for making the tools available in other languages as well.

Primitives Isolate Machine Dependencies

In the Software Tools package, system dependencies are isolated in the primitives, a set of routines that make up the tools' interface to the operating system. The primitives provide standardized system services such as file manipulation, I/O, process control, and dynamic memory allocation. The tools and their subroutines access system services through these primitives. Tool source code can be moved from system to system without change. When the tools package is moved to a new system, only the primitives must be changed or rewritten.

The original implementers of the tools issued 2 prime directives to assure compatibility among a wide variety of operating systems. First, they decided to use the file types of the operating system. Internal file formats specific to the machine are hidden from the user by the primitive functions, allowing both local utilities and Software Tools programs to read and write the same files and providing a standardized way to access files on all systems. Second, changes to the local system, or interference with it to implement the package, are discouraged. Such changes, combined with the local system's idiosyncrasies, would make the package unstable in new system releases.

The primitives address the issue of machine efficiency; they minimize the demands of the software upon scarce system resources like memory or central processor time. For example, the utilities of the Software Tools package are oriented toward text processing and program development (writing source code, documentation, data preparation, etc.). These utilities are characteristically limited by I/O rates. Because the I/O capabilities are isolated in the primitives, the effect of this problem can be reduced through efficient implementation of the I/O primitives. Because all utilities access resources through the primitives, they automatically benefit from such optimization.

The Software Tools Users Group (STUG)

The need for cooperation among implementers and users of the tools led to the formation of the Software Tools Users Group at Menlo Park, California. It originated at the Lawrence Berkeley Laboratory and was initially funded by the Department of Energy. Since its inception in 1978, the group has become an international body performing the following functions:

- Establishing and publishing standards for the primitives and tools and supporting an ongoing standards committee
- Collecting and distributing information on current developments to avoid duplication of effort
- Collecting and evaluating new utilities, extensions, and variants
- Holding semi-annual meetings in conjunction with the Usenix Unix users group
- Publishing a newsletter and software catalog
- Distributing tapes containing collections of utilities from different organizations

Much of the tools' source code is now in the public domain and freely distributed. The primitives, however, are generally developed, licensed, and maintained by vendors.

The standardization procedure used by the tools group is unusual. New utilities are collected and distributed early in their development phase, allowing users to experiment with new ideas and reject those that prove unportable or functionally undesirable. Code sharing also allows users and developers to glean ideas from new offerings and incorporate them into their own developments. As ideas are distilled and utilities enhanced or extended, the utilities are redistributed, and those receiving popular support are eventually returned to the tools group. There, they pass to the Implementers Committee, which makes final decisions on acceptance and standardization. Thus, standards are always based on ideas or utilities tested and proven by the community, rather than on newly-designed products or untested ideas.

The sharing of code and feedback from users enables developers of new tools to build on each other's work, creating an environment in which new ideas can be quickly and thoroughly tested. The sharing results in natural selection of useful tools that have been tried and accepted by a large number of users with varying needs on many different systems.

The Present and the Future

Development of the Software Tools is proceeding on 2 fronts: the basic package is being implemented on new systems, and user interfaces are being extended. The original package provided an environment for effective development of programs and manipulation of textual data and materials. However, the tools approach is applicable to most software projects, including those involving networks, database management, graphics, and word processing. Among the portable packages being developed are experimental shells, statistical analysis systems, electronic-mail systems, screen editors, data-management packages, data-analysis packages, and source-code-maintenance systems. The tools group is actively evaluating suggested enhancements and extending the primitive set to provide as dynamic and creative an environment as possible.

Some hardware manufacturers avoid the Software Tools package because easy portability to a competitor's hardware is obviously bad for business. Increasingly, however, independent companies are marketing specific system implementations of the tools. These firms typically implement the primitives and provide maintenance and upgrade support. The high-level source code (utilities and portable sections of the library) is left unlicensed, so the Software Tools Users Group handles variations, extensions, and standards (a compromise between the need for vendor support and the desire for user control).

The Software Tools package is already running on most mini-computer and mainframe systems, and extensions into the microcomputer world have begun.

Implementing the Tools

Writing programs in a language that is available on many systems is insufficient; you must also define an interface layer that isolates an application program from the details of any particular system. The primitives form the tools' interface layer and are the key to their success. They are the only allowed connection between the tools and the underlying operating system. Porting, or adapting, the tools to a new operating system involves writing the code for the primitives for that new system.

The primitives are more than just a collection of subroutines; they provide a complete environment for the tools. In a sense, they coordinate the "world view" of the tools with the world view of the host operating system. The task is simple if the tools and the new system have similar views of the programmer's environment; the task is difficult if the new system has a different view. For example, it took less than a week to write and test the tools' primitives for Unix because Unix's view of the programmer's environment is similar to that of the tools. But implementing the tools' primitives on

CP/M and MS-DOS (which are based on very different views) took more than a year.

When implementing the primitives, it is essential to keep in mind the 2 prime directives: maintain correspondence of file types and avoid interfering with or changing the host system. An example of the relationship between the tools and the host system is illustrated in the implementation of the Carousel Toolkits on CP/M (see Figure 1).

```
          HIERARCHY OF PRIMITIVES

      ┌──────────────────────□□
□□───────────────□□
□□─────────□
────□
──
□─
□□

      │           SHELL
□□
      ├- - - - - - - - - - - - -┬─────□
────□
──
□─
□□

      │ PORTABLE SOFTWARE TOOLS │
 □
      │ UTILITY PROGRAMS        │
 □
      ├- - - - -┬- - - - - - - -┤
  □
      │ PORTABLE |          │ NON-TOOL
 □
      │ UTILITY  |          │ APPLICATI
 □
      │ LIBRARY  |          │ PROGRAMS
 □
      ├- - - - -┘           │
  □
```

```
│                    │
│ PRIMITIVE LIBRARY  │
│                    │
├────────────────────
│        BDOS
├────────────────────
│        BIOS
├────────────────────
│     HARDWARE I/O
└────────────────────
```

Figure 1: The hierarchical dependence of interfaces in the CP/M-80 version of the tools. At the CP/M level, only the BIOS (basic input/output system) knows how to do direct hardware input and output, and only the BDOS (basic disk operating system) knows how to talk to the BIOS. These clean divisions were the key to the early success in moving CP/M to many different types of hardware. The Software Tools are built in isolated layers in the same way. Note that only the primitive functions know how to talk to the BDOS. The primitives are the communication channel between the portable tools and a specific operating system, such as CP/M or MS-DOS. The tools themselves can use the primitives or the library of utility routines that are also part of the tools package. The clean boundaries between the various interface layers in a system such as this are very important for maintaining clean portable programs. Any time these separations are violated, the resulting program may prove expensive to maintain and difficult to move to new machines.

File and Directory Names

The Software Tools view all I/O operations as actions on named files. As in Unix, use of files from within programs must be as device independent as possible because the program does not know whether the I/O is being done with a terminal, file, or another program. The file to be used is specified when the program is run instead of when it is compiled. When the host provides some sort of directory structure, it should appear to the user as the Unix model of a hierarchical directory structure does. These requirements have effects at both the RATFOR library level and at the tools execution level. For example,

data	The file "data" on the current directory
/b/data	The file "data" on drive B in the current user area
/2/a/data	The file "data" in user area 2 on drive A
/tty	The programmer's terminal
/nul	The "bit bucket", a place for unwanted output
/lst	The printer

File names of these forms can be used anywhere a file name is needed. For example, in the tools *open* primitive, the statement

```
fd = open ("/0/c/foobar.dat", READWRITE)
```

results in the file /0/c/foobar.dat being opened in a mode allowing random reads and writes. The command

```
diff /1/b/prog.bas prog.bas
```

displays the differences between the version of prog.bas on drive B in user area 1 and the version in the current directory.

By putting CP/M's user-area number at the higher level in the hierarchy, a programmer can operate within a given area on several drives without specifying the user area. In accordance with the prime directive, a CP/M style of directory naming is also recognized (e.g., 1b:prog.bas). In addition, the temptation to further follow the Unix style and allow user-named subdirectories, as opposed to the hard-wired CP/M user/disk names, was tempered by the prime directive's requirement that all tools files be available on the host system with recognizably similar names.

Memory Allocation and Disk

The tools package includes primitives to dynamically allocate memory areas for temporary use within a program. This feature has proven easy to provide on single-user systems such as CP/M and MS-DOS, where the programmer has access to all memory not occupied by the program or operating system. However, bulk- storage I/O devices, usually floppy disks, are so slow that it is desirable to use as much high-speed memory as possible for a cache of recently-used or soon-to-be-used data. These 2 requirements force the dynamic-storage primitives for CP/M to share the memory with the I/O primitives. This provides the tools with dynamically available storage while using all remaining memory to speed up disk operations.

The Software Tools package also enables a user to quickly access the large collection of the tools' utilities on a small system. Sixty non-trivial tools could easily occupy a large amount of disk space. Unlike integrated programs in which all functions are available to the user within one large complex program, the tools are a collection of single-purpose programs, each of which must be loaded into memory when needed. To provide both fast program load times and small disk-space usage on CP/M, the tools were stored on disk as overlays of each other. Because they all share the common primitives, the primitives need be loaded into memory only once. When a tool program is run, only the part of the program that is different from one tool to another need be loaded. This has proved effective in reducing disk usage and program load time.

Process Control

The most difficult primitives to implement on single-user microcomputer operating systems are for process control. Unix views the world as process- rich -- a place in which processes are created for each command. The single- user CP/M system, on the other hand, supports only one process. To provide a Unix-like environment in this case, the primitives must emulate multiple processes. The only practical way to simulate several parallel processes on a small-memory, floppy-disk-based system is by a sequence of programs that are not executed simultaneously.

Unix enables process creation and program execution by the function pair *fork* and *exec* (see Reference 4). Fork creates a clone process and exec overlays the current process with a new program. The most common sequence in Unix is

fork - wait - continue (in the parent process)
fork - exec - die (in the child process)

The standard tools package provides a model of this sequence in the *spawn* primitive. Spawn executes a program by creating a child process and allowing the parent to wait for its completion. Because of the relatively slow, low-capacity disk storage available on the CP/M and MS-DOS systems, the spawn primitive has been simulated with a Unix-like exec. Therefore, the portable shell could not be used, and a new shell was written that uses only exec and creates a chain of programs that always end with a new invocation of itself. This new shell can also be used on other systems where process generation is allowed but is restricted or slow.

The spawn mechanism is different from those used by other command-interpreter replacements for CP/M that always expect to reside in memory. The Software Tools utilities are loaded quickly because they use the overlay technique.

Conclusion

The Software Tools package provides the features of Unix when Unix is not desirable, available, or appropriate. The tools incorporate many of the features of Unix: elegance achieved through simplicity of style, consistency of use, modularity, and a common-sense approach to programming tasks. A large and active Software Tools Users Group has brought these tools to most operating systems.

Software Tools packages are available from several sources. A source code for the utilities and specifications for the primitives is available from the Software Tools Users Group (STUG) for a nominal charge. If you choose to purchase this code, you must write your own primitives, which may be difficult.

You may be able to obtain a complete tools implementation for your system from someone who has already done it for a similar system. The tools group distributes versions for a few mini-computers and mainframe systems. These are provided without support.

You may also purchase specific implementations of the Software Tools from a vendor. If you do so, you should expect a version of the primitives optimized for your system, with continuing support and contact with the Software Tools Users Group.

2 Software Tools Shell

(Carousel Microtool's CP/M Implementation)

The shell is a command-line interpreter; it reads lines from the terminal or a file and interprets them as requests to execute other programs.

Commands

In its simplest form, a command is the file name of a program to be run, followed by arguments given to the program. The command name may specify any file in the system. CP/M enables a user number to be part of the command (file) name. The command may be a Software Tool or any other program. The shell searches for the named file in a series of directories specified by the user in an environment file. When the command is located, it is loaded into memory and executed. When the command is finished, the shell resumes its own execution. For example, giving the command

```
sort file1 file2
```

causes the shell to locate and execute the command *sort*. Sort, in turn, merges and sorts the contents of the 2 named files and puts the output on the user's terminal.

I/O Redirection

Software Tools programs have 3 files automatically available to the user:

1) standard input
2) standard output
3) standard error output

All 3 are assigned to the user's terminal, unless specifically redirected to disk files or other devices. Redirection is specified by preceding the desired device or file name with a special character:

< file	Read standard input from "file"
> file	Send standard output to "file"
? file	Send standard error output to "file"
>> file	Append standard output to "file"
?? file	Append standard error output to "file"

In the above example, the sorted output could be saved on a file:

```
sort file1 file2 > sorted
```

or sent to the printer:

```
sort file1 file2 > /lst
```

(/lst is the tools form of the name for the printer).

I/O redirection is actually performed by each tool individually, rather than by the shell.

Pipes

A sequence of commands separated by vertical bars (|) causes the shell to execute each command in sequence and arranges to have the standard output of each command delivered as the standard input to the next command in the sequence. The sequence:

```
sort list | uniq | crt
```

sorts the contents of file *list*. The sorted output passes to *uniq*, which removes extra copies of duplicated lines. This output then goes to *crt*, which paginates output for viewing on a terminal.

Command Separators

Commands need not be on different lines; instead, they may be separated by semicolons:

```
ar -x program rtn ; e rtn
```

extracts the member *rtn* from the archive file *program* and then enters the editor.

Background Processes

Unix shells enable processes to be started and have control returned immediately to the shell. The new process continues running in the background, sharing resources with the shell process. This mechanism is impossible to implement on single-process systems such as those using CP/M. However, to simulate the mechanism in some reasonable way, the Carousel shell saves any commands indicated as background processes and executes them at the end of the session, when the user logs out of the shell For example,

```
format doc > /lst &
```

formats the file *doc* and sends it to the printer at the end of the session (the ampersand (&) indicates a background process).

Script Files

The real power of the Unix and Software Tools shells comes from the ability to generate new commands by combining existing commands. This feature is possible because the shell not only executes programs, but also treats script files (text files containing yet more commands) as commands. These scripts may participate in pipelines, have their I/O redirected, and appear in any context that a regular command may. Scripts may be nested by referencing scripts that may, in turn, reference other scripts.

Scripts are useful for creating new commands and for grouping commands together for multiple re-execution. For example, you could create a standard procedure by editing file *fix* to fill it with the following commands for the shell:

```
ar -x book chap1
e chap1
format chap1 | crt
ar -u book chap1
```

Then, by typing *fix*, the system would extract *chap1* from the archived file *book*; edit *chap1*; send *chap1* to the formatter and display it page-by-page on the terminal; and finally update it in the archive file *book*.

Arguments can also be passed to script files. Character sequences of the form $n, where n is between 1 and 9, are replaced by the nth argument to the invocation of the script. If *book* has more than one section, the script could be written:

```
ar -x book $1
e $1
format $1 | crt
ar -u book $1
```

Then you could type:

  fix chap1
or *fix chap7*
or *fix intro*

to edit, view, and update the respective sections of *book*.

Script files can include inline explicit data that the tools can read as their standard input. The special input redirection notation << is used to achieve this effect. For example, the editor takes its commands from standard input, normally the terminal. However, within a shell script, commands may also be embedded this way:

```
e file <<!
(editing requests)
!
```

(The ! is arbitrary; any character can be used.) The lines between <<! and ! are called, in Unix terminology, a "here document"; they are read by the shell and made available to the command as its standard input.

Finally, as an indication of the power of script files, Listing 1 shows an example of a script file to show changes that have been made to command files of dBASE-II, a database management program.

Listing 1: The alterations to dBASE-II command files.

```
# Shell command file to show work done to dBASE-II command files.
# usage: dbdiff dir (where dir is a backup directory)
# "dir" should be specified in tools form, e.g. "/2/B"
# dbdiff will print all new dBASE command files and
# will print existing dBASE command files with any
# changes marked with a "|" in the right margin.

# Collect names of .cmd files in both directories.
ls .cmd >1.tmp
ls $1 .cmd >2.tmp

# Find and print new dBASE commands.

# Here, comm reports lines in 1.tmp which are not present in 2.tmp;
# field changes that report into a series of print commands;
# and sh then executes those print commands.
# The "@" signs suppress the following newline,
# effectively continuing the shell command across several lines.
comm -1 1.tmp 2.tmp | @
 field "pr >/lst $1" | @
 sh

# Find existing dBASE commands and show changes.

# Here comm reports files listed in both 1.tmp and 2.tmp;
# e (the editor) changes each file name reported by comm
# into a series of commands to:
#   print the file name;
#   print the current date & time;
#   print the differences between the versions
#   in this directory and in the other directory;
# and cat puts a few formatter commands into 4.tmp,
# to be called upon by each line of 3.tmp.
comm -3 1.tmp 2.tmp >3.tmp
e 3.tmp <<!
1,$s~?*~echo & >/lst ; date >/lst ;
   diff -r $1/& & | format 4.tmp - >/lst~
w
q
!

cat >4.tmp <<!
 .nf
 .in 5     (ROCHE> WordStar does not like "dot commands"...)
 .rm 70
!

# Finally, the shell runs the commands that e just prepared
# and rm removes all 3 scratch files.
sh 3.tmp $1
rm 1.tmp 2.tmp 3.tmp
```

Environments

Like Unix, The Carousel shell maintains an environment file. This file contains information about the user's system and needs, such as the date, tab settings, and the directories in which to search for user programs or tools. The environment file is available to all tools and is modified by a few. In addition, users are free to adjust the information for their own needs.

Control Structures

Constructs of the nature:

```
if ... then ... else ...
while ... do ...
for ... in ... do ...
```

aid in re-iteration and conditional execution within scripts. The Software

Tools Users Group is currently standardizing the syntax for these shell control structures.

3 What is RATFOR?

RATFOR (Rational FORTRAN) is the implementation language for the Software Tools. It is closely patterned after C in its control structures, but it is compiled into FORTRAN by the RATFOR preprocessor. The availability of FORTRAN allows RATFOR to be easily installed on a wide variety of systems. In addition to being a portable language suitable for implementing the Software Tools, RATFOR is a convenient language for program development. The control constructs of RATFOR are those of C, and the data structures are those of FORTRAN.

RATFOR's nature can most easily be described with examples of some actual code. A file of standard definitions is automatically processed by the RATFOR compiler to define new symbolic constants. A section of this file is:

```
define (EOF, -1)
define (EOS, 0)
define (MAXLINE, 128)
define (STDIN, 1)
define (STDOUT, 2)
define (character, integer)
```

Using these definitions, the following code is an example of a program in RATFOR that finds the length of the longest line read from standard input:

```
DRIVER
character line (MAXLINE)
integer getlin, length, len, size
size = 0
while (getlin (line, STDIN) != EOF)
  {
  len = length (line)
  if (len > size)
    size = len
  }
call putint (size, 5, STDOUT)
call putch (NEWLINE, STDOUT)
DRETURN
end
```

The macros DRIVER and DRETURN are also defined in the standard definition file and are used to start and end all RATFOR programs.

The following code is the same program written in C:

```
#include <stdio.h>
#defined (MAXLINE, 128)

main()
{
char line[MAXLINE];
int fgets(), strlen(), size = 0, len;
while (fgets(line, MAXLINE, stdin))
  {
  len = strlen(line);
  if (len > size)
     size = len;
  }
fprint(stdout, "%5d\n", size);
}
```

The similarity between the RATFOR and C versions is obvious. Notice that the RATFOR example consists almost entirely of standard FORTRAN statements especially assignment statements and subroutine calls. The RATFOR compiler passes these statements through to the FORTRAN version almost unchanged. What RATFOR adds to FORTRAN are file inclusion, token substitution, macros for text replacement, and the following control constructs:

if-else for conditional execution,
while, for, and *repeat-until* for looping,
break and *next* for controlling loop exits,
switch-case-default for selection of alternatives,
braces ({}) for statement grouping.

RATFOR's syntax was intended to liberalize FORTRAN's syntax restrictions as much as possible. As a result, RATFOR source code is naturally *concise* and reasonably pleasing to the eye. RATFOR features are as follows:

- free-form page layout
- unobstrusive comments
- use of <, <=, >, >=, ==, !=, etc. for comparison expressions
- *string* data type
- quoted character strings and character constants
- *define* statement for symbolic constants
- *include* statement for source-file inclusion
- *macro preprocessor* for textual manipulation

RATFOR code is often easier to read and understand than the corresponding section of code as normally written in C. For example, the 2 following fragments of code each copy a string from one buffer to another:

```
# RATFOR version

for (i=1; from(i) != EOS; i=i+1)
   to(i) = from(i)
to(i) = EOS

/* C version */

char *t=to, *f=from;
while (*t++ = *f++);
```

One could argue that a good C compiler sometimes produces faster code but, in large programs, the readability of the RATFOR style is often an advantage over the more terse C style.

4 References

----

1. Hall, Dennis, Deborah Scherrer & Joe Sventek
"A Virtual Operating System"
Communication of the Association of Computing Machinery,
Volume 23, Number 9, Pages 495-502.

2. Kernighan, Brian W.
"Why Pascal Is Not My Favorite Language"
Bell Laboratories Technical Note #100, Murray Hill, NJ.

3. Kernighan, Brian W. & P. J. Plauger,
"Software Tools"
North Reading, MA: Addison-Wesley, 1976.

4. The Bell System Technical Journal,
Volume 57, Number 6, Part 2 (July-August 1978).