;login:

The USENIX Association Newsletter

Volume 14, Number 1 January/February 1989

CONTENTS

Call for Papers: Workshop on Software Management ........................................... 3
Call for Papers: Workshop on UNIX Transaction Processing ............................. 4
Call for Papers: Summer 1989 USENIX Conference ........................................... 5
Enhancing the 4.3 BSD UNIX Serial Line Interface ........................................... 6
An Update on UNIX Standards Activities ....................................................... 30
Letter to the Editor ......................................................................................... 44
The EUUG ....................................................................................................... 45
EUUG Spring '89 Conference ........................................................................... 47
Future Events ................................................................................................... 48
Publications Available ...................................................................................... 48
Long-Term Calendar of UNIX Events .............................................................. 49
Large Systems Administration Workshop ......................................................... 50
New Release of 2.10 BSD Available ................................................................. 50
1989 USENIX Membership Form ..................................................................... 51

The closing date for submissions for the next issue of ;login: is February 24, 1989

If you have not paid your 1989 membership dues, this is your last issue of ;login:. Use the form on the inside back cover (p. 51).
**NOTICE**

;login: is the official newsletter of the USENIX Association, and is sent free of charge to all members of the Association.

The USENIX Association is an organization of AT&T licensees, sub-licensees, and other persons formed for the purpose of exchanging information and ideas about UNIX and similar operating systems and the C programming language. It is a non-profit corporation incorporated under the laws of the State of Delaware. The officers of the Association are:

President

Alan G. Nemeth
usenix%nemeth

Vice-President

Deborah K. Scherrer
usenix%scherrer

Secretary

Rob Kolstad
usenix%kolstad

Treasurer

Stephen C. Johnson
usenix%scj

Directors

M. Kirk McKusick
usenix%mckusick
Sharon Murrell
usenix%murrell
Michael D. O'Dell
usenix%mo
dle
John S. Quarterman
usenix%fsq

Executive Director

Peter H. Salus
usenix%salus

Deputy Executive Director

Ellie Young
usenix%ellie

Membership Secretary

Eeva McFeely

USENIX Association Office

P.O. Box 2299
Berkeley, CA 94710
(415) 528-8649
(ucb,ucb,vax,usenix)%office

The editorial staff of ;login: is:

Editor & Publisher

Peter H. Salus

Copy Editor

Michelle Dominijanni
(masscomp,usenix)%mmp

Production Editor

Tom Strong
usenix%strong

Judith F. DesHarnais Conference Coordinator

USENIX Conference Office
P.O. Box 385
Sunset Beach, CA 90742
(213) 592-3243 or 592-1381
(ucb,vax,usenix)%judy

John L. Donnelly Tutorial & Exhibit Manager

USENIX Exhibit Office
5398 Manhattan Circle
Boulder, CO 80303
(303) 499-2600
(ucb,vax,usenix)%johnd

Contributions Solicited

Members of the UNIX community are encouraged to contribute articles to ;login:. Contributions may be sent to the editors electronically at the addresses above or through the U.S. mail to the Association office. The USENIX Association reserves the right to edit submitted material.

;login: is produced on UNIX systems using troff and a variation of the –me macros. Contributions should be in n/troff input format, using any macro package.

**UUNET Subscriptions**

UUNET Communications
P.O. Box 2685
Fairfax, VA 22031
(703) 764-9789
(ucb,vax,usenix)%uunet%request

**Acknowledgments**

The Association uses a SUN 3/180S running SUN OS for support of office and membership functions, preparation of ;login:, and other Association activities. Connected to the SUN is a QMS Lasergrafix 800 Printer System donated by Quality Micro Systems of Mobile, Alabama. It is used for general printing and draft production of ;login: and Computing Systems with ditroff software provided by mt Xinu. The membership and mailing lists are maintained using Sybase.

This newsletter is for the use of the members of the USENIX Association. Any reproduction of this newsletter in its entirety or in part requires written permission of the Association and the author(s).
Call for Papers

Workshop on Software Management

New Orleans Hilton and Towers
New Orleans, LA

April 3-4, 1989

David Tilbrook and Barry Shein will be chairing a workshop in New Orleans, LA on Monday and Tuesday, April 3-4, 1989. The workshop will concern the management and processing of source, and the discipline of managing, maintaining, and distributing software. The ultimate objective of software management is the unremarkable and painless installation of software and its subsequent upgrades at a remote site. The objective of the workshop is to present, discuss, and increase awareness of the issues involved with software management, in order to improve and facilitate the distribution and sharing of source throughout the UNIX community. Possible topics include:

- Release engineering
- Configuration management
- Installation tools and techniques
- Construction tools and techniques
- Source code control systems
- Testing

The workshop will include full length papers, short presentations, and a panel discussion on tools (e.g., is PCTE a good or viable idea?).

Among the speakers already scheduled are Vic Stenning (keynote), Steve Bourne, Andrew Hume, Kirk McKusick, and Evi Nemeth.

Abstracts of 350-700 words in PostScript or troff format should be submitted to software@usenix, by January 25, 1989. Full papers will be required by March 2, 1989. Authors will be notified by February 6, 1989 or at the San Diego Conference.
Call for Papers

Workshop on UNIX Transaction Processing

Pittsburgh Hilton Hotel
Pittsburgh, PA
May 1-2, 1989

It is expected that the UNIX System will play an increasingly important role in hosting production transaction processing systems. This first transaction processing workshop will explore existing technology applicable to UNIX-based transaction processing, and hopefully generate technical discussion on future requirements. The intent is to have short papers and presentations which include (but are not limited to) the following topics:

- Transaction Integrity
- Two-Phase Commit
- Distributed Transactions
- Client-Server Transaction models
- Transaction queuing and scheduling
- Data Entry Systems
- Transaction Benchmarking
- Transaction system performance modelling
- Operating System Support for Transaction systems

The workshop will focus on short papers and presentations. Please send electronically or on paper a one to two page single-spaced summary describing your paper or presentation to Doug Kevorkian by February 1, 1989. All submissions will be acknowledged, and authors will be notified of acceptance by March 15, 1989.

For further details about the workshop, contact the program chair:

Doug Kevorkian
AT&T Bell Laboratories
Room 5-340
190 River Road
Summit, New Jersey 07901

(201) 522-5086 (voice)
(201) 522-6621 (FAX)
attunix@dek
Call for Papers
Summer 1989 USENIX Conference

Baltimore, Maryland
June 12-16, 1989

Papers in all areas of UNIX-related research and development are solicited for formal review for the technical program of the 1989 Summer USENIX Conference. Accepted papers will be presented during the three days of technical sessions at the conference and published in the conference proceedings. The technical program is considered the leading forum for the presentation of new developments in work related to or based on the UNIX operating system.

Appropriate topics for technical presentations include, but are not limited to:

Performance:
Kernel enhancements
Compute and file servers
Scaling issues resulting from more MIPS

File systems: CD-ROM, WORM, network, archival

Networks: WAN, LAN, UUCP, OSI, distributed services

User interfaces
High reliability/availability, fault tolerance
Heterogeneous environments: mainframes, DOS/UNIX migration

Media: graphics, video, audio, art, education
System/network administration and security

Trends:
Lightweight processes
Neural networks
Object-oriented extensions

All submissions should describe new and interesting work. Like recent USENIX conferences, the Baltimore conference is requiring the submission of full papers rather than extended abstracts. The review and production cycle will not allow time for rewrite and re-review. (Time is, however, scheduled for authors of accepted papers to perform minor revisions.) Acceptance or rejection of a paper will be based solely on the work as submitted.

To be considered for the conference, a paper should include an abstract of 100 to 300 words, a discussion of how the reported results relate to other work, illustrative figures, and citations to relevant literature. The paper should present sufficient detail of the work plus appropriate background or references to enable the reviewers to perform a fair comparison with other work submitted for the conference. Full papers should be 8-12 single spaced typeset pages. All final papers must be submitted in a format suitable for camera-ready copy. For authors that do not have access to a suitable output device, facilities will be provided.

An abstract should be submitted as soon as possible. Full details and requirements will be supplied to prospective authors. Copies of the full manuscript should be submitted by ordinary and electronic mail to the Program Chair. Electronic submissions are recommended; troff -ms if possible.

Four copies and one electronic copy of each submitted paper should be received by February 8, 1989. Papers not received by this date will not be considered. Papers which clearly do not meet USENIX's standards for applicability, originality, completeness, or page length may be rejected without review. Acceptance notification will be made by March 13, 1989, and final camera-ready papers will be due by April 7, 1989.

Neil Groundwater
Baltimore USENIX Technical Program
Sun Microsystems, Inc.
8219 Leesburg Pike #700
Vienna, Virginia 22180
(703) 883-1221

Abstracts, submissions, and questions:
usenet: (uchvax,decvax,decwrl,seismo)!
sun!balt-usenix
internet: balt-usenix@sun.com
Enhancing the 4.3 BSD UNIX Serial Line Interface

Alfredo Almada and David H. Williams
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968

ABSTRACT

This paper describes simple modifications to the 4.3 BSD UNIX serial line interface that allow serial lines to be individually customized according to the devices to which they are connected. This allows nonstandard terminals such as graphics displays, and nongetty devices such as plotters to invoke hardware or software flow control, and to achieve the proper amount of I/O processing in a manner that is transparent to the user. Previously these lines suffered from a lack of I/O processing such as the inability to invoke software handshaking, and over processing such as the generation of extra characters which corrupted commands for graphics displays. The solution consisted of setting the proper combination of parameters in the provided databases gettytab and princtab, and the creation of a new database, rawtab, that was employed to initialize ports not covered by the other databases. In addition, minor modifications were made to the program getty. A complete description of the serial line interface and its internals is provided as reference for possible future updates.

1. Introduction

This work results from several years of operational experience with a DEC VAX computer running Berkeley 4.2 and later 4.3 UNIX. During this time we found that the serial line communications were quite satisfactory for standard alphanumeric terminals, and conversely, that communications for nonstandard devices were unsatisfactory and required hacks for a semblance of proper operation. Nonstandard devices in our case consisted of “nongetty” devices which did not require a login (i.e. plotters), and graphics displays which might be used for text, but mostly were employed for graphics. Most larger installations have these types of devices connected to serial lines.

Hacks included setting the baud rate manually on “nongetty” devices that did not operate at the default baud rate, ignoring flow control by using the raw interface, or working around character mappings when a terminal was under the control of getty. In some cases, application software was written to manually set the line parameters each time the program was executed and the device was opened.

In our opinion, these processes violated the tenet of UNIX that all I/O activity should be identical to the user, regardless of the file (device) that is being accessed. Furthermore, it is the duty of the operating system to make this process transparent to the user. Output redirected to nonlogged in alphanumeric terminals should exhibit <cr> <lf> mapping; output only devices should exhibit proper flow control and baud rate settings; graphics and other nonalphanumeric terminals should have certain special characters enabled (e.g. xon, xoff) and others disabled to prevent “hits” with internal commands within the terminal.

The purpose of this paper then is twofold: to describe in a detailed manner how the 4.3 BSD UNIX serial interface works, and to describe simple modifications that have been made to the operating system in order to enhance the serial line interface with nonstandard devices. The paper illustrates how the terminal interface is handled internally by the kernel, and also discusses the different system calls available to the user to
;login:

properly communicate through terminal lines. Although it describes most of the terminal driver capabilities, the reader should refer to the existing documentation (i.e. manuals) for a complete description of what is available. The specific processes apply only to 4.3 BSD UNIX executing on DEC VAX computers, however, many of the overall principles should be applicable to other versions of UNIX as well.

2. File I/O

To the UNIX user, all file I/O activity should look the same without regard to what kind of file is being employed. That is, the same system calls and processes such as redirection are invoked to access a regular file, a terminal line, a tape drive, and so on. This section deals with how the operating system makes these actions transparent to the user, and provides a description of the kernel to device driver interface and also a discussion of the system calls related to file management.

2.1 Internal Representation of Files

The kernel handles all file I/O internally through the use of inodes. Inodes exist on disk, and the kernel reads them into in-core inodes when they become active (the first instance of the file is opened) to manipulate them. The inode contains information such as owner and group identifiers, access permissions, access times, size, and type. The in-core copy of the inode contains additional information such as whether it is locked, whether someone is waiting for it to become unlocked, whether it has been modified, the inode number, and the reference count (the inode contains more information, however, only information relevant to the scope of this paper has been presented) [1].

Depending on the type of the file, the inode contains information that serves the purpose of that file. For regular files and directories, the inode contains the disk block addresses in which the data of the file are located. In the directory case, the data are the names and inode numbers of the files in the directory. For character and block device special files, the inode contains the major and minor device numbers, which uniquely identify a device. The major number distinguishes among the different types of devices, such as terminals, disk drives, tape drives, etc., and the minor number differentiates among several devices of the same type. (Two other types of inodes, symbolic links and sockets, are not discussed in this paper.)

Eventually, the last instance of the file will be closed as indicated by a reference count of zero. This will cause the inode to be written back out to disk and possibly deallocated from the in-core table, in the event that another disk inode is waiting for a free spot.

2.2 Kernel I/O Interface with Device Special Files

There are two kinds of interfaces with which the kernel communicates with external devices [2]. They are the block and character interfaces. The block interface provides a buffering mechanism, for which the algorithms of the buffer cache are invoked. The character interface is a faster raw interface which bypasses the buffer cache. The two interfaces are implemented through the use of the block device switch table (bdevsw) and the character device switch table (cdevsw) respectively. From these tables the kernel takes the entry points to the specific driver routines to be used when invoked by the different system calls.

The major number of a device, taken from the inode, is used as an index into the bdevsw or to the cdevsw depending on the type of the device. The minor number is passed to the selected routine so that it can identify the particular device. Both interfaces contain entry points for the open and close procedures. The mount and umount system calls also invoke the device open and close procedures for block devices. The read, write, and ioctl system calls for the character interface also get their entry points to the driver from the character device switch table. However, read and write system calls of block devices and of files that are on mounted file systems invoke the algorithms of the buffer cache, which invoke the device strategy procedure. This is the entry point contained in the block device switch table. The routine nulldev is used when there is no need to perform a particular driver function. However, the routine nodev is used when it should be considered an error to try to perform that driver function, such as if that device was not configured.
The cdevsw also provides entry points to routines for other more device specific tasks, such as a stop procedure for terminal multiplexers to stop transmitting on a given line, and reset routines for those devices that need to do so. There are other fields in these tables that are not relevant to this paper and therefore are not discussed. Figure 2.1 illustrates the format of the bdevsw and cdevsw tables, and the following sections describe system calls which utilize their information.

2.3 Related System Calls

2.3.1 open

A file must be opened before it can be manipulated. The user opens a file with the open system call. The syntax is as follows:

\[
i = \text{open}(\text{path}, \text{flags}, \text{inode})
\]

where path is the pathname of the file; flags is a set of actions to be taken when opening the file such as whether the file should be opened for reading, writing, whether it should be created, truncated, etc.; mode specifies the mode of the file if it is to be created; and i is the file descriptor returned by the system call [3].

The open system call allocates a file structure from the system-wide file table, and a file descriptor from the per-process u_ofile table, which is located in the u area of the process. The allocated u_ofile entry points to the allocated file structure [4]. The pathname is then parsed into an inode (namei); if this is the first instance of the file, the inode is read from disk into an in-core inode and the reference count is initialized to 1; otherwise, there is already an in-core copy of the inode and its reference count is incremented (the allocation of inodes, and the mapping from disk inodes to in-core inodes is beyond the scope of this paper) [1]. In any case, a pointer to the in-core inode is obtained and stored in the file structure. The file structure has, in addition, other fields that are initialized during the opening process. There is a set of flags saved after the open that indicates whether the file was opened for reading, writing, or appending after each write. There is also a type field which indicates that the referenced object is an actual "file" (DTYPE_INODE) and not a communications endpoint (socket). In addition, a structure within the file structure is initialized with the entry points of the general I/O routines to be invoked when the user references the file. These are read/write, ioctl, select, and close. The reference count is initialized to 1. This is a different reference count not to be confused with the inode reference count. The reference count in the file

<table>
<thead>
<tr>
<th>BLOCK DEVICE SWITCH TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHARACTER DEVICE SWITCH TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Figure 2.1: Sample of bdevsw and cdevsw
structure keeps track of how many descriptors are sharing the same instance of the file, while the inode reference count keeps track of how many instances of the file are open. Finally, the offset which determines where the next read or write will take place within the file is initialized to 0.

In the case of character and block device special files, the kernel invokes the specific open procedures to validate and initialize the driver private data structures before returning to the user. The file descriptor, which is the smallest available integer used as an index into the u_ofile table, is returned to the user. The user needs only this file descriptor to reference the file.

There is an instance of a u_ofile descriptor and a file structure entry allocated for each open call, however, there is only one inode allocated per active file in the system. See figure 2.2 for further illustration. At initialization, the kernel allocates space for its data structures. The structures relevant to file management are the process table (more specifically the u_ofile table inside the u_area of the process), the file table and the inode table. The size of these tables generally depend on the maximum number of users specified at configuration time.

2.3.2 Dup and Fork System Calls

There are two other system calls besides open that allocate a file descriptor in the u_ofile table of the process. However, in contrast to the open system call, they do not allocate a file structure but instead share one that already exists. The first of these two calls is the dup system call. The dup call is invoked as

\[ \text{newfd} = \text{dup}(i) \]

where \( i \) is a previously allocated descriptor [3]. It causes the lowest numbered descriptor available from the u_ofile table to be allocated, and the file structure being pointed to by the i \( \text{th} \) entry to now be shared between the two descriptors \( i \) and newid. Figure 2.3 illustrates the effect of a dup call. The other call is the fork system call. During a fork, the u_ofile table of the parent process is inherited by the child process. As a result, the child process now shares with its parent all the file structures that had been allocated for the parent process. See figure 2.4 for illustration.

These two system calls cause the reference count in the particular file structure(s) to be incremented. As will be seen later, the close system call checks and decrements this reference count and calls the device closing procedures only when this count reaches zero.

2.3.3 Read/Write

The read and write system calls are implemented in very much the same way. In fact, the same internal kernel routines are used for both, and a flag set by the user callable read and write routines, determines which of the two operations is to be performed. The syntax for these two system calls is

\[ i = \text{read}(fd, buf, count) \]
\[ i = \text{write}(fd, buf, count) \]

where \( fd \) is a previously allocated descriptor from one of the system calls described above, \( buf \) is the address where data is to be stored (read) or taken from (write), \( count \) is the number of characters to be transferred, and \( i \) specifies the number of characters actually transferred [3]. The kernel sets up this address and character count in the uio (user I/O) structure, in addition to some other fields that will be explained below.

As described in the open procedure, the file descriptor specified by the user is used as an index into the u_ofile table of the process to obtain the pointer to the corresponding file structure. The flags that were saved in the file structure after the open are used to check that the intended operation (read/write) corresponds to what the file was opened for. The offset that indicates where this read/write should start is also taken from the file structure and stored in the uio structure. If the file is a regular file opened in write/append mode, the offset is set to the size of the file, taken from the inode. An additional flag set in the uio structure indicates to the kernel that it should transfer the data from kernel to user space in the case of a read, or from user to kernel space in the event of a write. Once the uio structure is properly initialized, the inode read/write routine is invoked with the file table entry, the read/write flag, and the uio structure as arguments. On return, the offset is updated.
and the actual number of characters read/written is returned to the user.

The inode read/write routine checks the type of the inode and performs the appropriate task. In the case of block devices and files that are on mounted file systems (regular files, directories, and symbolic links) the routine invokes the algorithms of the buffer cache, which in turn invoke the device strategy procedure (bdevsw) [2]. The other type of inodes are character device special files. For
this type of inodes, the major and minor numbers are obtained from the inode itself. The major number indexes the cdevsw to pick the device driver entry point for a read or write operation. The minor number and the uio structure are passed as arguments to this routine. See figure 2.5 for illustration.

When a read occurs, the inode is marked as accessed so that its access time is modified. When a write occurs, the inode is marked as updated meaning that the file has been modified, and as changed meaning the the inode itself has changed. Updated and changed are two different states. When a file is updated, the inode is consequently modified. However, the inode can be modified without the file being changed, such as when changing the ownership of the file.

2.3.4 ioctl

The ioctl system call is used to customize communication parameters for devices. It is mostly intended for character devices (in particular, terminal lines) and communications endpoint types of files (sockets). Nevertheless, some limited requests are allowed on regular files and directories. Its syntax is as follows:

```c
ioctl(fd, request, argp)
```

where fd is again a previously allocated descriptor, request is the action to be performed, and argp is a pointer to the parameter list [3]. Request is an unsigned quantity four bytes long, each of which contains specific information about the request. The high order byte indicates whether the parameters pointed to by argp are in and/or out parameters, or neither. The next byte contains the size in bytes of the parameter list pointed to by argp; therefore a maximum of 127 bytes is allowed. The next byte contains the ascii code of a literal of the set {t, f, s, r, i} that identifies the class of the "file" on which this ioctl command is going to be performed; t is used for character devices (i.e. terminals), f is for regular files or directories, and s, r, and i are used for sockets. The low order byte contains a unique integer id within the class to identify the particular command.

The ioctl system call funnels through the cdevsw in the same way that the previously discussed routines do. Because of its particular correlation with terminal lines, this relationship will be deferred until the section describing the terminal I/O interface.

2.3.5 Close

Eventually, the user files are closed. Either this operation is performed by the user, or if a process still has open descriptors when it exits, all are closed automatically during the exit system call. The close system call is provided for closing files and is as follows:

```c
close(fd)
```

where fd is a previously allocated descriptor [3]. The close system call deallocates the fdth entry from the u_ofile table of the process, and decrements the reference count of the corresponding file table entry. If the reference count reaches zero, the closing procedures for this instance of the file are called. A reference count of zero in a file table entry means it is now free and can be reallocated to another descriptor. If the reference count is not zero the close system call returns immediately.

If the reference count in the file table entry indeed reached zero, it means there is now one less instance of the file open; thus the inode reference count is decremented to reflect the change. If the inode reference count is not zero, the system call returns to the user. On the other hand if it does drop to zero, the inode is written out to disk and possibly deallocated from the in-core inode table (if another disk inode is waiting for a free spot). For block and character devices the device closing procedures are invoked only at this point.
Figure 2.3: Effect of the dup system call on file tables
Figure 2.4: Effect of the fork system call on file tables
;login:

Figure 2.5: Illustration of the read and write procedures
3. Terminal I/O Interface

Terminal lines are a special case of character devices. In reality, terminal lines are usually controlled by terminal multiplexers, each of which controls several lines. The terminal I/O interface is controlled by the terminal multiplexer's driver, which in turn invokes the line discipline handling procedures for the different terminal lines. Line disciplines control the entire operation of terminal lines. They take care of opening and initializing the terminal state, performing all terminal settings, and providing a buffering mechanism appropriate for slow asynchronous communication lines, such as terminals. They also process input characters passed from the terminal multiplexer's interrupt service routine, and perform specific actions dictated by the different control characters. This part refers exclusively to the 4.3 BSD terminal line interface; however, many of the principles and examples still apply to other UNIX versions.

3.1 Terminal Multiplexers

A terminal multiplexer is the actual hardware device that controls the operation of terminal lines. When terminal multiplexers are configured into the system, the address of their control and status register and the names of the interrupt routines to call are specified in the configuration file [5]. The config program takes this information and produces a set of machine dependent routines to be invoked when the different types of interrupts occur [6]. These routines of course need to be compiled into the kernel. For a DEC VAX dmf device, the driver examines its configuration and adjusts the interrupt vectors during autoconfiguration.

The terminal multiplexer's driver entry points are located in the cdevsw entry corresponding to the major number of the device. These devices contain special registers for the hardware communications parameters of the different terminal lines. These include speed, parity, character length (8 or 7 bits) and modem control bits. There are other software communications parameters, such as control characters, terminal modes, etc., that are kept in the particular tty table entry of the terminal line. This part will refer to DEC VAX dmf 32 terminal multiplexers, each of which controls eight terminal lines; however, many of the principles also apply for other terminal multiplexers.

3.2 Line Disciplines

While terminal multiplexer drivers manage the hardware communication parameters like speed, character length, interrupt bits, and so on, they also invoke the specific line discipline procedures for the different terminal lines. Line disciplines control the operation of terminal lines. Upon opening, the line discipline open procedure establishes a control process group and a control terminal for distribution of signals. Line disciplines also handle all terminal I/O providing a buffering mechanism through the use of clists (character lists). The line discipline interfaces work in the same way that the character or block device interface work. The line discipline number is used as an index into the line switch table (linesw) from which the appropriate entry points to the tty driver are obtained. Refer to figure 3.1 for illustration.

There are two line disciplines available with the terminal interface. The first one is the old line discipline which is used with the Bourne shell. The second one is the new line discipline which has some additional job control features and must be used when using the C shell. Other disciplines may exist for special purposes, such as communications lines for network connections.

3.3 Clists

Clists provide a buffering mechanism for slow, asynchronous communication lines. A clist contains the number of characters in the list, and pointers to the first and last characters in the list. A clist is formed by a linked list of cblocks. A cblock has two fields, one of which is a pointer to the next cblock in the list, and the other one is the character array containing the characters in the clist. At initialization, the kernel allocates space for a number of cblocks and initializes the freelist to contain all of these cblocks. As input is received from the terminals, new cblocks may be allocated to the particular input clist of a terminal. As characters are read from the clist and given to the reading process(es), empty cblocks are returned to the free list. Similarly, when a process
writes to a terminal, new cblocks may be allocated for that terminal’s output clist. As characters are transmitted from the output clist to the device’s transmit buffer, empty cblocks are returned to the freelist. Figure 3.2 illustrates the process.

Every tty table entry has two input clists and one output clist associated with the particular terminal line [7]. The two input clists are called the raw queue and the canonicalized queue. Input characters are placed in the raw queue and transferred directly to the reading process as soon as they are input when in cbreak or raw mode (see below). When in crmod, the raw queue is manipulated by the line editing functions. When any of the line terminating characters is recognized, the “updated” raw queue is transferred to the canonicalized queue, which is then given to the process. Obviously, the output queue is used for characters being output to the terminal.

3.4 Terminal Modes

Terminal I/O behavior is controlled by the terminal line mode. The following are the three different modes in which a terminal may be operating:

crmod: This is the default mode. In this mode lines of input are collected and edited before making the line available to the reading process. The end of the line is recognized when either a <cr>, <nl>, EOT, or t_brkc (normally undefined) is entered. <cr> and <nl> are made synonymous in this mode and mapped to <cr><nl> on output. All driver functions, such as input editing, interrupt generation, output processing (such as delay generation and tab expansion), flow control, etc., are performed in this mode.

cbreak: In this mode characters are made available to the user as they are typed. Therefore, no input editing facilities are performed. Flow control, literal-next, interrupt processing and output processing are still done.

raw: In this mode, 8-bit characters are placed in the input/output queue without being processed. None of the control and other special characters have any meaning whatsoever in this mode. On input, characters are made available to the reading process as soon as they are entered from the keyboard.

A fourth mode, tandem, can also be used in conjunction the above modes. In tandem mode, the system generates a stop character whenever the input queue reaches its high water mark, and a start character whenever the input queue empties to its low water mark. This mode is useful primarily for the communication between two CPUs and therefore will not be discussed further.

The driver recognizes the mode of the terminal line by checking the corresponding
bit in the flags field of the particular tty table entry (refer to ioctl documentation [3] for additional information). If the raw bit is set, the other two bits are meaningless. That is, the terminal operates in raw mode as explained above. However, cmod and cbreak can be used alone or combined. If only one of the cmod or the cbreak bits is set, the terminal behaves as mentioned above. On the other hand, if both bits are set, the terminal operates as in cbreak mode, with the exception that the <cr> and <nl> characters are still mapped to <cr><lf> on output. If none of the three bits is set the terminal operates in cmod. There are other flags that alter the way the driver processes certain characters, some of which will be discussed in later sections. For a complete discussion of these flags and of the special control characters, the reader should refer to the ioctl and tty documentation [5].

3.5 Opening a Terminal

When a device is opened, the specific device open procedures are invoked as the last step in the open system call. For a terminal line, the terminal multiplexer open procedure
is invoked through the cdevsw. The minor number is passed as an argument to this routine to identify the particular terminal line. This number serves also to identify the particular multiplexer and is used as an index into its tty table. The existence of the multiplexer is checked and the tty state is initialized to the default settings. This includes initialization of both the tty entry and the terminal multiplexer’s registers. The default settings are 9600 baud, 7-bit character length and parity enabled (either). These are the actual hardware parameters kept in the terminal multiplexer’s registers. There are also software parameters such as the control characters (start/stop, interrupt, quit, etc.), all of which are initialized to their default values and stored in the tty entry. These parameters, as well as most other terminal settings, can be customized using the ioctl system call.

The open procedure waits for a carrier signal and then calls the line discipline specific open procedure. The opening process will not wait for a carrier signal if the line was not configured to support full modem control, as in the case of hardwired lines. This is indicated by the lower byte of flags in the configuration file. If a bit is turned on, that line does not support full modem control. For a dmf device, the upper six lines should be configured in this way since the dmf itself does not support full modem control for those lines.

The main role of the line discipline open procedure is to establish a process group and a control terminal for distribution of signals. The opening process (usually getty) is the control process, and the opened terminal is the control terminal. If the terminal does not have a process group associated with it (as will be the case if this is the first instance of the terminal being opened), The process group id is made to be the process id of the opening process; this id gets associated with the terminal by storing it in the tty entry of the control terminal. If the terminal had a process group already associated with it, that process group id remains the same. The resulting process group id is also stored in the process table entry corresponding to the opening process, to be inherited by all of its descendents. In this way the opening process becomes the process group leader.

The above association will only take place if the opening process does not have a process group already associated with it (as in the case of getty). This prevents a regular user process (which resulted from the user shell) to become the process group leader when it opens another terminal line. If this were not the case, user processes that open different terminal lines would have their process group id changed, and the whole purpose of the process group concept would be defeated (see below). As will be brought out later, the getty process becomes the login shell and therefore the login shell is the process group leader. It controls every process initiated at the terminal. Since the process group id is inherited through the fork system call, every descendant of the shell has the same process group id.

In particular, when interrupt and quit characters are received from the terminal, and when the user hangs up, the corresponding signal is sent to every process with the same process group id as the one associated with the terminal (gsignal). By default most of these processes exit as a result of the received signal. In this way user processes are not left around when a user suddenly hangs up the line. Nevertheless, some of these processes may have been set to ignore the hangup signal and will continue executing. To prevent these processes from receiving unwanted signals from the next terminal session, after a hangup the terminal is disassociated from the process group so that processes in that process group can no longer receive signals originating at the terminal. The new line discipline, which provides additional job control features, also distributes the stop signal to the process group when the stop character (usually ctrl-Z) is entered at the terminal.

3.6 Terminal Settings

As was mentioned in section 3.4, a terminal line behaves according to its settings. The ioctl system call is used to customize terminal lines to meet the appropriate requirements of the particular device attached to them, such as regular terminals, graphics displays, etc. The ioctl system call manipulates the relevant fields (depending on the command) in the tty entry corresponding to that terminal line.
Among other things, each tty entry contains four structures which contain settable values for each port. A description of these structures is contained in the discussion of the tty special file in reference [5]. The sgtyttyb structure contains flags for setting baud rate, terminal delays, terminal modes (RAW, CBLER, etc), echoing, and parity. The tchars structure contains the special character settings such as interrupt, quit, etc., that are defined for both the old and new terminal interfaces. The local mode word contains flags for specifying such things as the erase mode, and 7- or 8-bit character input. And finally, itchars sets special characters that are defined only for the new terminal driver. Of the four structures, the information contained in the sg_flags field of sgtyttyb and that in the local mode word is of special importance in establishing proper communication with nonstandard devices.

If in parameters are specified in the ioctl request (section 2.3.4), the kernel transfers the number of bytes specified in the request argument pointed to by argp from user to kernel space (copyin). With this data it then updates the necessary fields in the tty entry. If request is one which requires modification to the actual terminal multiplexer’s registers (i.e. the ones involving speed, parity, 8- or 7-bit character length, and modem control bits), those registers are updated also. If out parameters are specified, the kernel extracts the requested information from the tty table entry and copies it to the user area pointed to by argp (copyout). The reader should refer to the ioctl and tty documentation for a complete description of the settable parameters that are available to customize a terminal line [5].

3.7 Reading from a Terminal

When a process reads from a terminal, the terminal multiplexer read/write routine is called (again using the cdevsw) after the uio structure has been properly initialized. The routine uses the minor device number as an index into the ttytable and invokes the line discipline specific read/write procedure (figure 3.1) with the tty table entry and uio structures as arguments. This routine takes appropriate action depending on the terminal mode.

In raw mode, the input raw queue is scanned for characters to be read and if present, characters are transferred one at a time to the user address space (getc, uread, subbyte) until the characters in the queue are exhausted or the user requested amount is satisfied. Full 8-bit characters are passed. If no characters are present two actions may be taken: If the terminal was set in non-blocking mode, the read system call just returns the value zero, indicating that no characters were read and the global variable errno is set to EWOULDBLOCK indicating no input present; otherwise, the reading process is put to sleep on the event that a character is entered at the terminal (the address of that terminal’s raw queue). The process will then be awakened by the interrupt service routine when a character is received.

The same thing happens in cbreak mode, except that 7-bit characters are given to the user, unless the PASS8 flag has been specified, in which case the full eight bits are passed. This is about the only similarity between cbreak and raw mode. As stated in the terminal modes section, the amount of input processing varies greatly in the two modes. However, as will be explained shortly, this processing takes place at the interrupt service level, right when the character is received and before it is placed in the input queue to be given to the user. The exception to this is the delay suspend character, which is processed only in the new line discipline read routine, and when in cbreak or crmod. This character causes the stop signal (SIGHUP) to be sent to every process in the process group associated with the terminal, but in contrast to the suspend character, this action is delayed until a process attempts to read from the terminal.

In crmod things are done differently. First of all, the canonicalized queue is scanned for characters rather than the raw queue. As in the other two modes, if no characters are available the process will sleep waiting for a character. Again, if the terminal is in non-blocking mode, the process will not sleep but the read system call will return the value zero and errno will be set to EWOULDBLOCK. Secondly and most importantly, the sleeping process will not be awakened by the interrupt service routine as soon as any character is received, but only when a line terminating character is received. This is how lines of
input are collected (in the raw queue) and edited before making them available to reading processes by transferring them to the canonicalized queue (canq). When characters become available in the canonicalized queue, they are transferred one at a time to the user address space (getc, uread, subyte) until the line delimiter character is found or until the user specified amount is satisfied. If more characters were placed in the queue than the user requested, they will remain there for the next read operation.

As an aside, when processes are awakened, care should be taken that the conditions that put them to sleep no longer exist. This is to prevent race conditions to cause unexpected results. For example, suppose that two processes are reading from the same terminal but no input is available for either process. Furthermore, suppose the terminal is operating in raw mode. Since both processes are sleeping on the same event, both of them will be awakened when a character is received. The one that is scheduled first will get the character. However, the other one has no character to get and return to the user. Consequently, the appropriate thing for these processes to do upon awakening is to check again that a character is in fact present and if not, sleep again.

When reading from a terminal, the character count specified by the user in the read system call will not necessarily be satisfied. In most cases, the process will sleep awaiting terminal input. In raw or cbreak mode, as soon as one character is entered the process wakes up and the read system call returns with only that character being read. In crmod, the read system call returns as many characters as are found before a line delimiting character, or when the user specified amount is reached.

3.8 Writing to a Terminal

When a process writes to a terminal the line discipline write routine is invoked in the same way as the read routine. The user data, with base address and character count specified in the uio structure, are transferred from user to kernel space (uiomove, copyin). Once the data are in kernel space, they are processed according to the terminal mode and flags before they are placed in the terminal's output queue (b_to_q, putc), from which they will be transmitted to the actual device. First of all, the number of characters currently in the queue is checked. If it exceeds the high water mark, the writing process is put to sleep on the event that the output queue empties (the address of that output queue). The process will be awakened later when the character count has drained below the low water mark (the high- and low-water marks depend on the output speed of the terminal line).

If the terminal is in raw mode or if the LITOUT flag is set, all output translations are suppressed and the characters (8 bits) are placed directly in the output queue. Otherwise, 7-bit characters are used and output processing is done. If the cbreak bit is not set (the terminal is in crmod), EOT (normally "D") characters sent to that terminal are stripped off (never put in the output queue) to prevent certain terminals from hanging up. If tab expansion is specified, tabs are expanded either in cbreak or crmod and the corresponding delay is generated. The driver also provides mapping of all characters to upper case and mapping of the character "" to the character "" for terminals that require so. The LCASE and TILDE flags should be set for these terminals respectively. Finally, if the crmod bit is on, newline characters are translated to <cr><lf>. Proper delays are generated also. Some additional output translations are provided when echo received characters, such as translating the erase character into a <bs><sp><bs> sequence for crt erasing.

As will be seen later, these output translations can cause some unexpected results if the proper mode and flags are not used for the particular application. Once the characters have been processed and put on the queue, the transmitter is started to start transmitting the characters to the appropriate device. If there were any processes sleeping because the queue was full (above the high water mark), the transmitter wakes them up when the queue has drained below the low water mark.

3.9 Terminal Interrupts

When an interrupt occurs, the terminal multiplexer's interrupt service routine is invoked by picking its entry point from the particular vector address. Of course, every device (i.e. dmf) has its own set of vector
addresses. The interrupt routine sets up a parameter to be passed to the general dmf interrupt service routine, to allow it to identify the particular dmf that caused the interrupt. The dmf itself identifies which of the eight lines caused the interrupt. In the case of an input interrupt, a dmf has a silo in which it can accumulate a small number of characters to be serviced during one interrupt. Embedded in these characters is the particular line that they came from, along with status information such as whether a parity or framing error occurred, or whether the silo overflowed. In this way, the dmf interrupt service routine grabs all the characters present in the silo and passes them to the specific line discipline interrupt service routine of each terminal line.

A transition in the carrier signal interrupts in the same way as an input character. If the line was not configured for full modem control (such as hardwired lines) this transition is simply ignored. Otherwise, action is taken depending on the state of the line. If carrier is now present and the line was waiting to complete the open (such as in getty), the state of the line is set to indicate so and sleeping processes are awakened; if the line was already open and it is doing flow control depending on the carrier state (the MDMBUF flag indicates this), the transmitter on the line is restarted. If the line loses carrier two things can happen: if the loss of carrier was due to flow control, the terminal multiplexer's stop procedure is invoked for that terminal line; otherwise, the loss of carrier is assumed to be the result of the user hanging up which causes the line to be turned off and the hangup signal to be sent to the process group associated with that terminal line. If no transition in carrier is detected (either because carrier was already present or because the line is hardwired), the received characters are checked for errors before they are passed to the line discipline receiver interrupt routine. If a parity error occurred and the user indeed specified only one of odd or even parity, the character is discarded. If a framing error occurred (which might have been as a result of a BREAK character) a null is generated if the terminal was in raw mode or an interrupt character (usually ctrl-C) is generated otherwise. As will be seen later, getty switches baud rates if it gets a null or an interrupt. If a silo overflow occurs a small warning message is logged on the console. The resulting character is passed to the line discipline receiver interrupt routine to be processed.

As discussed in the terminal modes section, the amount of input processing varies according to the mode the terminal line is operating on. In raw mode input characters are put directly in the raw queue. Processes awaiting input from this terminal are awakened. In cbreak or crmod, if the PASS8 flag is not set the 8th bit (parity) is stripped off. If the terminal was in the literal-next state (as a result of the previously received character being the literal-next character), the character is not interpreted but just put in the input queue (literal-next is only implemented in the new line discipline). If the stop character is received, the terminal multiplexer's stop procedure is invoked to disable further transmission interrupts on that line. When the start character is received the transmitter is reenabled. Actually, the transmitter will be restarted when any character is received in the line. To prevent this and force the line to wait for the start character before it can be restarted, the DECCET flag must be set. When interrupt and quit characters are received the corresponding signal is sent to the process group associated with that terminal line. The new line discipline provides additional control characters such as literal-next, flush-output, and suspend characters. The literal-next character works as explained above, the flush-output character flushes any pending output to the terminal and the suspend character distributes the stop signal to the terminal's process group. None of the control characters are put in the input queue, they are simply interpreted and the necessary actions are taken.

In cbreak mode characters are given to the reading process as soon as they are received. Thus, none of the line editing characters have any meaning in cbreak mode. Consequently they are simply put in the input raw queue as if they were regular characters. Processes that had been waiting for such an event are awakened. In crmod the line editing characters cause the appropriate action to be taken. The erase character removes the previously queued character from the raw queue. The kill character causes all the characters currently in
the queue to be removed. If the CRTKILL flag is set, characters are erased from the screen with <bs><sp><bs> sequences. Otherwise a new line character is simply echoed to the terminal. The erase and kill characters can be escaped with the backslash character. In this case they have no effect and are simply put in the input queue. The new line discipline provides a word erase character and a reprint line character. The first one dequeues characters until a word delimiting character (blank or tab) is found. The latter reprints the current line of input, which is useful when such a line has been corrupted by a program outputting characters to the terminal. Normal characters are simply put in the raw queue until a line delimiting character is detected. When this happens, the updated raw queue is transferred to the canonicalized queue and processes sleeping on this event are awakened.

Characters are echoed back to the terminal unless the echo flag is turned off. Control characters are echoed differently depending on some control flags (such as CTLECH, CRTERA, etc.). There are some special characters like the start and stop characters, that are not echoed even if the echo flag is on. If they were, they could corrupt output sent, for example, to a graphics display.

3.10 Flow Control

There are two ways in which a terminal line can perform flow control. Lines with full modem control can do flow control on carrier signal, for which the MDMBUF flag should be set in the particular tty table entry. All lines can do flow control using the start (xon) and stop (xoff) characters. Most devices are manufactured so that they can either drop DTR (or one of the RS232 lines) or send an xoff character when their buffer fills up beyond a certain threshold. When the buffer empties below the lower threshold the device either activates DTR or sends the xon character. When a given line loses carrier (as a result of the attached device dropping DTR), or when it receives the xoff character, the kernel invokes the terminal multiplexer's stop procedure to disable further transmission interrupts on the given line. The process that was sending output to that line will eventually fill up that terminal line's output queue and will be put to sleep. When carrier is detected again, or when the xon character is received the kernel restarts the transmitter on the given line. The transmitter reenables transmission interrupts and once the output queue has drained below the low water mark, waiting processes are awakened. The cycle repeats until all data are transferred.

4. User Interface

The user interacts with the UNIX operating system through the user shell. This part describes the processes that set up the user environment and initialize the terminal line before the user starts working on the system. For full details about what these processes can do, the reader should refer to their specific manual pages [7].

4.1 Init

The init process is invoked as the last step in the boot procedure. When in multi-user operation, init creates a getty process for each terminal line in which a user may login, and goes into an infinite loop waiting for a death of child signal. The getty process opens and initializes the terminal line, reads a login name, and invokes login. The login process reads the user password, verifies it, and gives the user a shell. Eventually, the shell terminates as a result of an end-of-file or because of a received signal. Since the getty process had turned itself into the shell process, the shell is a child of init. When the shell exits it sends the death of child signal to its parent (init). Init wakes up, identifies the particular line and creates another getty to reopen and reinitialize the terminal line.

Getty does not create another process (does not fork) when it invokes login, but instead overlays itself with the image of the login program through an execv system call. The login program does the same thing; once it verifies the user password, it overlays itself with the image of the login shell.

Init reads the file /etc/tyty and creates a getty for every line whose status is on. From that same line, it also gets the parameters to invoke the getty program with. Of particular importance is the first argument after the program name (getty). If present, getty will use this argument as an index into the gettytab database, from which the terminal
specifications will be obtained. These terminal settings will overwrite previous settings obtained from the default entry. Further logins can be prevented on a particular line by changing its specific entry in the file /etc/lines, and sending a hangup signal to init. init interprets this signal to mean that the file /etc/lines should be read again and terminates any processes associated with the terminal line whose entry has been turned off or no longer exists [8].

4.2 Getty

The getty process opens and initializes a terminal line. The specific device file to open is passed as an argument from the init program. Getty opens this file and executes two consecutive dup system calls. In this way, file descriptors 0, 1, and 2 share the same instance of the opened file, with a reference count of three. Since getty becomes the shell, the shell has these three descriptors already allocated. Furthermore, every user process is created directly or indirectly from the shell through the fork system call (recall that the fork system call increments the reference count on all shared descriptors). This implies that when the user closes descriptors 0, 1, or 2, (s)he would be merely decrementing the reference count on that instance of the file. This guarantees that the device closing procedures will be invoked only when the shell exits (closes all of its descriptors). In this way terminal settings will remain as long as the shell is executing (unless the user purposely decides to change them with ioctl calls).

The getty program reads the gettytab database to get the characteristics of each terminal line. It first sets global defaults defined in the default entry for all terminal lines. If a type argument was passed from the init program, getty also reads that entry in the gettytab database to overwrite default settings. The getty program issues different ioctl requests to set the default and specific characteristics obtained from gettytab. It sets the input and output speeds, parity, line discipline, the value of the different control characters, the terminal modes, and so on. For dial-up lines, the nrx field in gettytab is used to change baud rates upon receipt of a null or interrupt character (this can happen as a result of the user hitting the break key). Parity can be set to accept either odd or even parity on input. If only one is specified, characters with the wrong parity are discarded. On the other hand, if both or none are specified, either parity is accepted. Even parity is generated on output, unless the odd parity bit is set and the even parity bit is cleared, in which case odd parity is generated.

The line discipline is set to the old line discipline. The login program will change this to the new line discipline if the user logging in has a C shell. The different control characters are set to their default values, unless otherwise specified in the gettytab entry. If for some reason these values are modified in the gettytab entry (and therefore remain for the life of getty), the login program will set them back to their default values to be used during the shell. There are three different sets of terminal modes that are used during the getty process. The first one is used to print out a banner and a login message. The second set is used while getty is reading the login name. Finally, the third set is used when getty has read the login name to leave the terminal state properly set for an interactive session (the shell). After getty has initialized the terminal line and gotten a login name, it invokes the login program.

4.3 Login

The login program, as mentioned above, resets all the control characters to their default values. Also, if the user login in has a C shell (as specified by the last field in the entry for that user in the password database), the line discipline is set to the new line discipline. The main role of the login program is to read the user password, verify it against the password database, and invoke the user shell. The login program sets the user and group ids of the process as taken from the password database entry for that user. It also initializes the basic environment variables such as the user's home directory, the type of shell, the type of terminal (as specified in the /etc/term file), the user name and a default path. Once login has done its task it invokes the user shell.

4.4 The Shell

The login shell is the process through which the user interacts with the system. The shell could be seen as a command line
interpreter. It goes into a loop in which it prompts the user for a command, reads the command, and executes it. The shell finishes the loop and exits when it receives the end of file character (usually ctrl-D). The shell can also terminate because of a received signal, such as the hangup, terminate, and kill signals. To accomplish the above task the shell reads a line of input in which the first token is the command to execute, and the rest are the arguments to invoke the command with (piped commands are exceptions to this rule). The shell executes a fork system call, and the child process overlays itself with the image of the specified command (execv). The parent process (the shell itself) waits for a death of child signal before prompting the user for the next command. When the child process (the command) exits it sends the death of child signal that the shell had been waiting for, and the user is prompted for another command.

There is an exception to this sequence that occurs when the user specifies that a command should be run in the background. The user does this by putting an ampersand (&) at the end of the command line. The only difference in this case is that the shell (the parent) will not wait for the command (the child) to finish. Instead, it will prompt the user right away for another command. In this way a user may have several processes running at the same time. Another way this can happen is when processes created by the shell create (fork) child processes of their own.

The shell controls every process initiated at the terminal by means of the process group. Every process created at the terminal directly or indirectly by the shell will inherit the process group id of the shell. In this way every process initiated at the terminal will be in the same process group as the shell. Furthermore, this process group was associated with the terminal in the line discipline open procedure. When signals originate at a particular terminal, the kernel extracts the process group from its tty table entry and sends the signal to every process that has that same process group id (gsignal). Of course, the shell itself must ignore interrupt, quit, and stop signals to prevent these signals from terminating it [8]. On the other hand, the hangup signal is considered to be the result of the user hanging up the line. Thus, the shell is terminated along with every other process in the process group.

5. Modifications

Terminal lines may be used for different types of devices, such as regular terminals, graphics displays, printers, plotters, etc. The amount of I/O processing needed for a given line depends on the device attached to it. For example, in the case of a regular user terminal, input characters like delete, kill, interrupt, and so on are available to make the user interface more friendly. Also, output characters like <cr> and <nl> are mapped to <cr><nl> to make output more readable. However, for other type of devices like plotters and graphics displays, this I/O processing would obviously cause disastrous results. For this reason, serial lines need to be customized according to the device that is attached to them. The three major terminal modes: ccrn, cbreak, and raw, set the base level of I/O processing. Other flags in the tty table entry such as LITOUT, PASS8, CRTERA, etc., and a set of control characters prevent or provide additional I/O processing. This part describes the problems incurred by several devices connected to terminal lines, and modifications employed to solve these problems.

5.1 Existing Problems and Solutions

5.1.1 Flow Control

Several problems were encountered in generating flow control and primarily involved the use of raw mode for devices such as plotters and graphics displays that did not need any output processing. These devices should receive the user generated data just as they are, without being disturbed by the terminal driver output capabilities. Raw mode seemed to be the ideal mode for these terminal lines. However, since no characters are interpreted on input either, software handshaking is not possible because start and stop characters have no special meaning in this mode. Therefore, unless the device can process the data faster than it is being sent, or invokes flow control via hardware handshaking, raw mode is practically useless.
Crmod would solve the flow control problem, but it would introduce the problem of output processing being performed. As a result, EOT characters (normally ctrl-D) that could legally represent a graphics parameter, would get stripped off by the terminal driver and the device would never see them. Similarly, newline characters would be converted to a return-linefeed pair that would obviously not produce the intended result.

The solution is to use cbreak mode. Software flow control is invoked, and most of the above output processing is avoided. Many times the remaining output processing may still cause problems and must also be disabled. This can be accomplished by setting the appropriate parameters in the tty entry, and is described in the next section.

5.1.2 Tty Parameter Settings

The tty table entry contains flags which enable and disable certain terminal control processing settings. (See section 3.6.) Selected values are given in this section, however, for a complete description of each flag, refer to documentation on the tty special file [5].

In crmod and cbreak modes, eight-bit characters can be used on input by turning on the PASS8 bit in the local mode word. Similarly, eight bits could be used on output without any output translations (just like in raw mode) by setting the LITOUT bit. In addition, in cbreak mode, all control characters except the start and stop characters can be individually disabled to prevent distortion of specialized terminal commands.

Initially, it was not clear if these control characters should in fact be disabled or if they could be left set to their default values. The answer was that these control characters could be enabled for output only ports without causing any disturbance on that port’s output. The reason is that all control characters are processed on input, not on output. Most output devices (i.e., plotters) cannot generate any of these characters in the first place. Even if a device used as an output device can input characters (such as a graphics display with a keyboard), they won’t corrupt that port’s output queue as long as they are not echoed. Echoing can be disabled by resetting the ECHO flag in sg_flags. The line editing characters (erase, kill, word-erase, etc.) do not have any meaning in cbreak since line editing is not possible. Remember that in cbreak mode every character is given to the reading process as soon as it becomes available. Furthermore, characters that generate signals (interrupt, quit, and stop) will not cause any problem because there won’t be any process group associated with this port. This is because output ports are normally opened by user processes which are descendants of that user’s shell. As explained previously, these processes will not become process group leaders because they already belong to the process group of the shell. Moreover, line printer ports opened by the line printer daemon (lpd) don’t have this problem either. The reason in this case is because the line printer daemon disassociates itself from any terminal through an ioctl system call. Thus, it cannot receive signals originating at any terminal line.

5.2 Ports Controlled by Getty

5.2.1 Alphanumeric Terminals

The getty process employs three terminal modes while printing the login message, reading the login name, and setting the terminal state for the shell after login (see section 4.2). The operation of an alphanumeric terminal was entirely satisfactory after login, and after the corresponding mode settings (crmod, etc.) were invoked. Before login, however, a problem existed because the terminal line was given a default setting of raw mode by getty. Output redirected to these terminals would be corrupted because of the lack of flow control and output processing. No flow control meant that a subsequent loss of data occurred. Moreover, no output processing meant that lines of output were hard to distinguish from one another because they were not terminated by a <cr><nl> sequence.

Both problems were solved by using crmod instead of raw mode. However, since the getty program echoes the input characters itself, instead of letting the kernel do it, characters need to be available to the getty program as soon as they are typed. Therefore, cbreak mode was used also. If cbreak mode were not used, the getty program would not get any character of the user’s login name until a line delimiting character was entered.
Consequently, the login name could not be echoed until it was completely typed, possibly misleading the user to believe that the terminal was not receiving input. These two bits, along with other appropriate flags such as new line and tab delays were set using the fl field in the gettytab entry, which specifies the set of flags to be used while reading the login name (section 4.3). The gettytab entry is specified in /etc/ttys, with one entry for each terminal.

5.2.2 Nonstandard Terminals

These types of terminals were mostly used as output devices. However, since the devices could also be used for input, it was decided that a user could login on those ports thus requiring a getty process. An example of such a device is an AED graphics display, which also can be used as an interactive terminal. For these terminal lines, a new gettytab entry was created, and the /etc/ttys file was modified accordingly to contain the name of this entry. In this entry, a different set of flags was used while reading the login name (fl field of the gettytab entry). Cbreak mode was used to prevent EOT characters from being stripped off the output sequence, and to achieve flow control. The crmod bit was turned off to prevent <cr> and <nl> to be mapped to <cr><nl>, as were the flags for tab expansion. These settings prevented graphics commands from being corrupted by unwanted output processing. All the above output processing could also be suppressed by simply enabling the LITOUT bit in the local mode word. As it turned out, this bit was necessary to be able to send 8-bit data to the graphics display anyway. This last setting required a minor modification to the getty program to actually update the flags in the tty entry with the local mode word just before reading the login name.

All the control characters except the start and stop characters were disabled since they would be of no use anyway when sending output to the display. This was done by setting the appropriate fields in the gettytab entry to the octal value 377 (negative one). As explained in section 4.3, when getty obtains a login name it resets the mode flags to be left for the login process. Also, the login process (section 4.4) resets the control characters to their default values to be used in an interactive session (the user shell). In this way, these terminals can be used as a graphics display while waiting for a user to login, and then have their settings changed to be used as a regular terminal once a user logs in.

5.3 Line Printer Ports

Line printers are handled by the line printer daemon. The line printer daemon opens and initializes the line printer ports. Since line printers are attached to terminal lines, the same underlying communication parameters apply. The line printer daemon reads the printcap database to get the characteristics of each line printer attached to a terminal line. The syntax of the printcap database is very similar to the syntax for the gettytab database. The reader should refer to the printcap documentation for further information about the printcap capabilities. The line printer interface is handled through the use of sockets, which are not discussed on this paper. Moreover, the line printer interface was not modified since it was already satisfactory.

5.4 Ports with Other Devices Attached

Along with ports with user terminals and printers handled by the getty and lpd processes, there were other ports that were not initialized by any process. These ports were connected to a raw device (no I/O processing needed) that did not require logging in, and therefore did not require a getty process. An example of such a device is a plotter. Originally, the user process would have to set the appropriate communication parameters before sending data to the port. However, this violated the UNIX philosophy that a user should be able to send information to any (suitable) file or device without any such intervention.

This problem was solved by spawning a process from /etc/rc.local which initialized all the "raw" ports with their specific settings. (The rc.local script is invoked just before starting multi-user operation.) This process read a new database, /etc/rawtab, to make the settings for every uninitialized port. The format of the rawtab database was made identical to that of gettytab. In our case, since these devices did not require as much I/O processing as regular terminals, the database was reduced to the
hardware communication parameters (speed, 8- or 7-bit characters, and parity) and the establishment of a flow control protocol. If hardware handshake is available (such as for lines with full modem control), the MDMBUF flag can be set to invoke flow control via the carrier flag. In this case, raw mode is employed with 8-bit characters and no parity. If 7-bit characters are desired, cbreak mode can be used instead. Of course, parity can be specified when using 7-bit data. For software handshaking, cbreak is used to allow the start and stop characters to be interpreted. If 8-bit data is required, the PASS8 flag is used for input, and the LITOUT flag for output. No parity is available in this case.

In our case, one port that was connected to an HP7475 plotter was modified in this manner. The hardware handshake method was tested to confirm the methodology, and required a cable with an extra wire to connect the DTR line (pin 20) of the plotter to the carrier detect line (pin 8) of the dmi port. The port was set to raw mode and communicated correctly and with proper flow control.

In spite of this success, software handshaking was preferred since a dmi multiplexer supports full modem control only on two of its eight ports. Thus hardware handshaking would require the plotter to be connected to one of those two ports. Consequently, software handshake was selected, and cbreak mode was used. The speed was set to 9600 baud. Both the PASS8 and LITOUT bits were turned on to allow for 8-bit characters required in graphics applications. A “de” field was added to rawtab to specify the device file name to be opened. Gettytab does not need this field since the device file name to be opened is passed from the init process to the getty process. Printcap also provides this field (although its name is different).

As stated in section 3.6 all of the control characters have no effect on the I/O processing of (primarily) output only devices. There was a question, however, as to whether enabled start and stop characters could conflict with hardware handshaking. Similarly, there was a question as to whether an enabled carrier flag could conflict with software handshaking. Neither case presented a problem for the following reasons:

1. Most devices are manufactured so that they can only do one form of handshake at a time. Therefore, as long as the device and the system agree on which protocol they are using, no conflict will occur.

2. Even if there exists a device that can do both protocols at the same time (say drop DTR and send an xoff character), no conflict will occur. This is because either the change in carrier state or the receipt of the xoff character will be serviced before the other. The one that is serviced first will set the terminal state to stopped and will disable the transmitter on that line. The other will find the state of the terminal line already stopped and will have no effect, just as when typing two consecutive stop characters at a terminal, the second one has no effect. Similarly, when carrier is detected and the xon character is received, the first one will clear the stopped state and will restart the transmitter. The other one will “clear” the already cleared stopped state, and will “start” the already started transmitter.

This approach of running a process from /etc/rc.local may have a possible drawback compared to getty and lpd. The problem would occur if a user intentionally or accidentally changed the settings on a given line. In the case of getty, if the user changes any parameters from the shell, the line will return to its normal state when the shell exits and init creates another getty on that port. Line printer ports do not have this problem because the user does not have direct access to the parameters of these ports. If a user modifies any raw port parameters, however, they must be manually reset. Consequently, the user must leave the port in the same state as it was initially. This problem is not considered to be too serious since user intervention is no longer needed for proper operation of these lines, except for very special circumstances.
6.0 Conclusions

This paper presented the 4.3 BSD UNIX terminal line interface. Although many of the principles and algorithms are the same as for other UNIX versions, some variations exist. Of course no source code has been included due to copyright restrictions. The two major problems, flow control and lack or excess of I/O processing were solved for every terminal line. The solution consisted of specifying the right parameters for each line in the provided databases gettytab and printcap, and in the newly created database, rawtab. For the latter, a new process was created to actually read these parameters and initialize the specified ports. Although it was initially believed [9] that a kernel modification would be required to prevent special control characters from being automatically reset upon opening, this was not necessary since these characters have no effect on output ports.

The terminal interface has been handled in such a way that the user does not have to set the parameters for any port. All that is necessary is to open the port, if it has not already been opened by the system, and send data.

References


Appendix: Internal Kernel Functions

nodev()
Simply returns the ENODEV error code.

nulldev()
Does nothing.

namei(pathname)
Converts a pathname into a pointer to a locked inode. It uses several other algorithms beyond the scope of this paper.

openi(inode, mode)
Invokes the specific open procedures for character and block special files.

sleep(event, prio)
The process sleeps waiting for event. Wakeup will notify the process when event has occurred. When awakened, the process will enter the scheduling queue at priority prio.

wakeup(event)
Scans the sleep queue and wakes up processes waiting on event.

gsignal(pgrp, sig)
Sends signal sig to every process with process group id equal to pgrp.

psignal(proc, sig)
Sends signal sig to process proc. Called from gsignal.

subyte(addr, c)
Transfers a character from kernel to user space. The value of c is placed into the user address addr.

bcopy(from, to, count)
Copies the specified number of bytes within kernel space.

copyin(from, to, count)
Copies the specified number of bytes from user to kernel space.

copyout(from, to, count)
Copies the specified number of bytes from kernel to user space.

uiomove(addr, cnt, rw, uio)
Transfers the specified number of bytes from user to kernel space (copyin), from kernel to user space (copyout) or within kernel space (bcopy), depending on the rw (read/write) and uio structure flags. One of the addresses to transfer the data from/to is addr and the other one is contained in the uio structure. The offset, count, and base fields of the uio structure are updated.

getc(clist)
Gets the next character from the specified clist. Cblocks that become empty are returned to the freelist.

putc(c, clist)
Puts a character at the end of the specified clist. If necessary, new cblocks are allocated from the freelist.

ureadc(c, uio)
Transfers a character from kernel to user space (subyte) or within kernel space. Updates the base, count, and offset in the uio structure.

b_to_q(buf, count, clist)
Copies the requested number of bytes from buf to clist in at most the size of a cblock chunks. This is because bcopy transfers bytes to contiguous memory locations. New cblocks are allocated from the freelist.

q_to_b(clist, buf, count)
Copies the clist to buf in the same way as above. Empty cblocks are returned to the freelist.

canq(from, to)
Transfers the characters in the from clist to the to clist. Uses algorithms q_to_b and b_to_q.
An Update on UNIX Standards Activities

Shane P. McCarron, NAPS International

This is the fourth in a series of articles on UNIX related standards activities. In this narrative I am going to cover a slightly wider area than usual. There have been developments at the ANSI X3 level, the National Institute of Standards and Technology (formerly the NBS), and within the POSIX committees which deserve attention. I will apologize at the outset for the length of this article, but I feel that all of the information is timely and important. In addition to information on group activities, included with each report is a contact person from whom you can get more information about these developments, and the names of USENIX Standards Watchdog Committee members through whom you can relay your opinions to the specific standards committees.

On the subject of the USENIX Standards Watchdog Committee, this series is now an activity of that group. Last quarter I used the article to solicit participation in the committee, and I am pleased to report that we have a number of new associate members. While I don’t know everyone involved, I would like to thank those who have contributed: Anna Marie de Alvaré, Ted Baker, Mark Colburn, Doug Gwyn, Sol Kavy, Doris Lebovitz, Kevin Lewis, and Stephen Head. We are still in search of members for this group. While we will accept all comers, we are particularly interested in filling out our rather lean international input department. If you would like to be involved in the Watchdog activities, or know of someone who might be a good candidate, please contact:

John S. Quarterman
Texas Internet Consulting
701 Brazos, Suite 500
Austin, TX 78701-3243
(512) 320-9031
jsq@longway.tic.com

or

Mark Colburn
NAPS International
117 Mackubin St., Suite 1
St. Paul, MN 55102

(612) 224-9108
mark@naps.mn.org

C Language Standard

X3J11 (ANSI C standardization committee) met 26-30 September 1988 in Sunnyvale, CA. Principal business of the meeting was to respond to comments received during the third round of formal public review, which had closed earlier. In addition to the 15 letters formally registered with CBEMA’s X3 Secretariat, 27 unregistered letters were included. There were 632 items contained in these 42 letters. In order to address them all, the committee was divided into response preparation subgroups, each of which tackled a subset of the total list of items. From time to time, the whole committee reassembled to hear issues that the subgroups were not able to completely resolve by themselves. In several cases a straw vote was taken to determine the sense of the committee. The resulting responses were formatted to produce the official X3J11 Response Document.

At the Sunnyvale meeting, several editorial changes to the draft standard were approved. The working definition of “editorial” was: A change is editorial if it clarifies the original intent of the committee; it is substantive if it changes the committee’s intent.

There were several issues that were of particular interest to the UNIX/POSIX community:

- A change was made that clarified the ability of an application to portably reestablish a signal handler for the signal that caused entry to the handler. This is indeed allowed under the standard. The important passage reads:

If the signal occurs other than as a result of calling the abort or raise function, the behavior is undefined if the signal handler calls any function in the standard library other than the signal function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler) or
refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile
		sig_atomic_t.

- IEEE Std 1003.1-1988 (POSIX) requires that the fflush function specified by X3J11 have some additional semantics. The committee confirmed that this was indeed allowed by ANSI C.

- The IEEE P1003.1 working group had asked X3J11 to consider making the symbol "environ" a reserved external identifier. This would mean that an ANSI C conforming portable application could not use the symbol. This request was made because in traditional UNIX implementations application launch routines initialize this variable to be a pointer to the user's environment variable list, and this may not be what a strictly conforming ANSI C application would expect. This issue was raised before the committee, but found no support for a change; the committee response for this was as follows:

The ANSI C and IEEE 1003.1-1988 standards are not necessarily in conflict here, although it is true that in order to avoid the name-space conflict a mutually conforming implementation must rely on some mechanism such as 'global symbolic equate' or a zero-size global object 'environ' in a separate library module immediately preceding the module that defines storage for '__environ' (the name used by the C run-time startup code). Implementor control over the way the linker operates, while inappropriate to require for the more universal C Standard (hence the constraint on uniqueness of external identifiers), is not unrealistic to expect for most POSIX implementations. Several implementors have in fact indicated their intention to provide such a feature.

Another solution, of course, would be to use separate run-time startup modules for strict ANSI-conforming and vendor-extended (possibly POSIX-conforming) implementations, perhaps via a compiler flag. This may be useful anyway for hiding extensions in certain standard headers.

Because no substantive changes to the proposed standard resulted from the third-round review process, X3J11 voted unanimously to submit the standard as edited to reflect approved editorial changes to CBEMA X3 as the proposed ANSI C standard, pending completion of additional review as described below.

The draft Response Document was reviewed first by a small group of X3J11 members using electronic mail, then by a group meeting at Plum-Hall in Cardiff, NJ, on 20-21 October 1988. The responses were checked for completeness, consistency, and accuracy, and occasionally the original responses were changed to achieve those goals, or to meet the additional requirement that no unauthorized substantive change to the proposed standard could be promised by any response. Changes made at the review meeting were subsequently edited into the master Response Document. Two significant areas of the standard were affected by editorial changes resulting from the response review process: the description of pointer arithmetic was substantially reworked to avoid reliance on an assumption of byte addressability, and the specification of the role of type qualifiers was rewritten to clarify the significance of what was called the "top type" (now called "type category").

On 1 November 1988, the draft proposed Standard itself was reviewed by several X3J11 members in a meeting at Summit, NJ. Since the draft already contained the results of the Sunnyvale meeting and response review meeting, very few changes were found necessary at the meeting.

On 9 November 1988, the Rationale Document (designed to accompany the Standard) was reviewed by a group of X3J11 members meeting in Cambridge, MA.

On 14 November 1988, copies of all three documents (Response, Standard, Rationale) were express-mailed to the 15 X3-registered commenters, who had 15 working days (from November 18) in which to reply to X3 if they felt that their items were not properly addressed by X3J11. The commenters were encouraged to first discuss problems with X3J11 members, in hopes of reducing the amount of negative feedback to X3.

On 9 December 1988, all three documents plus any feedback from the commenters were to be submitted to CBEMA's X3 Secretariat as
the official X3J11 proposal for the ANSI Standard for Programming Language C. After review by X3, assuming no problems arise, the proposed Standard will then be submitted to ANSI for official ratification as an ANSI standard. It seems probable that the final ANSI C standard will be published some time during 1989.

The USENIX Standards Watchdog Committee contact person in X3J11 is Doug Gwyn. He can be reached at:

Doug Gwyn
US Army Ballistic Research Lab
801 L Cashew Ct.
Belair, MD 21014
gwyn@brl.mil
+1 (301) 287-6647

National Institute of Standards and Technology

On August 30, 1988 (four days after publication of the previous in this series) the NIST published their Federal Information Processing Standard for POSIX. Sufice it to say that this FIPS is finally approved, but differs substantially from the approved IEEE standard in a few key areas. The NIST is now working to revise the FIPS so that it is more in line with the real standard. This new FIPS should be announced in the Federal Register in early January, and after time for public comment and review, will be formally approved. The NIST expects approval sometime in summer 1989.

In the last article I mentioned that the NIST had announced their intent to create FIPS in other areas. They have now released a preliminary FIPS for System Administration and are about to release one for Shell and Tools. They have also stated that, by year's end they will release a FIPS on utilities with User Interfaces (like vi). While in the case of Shell and Tools the NIST is going to use Draft 8 of the 1003.2 standard, there are no existing formal standards in the other areas. Instead of waiting for standards bodies to develop mature documents, the NIST is going to a number of different versions of UNIX, and picking those things that look neat. The System Administration FIPS in particular is disturbing. There are a number of utilities in there from AIX (IBM's version of UNIX), Xenix (SCO or Microsoft, I can't tell), and of course the SVID (from AT&T). This ensures that there is no existing system that will conform to the FIPS on day one, and also shackles the newly formed IEEE working group on System Administration.

I really don't know what the NIST is trying to achieve. It appears they are working toward their stated goal of creating a full suite of specifications to flesh out the Applications Portability Profile (a conceptual model of portability specifications created by the NBS over the last few years). I used to think that they had some sort of hidden agenda, but I don't believe that any more. I used to think that they were trying to railroad standards to make sure that the government's needs were satisfied. In this I have also been proven wrong. They have now shown their ability to create standards at will, thereby invalidating the work of the standards bodies before they can even begin. This interesting turn of events proves that in their previous heinous acts they were just being nice. They could have superceded the process altogether if they had really wanted to!

It was bad enough when the work of the committees was being affected by the arbitrary timelines imposed by the NIST, but now they have created a framework within which any standard on, say, System Administration, will have to fall if it is to be taken seriously by the vendor community. What vendor in its right mind would conform to a formal standard that was not in line with the standard being required by all U.S. federal agencies? The obvious answer is "vendors that don't want to sell to the government." In other words - none. Moreover, what vendor sponsored committee member is going to propose something for a standard that would make their employer not be able to sell to the federal government? Again, none.

I have given the NIST an opportunity to rebut the comments made above, and they are in the process of doing so. I will publish their comments as soon as I have them available. However, I would guess that they will say something like "These are just first cuts. In the future we will modify the FIPS to conform to standards produced by standards making bodies." That's great, but it really doesn't help. First, it would be a disservice to the
federal user community to force them to change from an environment in which they have become comfortable. Second, it is a mistake to assume that the vendors are going to want to conform to one standard for a while, and then change over later. If there is a standard that is being required by a substantial part of the user community, then that is the one to which vendors are going to conform. And if vendors conform to it, it then becomes the existing practice that must be formalized by standards bodies like IEEE P1003. It's a vicious circle, and in the end the losers are the users. They are being handed an ill-considered standard; one that is being foisted upon them just because some small group of people, after consulting with a handful of their (rather unique) user community, have decided that this is the way it is going to be.

In defense of the NIST, I know that they are not trying to destroy the standards making process. They are just a bunch of people trying to do their jobs the best way they know how. It is unfortunate that in doing so they may end up doing more harm than good.

The USENIX Standards Watchdog Committee has no contact person with the NIST. For further information on NIST activities you can contact me or Roger Martin.

Roger Martin
National Institute of
Standards and Technology
Software Engineering Group
Room B266
Technology Blvd.
Gaithersburg, MD 20899
rmartin@swe.icst.nbs.gov
+1 (301) 975-3295

1003.0 – POSIX Guide

At this meeting of 1003.0 the group was presented with the first working draft of the guide document. Throughout the week the committee met in both small groups and in plenary sessions to expand on the first draft and start nailing down the exact focus of the project. In particular the group concentrated on the issues that had been raised and entered in the Issues Log, the overall objectives and the scope of the document. The purpose of the discussions was in part to clarify the strategic goals of the committee, and in part to prioritize those items that have already been decided upon.

Each small group that met worked on a particular area of the draft, expanding on its contents. As the full working group could not decide on the level of detail that should be included in each section, it was left up to each small group and revisited later. Topics that are being covered include: the Benefits of Open Systems, Key Open Systems Areas.

The USENIX Standards Watchdog Committee contact for 1003.0 is Kevin Lewis. He can be reached at:

Kevin Lewis
DEC
Suite 645
1331 Pennsylvania Avenue NW
Washington, DC 20004
klewis@gucci.dec.com
+1 (202) 383-5633

1003.1 – System Services Interface

The big news from this meeting of the 1003.1 working group is that its Chair, Jim Isaak, has resigned after 5 years of work. Jim is also Chair of 1003, the convenor of the ISO work item on POSIX, and a passel of other things; consequently he felt that he could no longer contribute the amount of time to 1003.1 that is really necessary for a working group chair. I would like to take this opportunity to thank Jim for all of the effort he put in to making the first POSIX standard a reality. We are fortunate that there are people like him in the industry.

The new chair of the committee is Donn Terry. Donn has been co-chair for a couple of years now, and has been the real chair (if not in name, then in actions) since the standard went to ballot in November of 1987. He is one of the original members of 1003.1, and is also the chair of the US Technical Advisory Group on POSIX to ANSI. Donn coordinated the last two rounds of balloting on the 1003.1 standard, and did an excellent job. I'm confident that he will prove to be as able a chair as Jim.

Almost as important is that the standard is now available in print. The bound version of the standard, while almost unreadable because of IEEE enforced formatting changes,
and hard on the eyes because of its ugly split-pea-green cover, is now available for $16 (members) or $32 (non-members) from the IEEE office in New Jersey. For a copy, please contact:

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854
+1-201-981-0060

After electing the new chair, the working group got down to business. They continued their work on extending the first POSIX standard, IEEE Std 1003.1-1988. Their primary areas of focus are now a new archive format, a functional interface for terminal interaction, and cleanup of the first standard. In addition the group starting forming a subgroup to be the interpretations committee for the released standard. Each standard must have a "supreme court" of sorts. Users of the standard may submit formal questions to the IEEE, and those questions will in turn be conveyed to the interpretations committee. It is up to this committee to figure out the answers to the questions, and then to modify the standard if necessary so that in future printings the question doesn't come up. More about this as it develops.

One issue of great import is internationalization of the standard. The international community has some concerns, particularly in the areas of character sets and the use of the words "byte" and "character." These concerns were in particular voiced by the Japanese representatives at the October meeting of WG15 in Tokyo. The committee tried to be very careful when drafting the standard, but apparently not everything was covered. In any event, the working group now has to write an appendix to the standard which specifies the intent of the group regarding international implementations of POSIX. The standard is not really an implementors guide, but the appendix should provide a better guide to the intent of the group. Hopefully this appendix will be enough to keep the international community at bay long enough for the standard to be ratified as an ISO Draft International Standard (DIS).

On a related note, the ISO Working Group for POSIX (ISO/IEC JTC1/Sc22/WG15) has recommended that DP 9945 (the draft proposed international standard POSIX) be elevated to a DIS. This means that the standard has to go through another (international) balloting period before it can be a real international standard. Personally, I don't anticipate any trouble.

The 1003.1 committee hopes to ballot a revised version of the standard within two years. This revised version would contain a new archive format, some additional functions were left out of the original but are now felt to be necessary, and any clarifications that have come from the interpretations committee. In addition all of the interfaces in the standard will be described in a way that is programming language independent, and there will be a chapter that has the C language binding to this language independent description. It sounds like a big job, but the committee is optimistic. It is also small enough now that it might just get it done in that time frame.

I am the USENIX Standards Watchdog Committee contact for 1003.1.

1003.2 - Shell and Tools Interface

This working group never ceases to impress me. In September the group was given about three weeks to go over draft 7 of the standard and review it as if it were a formal ballot. This means that problems discovered in the draft must be reported to the committee using the formal POSIX balloting format, within the specified time limits, in order to be considered. A surprising number of people were able to work very hard and come up with about 1500 objections to the 600 page document.

Okay, so a lot of people, given 3 weeks, can really find a lot of problems with a somewhat immature document. Maybe not terribly impressive. Then a group of 40 people meet in Hawaii, not a place known to be conducive to work, and manage to review every single objection and resolve them! This is truly amazing, and I think everyone at that meeting (including myself) deserves a medal. Moreover, I would like to take this opportunity to publicly eat the words I wrote last quarter. They may just pull it off! The draft that goes out for balloting in the formal IEEE process is certainly in much better shape than the 1003.1 document was when it first went out. Also,
P1003 learned a lot from the .1 ballot, and that knowledge should help make the balloting of .2 smoother.

Some specific changes of interest were:

- Based on a decision from the previous meeting and several balloting objections, the fgrep and egrep commands have been removed from the standard, and the functionality that they provide is being encompassed in the definition of grep. This new grep will have options -E and -F which will give it the exact functionality of egrep or fgrep, respectively.

- The draft has a command in it called coldef. coldef allows the portable definition of collation sequences, which can then be used by utilities that do string comparisons with the C Standard function strcmp. The theory goes that an implementation will provide applications with a means for creating collation sequence definitions (coldef), and then also allow the application to specify which collation sequence to use when calling utilities like sort (through the environment variable LC_COLLATE).

It all sounds pretty good, but the definition of coldef was so incomplete and confusing that some balloters suggested it be removed from the standard altogether. The definition of this utility now provides for a lot of additional functionality, and is much clearer than it used to be. While this part of the standard is not talked about much, I believe that it is probably the most important part. The international aspects of POSIX are sort of obscure, but they will allow for more portable applications, and also allow for some previously unheard of uses for utilities like sort.

- A closely related utility, xform, was placed in the standard to allow for the transformation of strings by a shell script just as can be done using the strxfm function in Standard C. After much discussion in the small group, this command was removed from the draft. While there was some dissenting opinion, the majority thought that this would have very limited usefulness to a portable shell application. As I was the dissenter, I can say that I wanted it in because there is no other way to portably compare strings in the shell from an international perspective. If a user enters something and then later you want them to enter it again, you cannot portably compare those strings without the xform utility. Alas, you win some...

- An interesting development was the decision that the C language functions in the standard be moved into a chapter for C Language interfaces, and that their original position in the document be reserved for the language independent descriptions of some of the functions. In the end it may be that some of the functions are really not ones that need to exist in other languages, and as such should not be in the language independent section. This event is interesting because it shows the intent of this working group, and indeed all of the POSIX working groups, to describe their standards in a language independent manner. This was a requirement of the international community, and I am glad to see that it is being carried out.

- In what I consider a victory for the users of the world, the UUCP style commands in the standard have been moved out of the document and into an appendix. These commands, uuxqt and uuname, have been in the standard for about a year, but no one could really figure out why. As described there was no underlying transport mechanism or protocol defined, so they could not possibly have been reliable in any event. They were placed in the standard as a spear; something that you could throw out and have no idea if it worked or not. Depending on the feeling of the balloting group, these commands will either be fleshed out into a full definition of the UUCP “networking” system, or removed from the standard altogether.

- While the UUCP commands are gone, the message sending command sendto is still in the standard. This command allows an application to send text to an address with an implementation defined format to be deposited in an implementation defined location and delivered in an implementation defined manner. No kidding. That’s what it says. It also used to say sendto -r would try to read from your personal implementation defined storage location, but that it might not do anything. Fortunately, the working group couldn’t figure out a single reason why a portable application would want to read mail. While this is usually not enough cause to remove something
from a standard, when coupled with the danger that it might not do anything if executed, the evidence seemed to lean toward removal. This option has been axed.

- There is now a section of the standard on application installation. Actually, there has always been a section for that, but until now it has been full of stuff that wasn’t really worth reading. The new definition is a little bit complex, but it seems to be fine. It allows for an application, on installation, to determine what system resources are available, and to then sort of dynamically inform itself about them. There is also a system resource database, and all sorts of other neat stuff. I don’t have a handle on all of it yet, so stay tuned.

There were all sorts of other changes made to the draft, but they are primarily editorial, and are of course all subject to review by the balloting group.

The schedule for balloting goes something like this: Assuming the document gets to the balloting group in mid-January, the period will close in mid-February. Then all of the received objections will have to be resolved or commented on, and it will be recirculated. This may happen several times before the document is finalized. Since each recirculation & resolution period takes 3 to 4 months, it could be early 1990 before we see a ratified standard.

In the meantime, since the working group doesn’t have anything to do with a standard while it is going through balloting, work will progress on the new User Portability Extensions supplement. The idea here is that a supplement to 1003.2 will be released soon after the initial standard. This supplement will describe the traditional UNIX utilities that have user interfaces (e.g. vi). Note that the utilities to be described are the traditional ones, and have nothing to do with windowing-mouse interfaces. Work on that topic is progressing in other areas.

I am the USENIX Standards Watchdog Committee contact for 1003.2.

1003.3 – Testing and Verification

This POSIX working group met along with the others in Honolulu in October. The agenda included a status report on NIST activities, review of previously assigned action items, developing a strategy for future work with other P1003 (POSIX) working groups, revision of Draft 7.1 document, and assigning new action items.

Roger Martin (NIST & P1003.3 Chair) gave a status report on the current NIST FIPS and their Conformance Testing Policies for the POSIX FIPS. He stated that this “Initial” POSIX FIPS has been approved and they intend to revise the FIPS now that the P1003.1 Standard is finalized. The NIST Test Suite, PCTS, has been provided to NTIS (National Technical Information Service) for public distribution at a price of $2500 and is being distributed since September 5, 1988. Its distribution was awaiting FIPS approval. Roger Martin also presented a proposed schedule for a series of Application Portability Workshops sponsored by NIST. He described a workshop that had taken place in September 1988 covering Shell & Tools, System Administration, and X Windows. One of the areas to be covered in a future Application Portability Profile FIPS and workshop include the Terminal Interface Extension. The workshops are intended for implementors and users.

The remainder of the meeting concentrated on rewriting and restructuring the Draft 7.1 document, including test assertions.

During the week of meetings one small group of Test Assertion Reviewers continued to update the 1003.3 Draft 7.1 assertions.

Two other small groups concentrated on rewriting and restructuring 1003.3 Draft 7.1 document. One group’s emphasis was the development of 1003.3 Generic Test Method chapters (i.e. terminology, testing levels, generic PCTS output). The second group’s emphasis was in developing 1003.1 specific Test Method sections.

The P1003.3 group is gearing up for balloting this standard in early 1989. Each P1003.3 member is part of the “mock” ballot group, identifying and formulating any possible objections.

Future work of the P1003.3 committee was also addressed. The P1003.3 Working
Group wants to influence the other P1003 Working Groups into writing testable standards. To achieve this, a liaison program will be implemented to have members from P1003.3 working in a liaison fashion in each of the other working groups.

The P1003.3 working group Project Authorization (PAR) will need to be revised in order for the group to develop an overall Test Method standard and the development of specific standards for each appropriate 1003 activity.

The USENIX Standards Watchdog Committee contact for 1003.3 is Doris Lebovits. She can be reached at:

Doris Lebovits
AT&T
Rm 5-211
190 River Rd,
Summit, NJ 07901
lebovits@attunix.att.com
+1 (201) 522-6586

1003.4 – Real Time Extensions to POSIX

In the past I have written some things about this committee that were pretty critical. I saw them as progressing too slowly to have the impact I hoped they would have. I know that nothing I wrote or said motivated them, but I am now happy to report the following: 1003.4 is almost ready to go to mock ballot! Apparently it all came together in the last couple of months, and they are now ready to ask a wider group for an opinion. They plan, at the January meeting, to go through all of their working papers and appendices, integrate them into the draft, and them submit it for a mock ballot before the April meeting. The results of the trial ballot will tell them how much more work they need to do before going to formal ballot. If all goes well, they should be able to ballot after the July, 1989 meeting. Given the way ballots tend to go, that would mean a completed standard in early to mid 1990. This is particularly exciting since dates in 1991 had been handled about previously. Getting this standard out a full year earlier is astounding.

Many people are probably curious as to what is contained in a Real Time standard. Well, many things that didn't make it into 1003.1, for starters. Here is a partial list: Asynchronous I/O, Shared Memory, IPC, Asynchronous Event Notification, Process Memory Locking, Timers, Priority Scheduling, Semaphores, Synchronous I/O, and Realtime Files.

Some of these are going to be particularly contentious. In particular Events and Memory Locking could be a problem. The mock ballot should flush out these issues so it can be cleaned up before formal balloting in the fall.

The USENIX Standards Watchdog Committee contact for 1003.4 is Sol Kavy. He can be reached at:

Sol Kavy
Hewlett-Packard
19477 Pruneriidge
Cupertino, CA 95014
sol@hpda.hp.com or hpda/sol
+1 (408) 477-6395

1003.5 – Ada Language Binding

This group is interesting. They have now distributed draft 1 of their standard to the working group, but they are very close to finishing.

The primary goal of the P1003.5 working group is to produce an Ada language binding for the operating system services interface defined by the P1003.1 standard. This work has progressed to the stage of circulating draft chapters within the group. These chapters are to be reviewed at the next .5 meeting (in January).

The last .5 meeting was 7-9 September 1988 in Minneapolis, MN. One of the issues discussed there was improving coordination with the rest of P1003. The last two P1003 meetings conflicted with major Ada meetings, so that .5 chose to meet separately. This has not been good for communication. Fortunately, there are no major conflicts with the Ft. Lauderdale meeting, and they will attempt to synchronize future meetings with the rest of the P1003 working groups.

Major issues which were discussed at the September meeting included: (1) the relationship of Ada I/O and POSIX I/O, and how this relates to P1003.0; (2) (missing) support for
Ada in the P1003.2 standard; (3) real-time features required by Ada, and whether P1003.4 will provide these; (4) changes to .1 between draft 12 and the final version that will require changes to the .5 draft chapters; (5) the relationship of Ada tasks to POSIX processes; (6) whether the organization of the P1003.5 document should mirror the .1 document.

One of the central problems they face is reconciling the relationship between Ada tasks and POSIX processes. Unlike POSIX processes, Ada tasks share a common logical address space. If they map Ada tasks onto distinct POSIX processes, they need a way to share memory and file handles (opened after fork) between processes, which is not provided in .1. (Support for shared memory is on the .4 agenda, but the final form remains uncertain.) Moreover, there are applications of Ada tasks that require task switching, creation, and termination to be performed much faster than may be possible for POSIX processes.

On the other hand, they might implement tasks as multiple threads of control within a process, but then they run into other problems. Unfortunately, multiple threads of control within a process cannot be supported well without some cooperation from the OS. For example, a blocking system call by one thread should not block other threads. For another example, what happens when one task is in the middle of a system call and another one forks? (For now, P1003.5 agreed that Fork/Exec should be allowed, but that their effects in a multitasking Ada program are implementation dependent.)

The concept of POSIX support for "lightweight processes" is appealing. The group will explore the applicability of such a solution. In order to broaden the base of interest, they have agreed to sponsor a "Birds of a Feather" discussion on this issue at the Ft. Lauderdale meeting.

Another major problem is reconciling POSIX signals with Ada semantics. The .5 group has done some preliminary work on this. The concept most closely corresponds to an asynchronous Ada exception, but this construct is of questionable legality. The legal Ada mechanism appears to be entry calls, but this presents other problems. Much work remains.

A third problem area is data representation, and character sets in particular. POSIX already has problems with international character sets, arising from special uses of certain glyphs, and from an implicit assumption that characters are represented as bytes. Ada makes this worse, since it specifies a very specific standard character set (ASCII). The .5 group proposes to recognize POSIX characters (and strings) as distinct from the Ada versions, and to provide transfer functions for situations where one must be converted to the other.

Due to a conflict with the ACM Tri-Ada conference, 1003.5 was not able to meet with the rest of the POSIX committees in Hawaii. However, several individual members volunteered to attend as liaison with the other groups. This will probably turn out to have been very helpful in resolving some questions about division of responsibility.

It became clear during the 1003.1 meeting that .5 should not feel constrained to mimic the C-language binding, but should move ahead boldly to create a true Ada interface. Further, it appeared that due to Ada's strong typing requirements .5 might be closer to a language-independent version of .1 (required by ISO) than the present .1 standard, and might well influence the form of the future .1.

Meetings with the .4 revealed that support for Ada's real-time requirements might be provided by that group, but not necessarily in a suitable form or soon enough. In particular, the subject of lightweight processes, which might be used to implement Ada tasks, is not on the .4 agenda. This leaves the subject open to be addressed by .5.

These, and observations by other .5 members serving as liaisons are likely to influence the direction of work when the group next gets together.

The USENIX Standards Watchdog Committee contact for 1003.5 is Ted Baker. He can be reached at:

Ted Baker  
Department of Computer Science  
Florida State University  
Tallahassee, FL 32306
Definitions

The following information was presented:

1. The structure of the definitions will be similar to 1003.1 structure: terminology section, conformance section, general terms, general concepts, and acronyms.

2. The draft 0 definitions were based on four documents: ISO, ECMA, IEEE Std 1003.1-1988, and the Orange Book.

3. The goal of this group is to assure that 1003.6 definitions are consistent and relevant to 1003.6 areas without overstepping or duplicating existing definitions from other 1003.x groups. In case some of the 1003.6 definitions conflict with 1003.X ones, the action will be to propose a redefinition of the term.

P1003.6Scope

The proposed Scope was discussed and the conclusion was that it needed reworking. The area of I&A was considered not addressed, as were trusted recovery (which the real-time people may need) and others. In the draft a lot of the issues that will not be supported right now are marked so because of lack of experience or not enough technical maturity. The important point is not whether we have the experience or not, it is to be aware of areas where users want security, areas where the committee thinks security should be provided, and point them out in the Scope. If areas become a problem later, they can be dealt with at that time.

For the next draft of the 1003.6 document, the table of contents will contain: Scope, Definitions, Feature Overview, Existing 1003.1 Functions, Existing 1003.2 Commands, Section for Each Feature, and an Appendix.

The Feature Overview covers a discussion, functional interface summary and command summary of each feature. Then in the feature section there will be the functions, commands, descriptions, and security specifications.

In the appendix there will be a rationale that maps to the document sections.

It was remarked that all the future features such as Networking and System Administration should be annotated in an
appendix as areas that will be covered as extensions.

Discretionary Access Controls

This group was the one with the most activity, generating a lot of conflicting ideas even within itself. However, they did resolve to put together first the Rationale section of the document and work on the agreeable parts, then later debate the contentious ones. One of the conflicting topics was default Access Control Lists. This is probably needed, but apparently will not be within the scope of the standard.

Privileges

Privileges is a topic wrought with philosophy, and computer professionals love to be philosophers. In spite of this, definitions of privilege and certain types of privileges were completed. A paper from IBM was taken as a framework for the privilege section. During the meeting a few operations were identified as necessary, although the list is far from complete: getpriv, setpriv, enable/disable_priv, droppriv.

Another issue brought to the whole group was Internationalization, and the decision was not to address it as long as they can. This is unfortunate, as the charter of POSIX is to be as international as possible. The 1003.1 committee learned the hard way that internationalization cannot just be stapled on later. It must be in there from day one or it becomes extremely difficult to make it work. In the case of security, labeling is an area in which internationalization is a must. If it is not placed in there initially, it may never get in.

The upshot of all this is that the small groups produced the guidelines for the next meeting and the topics that are going to be covered in the near future.

This group has targeted mid-1990 for a complete draft ready to ballot. The USENIX Standards Watchdog Committee contact for this group is Anna Maria de Alvaré. She can be reached at:

Anna Maria de Alvaré
Lawrence Livermore National Lab
L-303
PO Box 808

Livermore, CA 94450
+1 (415) 422-7007
annamaria@lll-lcc.llnl.gov
uunet!lll-lcc.llnl.gov!annamaria

1003.7 – System Administration

This new working group met as a Birds of a Feather session during the Hawaii meeting. During that session the group convenor outlined the goals and solicited input from the attendees. At a subsequent meeting in Monterey (in conjunction with the USENIX Large System Administration Workshop) the group took the input from that meeting and the work that had been going on off line and began producing a draft document.

So, what is the purpose of this body? To define a portable user interface for System Administration Utilities which would allow users to administer systems in a portable way, and allow developers to build system administration tools on top of consistent underlying commands and libraries. Since the work of this body will overlap with almost every other P1003 working group (and possibly other groups outside of POSIX), coordination is a major part of the standard development effort. Also, because the charter of this group is so broad (what is an administrative tool, anyway?), it is going to take quite a while to complete the standard.

Just to give you a rough idea of what is going to covered by this group, here are some possible areas: machine startup, process management, network, software licensing management, user management, password management, etc... At the meeting in Hawaii it quickly became apparent that the scope of this group is too large to accomplish anything in a reasonable period of time. Some of the time at the Monterey meeting was spent narrowing the scope of the group to a more manageable size. The group tried to identify items which could form a basic set of libraries and commands, and could be finalized in a two to three year time frame. After the initial standard is released, there may be continuing work into areas that the first cut was not able to address.

When I last wrote about this group, I was very critical of its charter and the possibility of it succeeding. I think it only fair to relate that
a number of people wrote me and said that I was too judgemental, and that I should take a wait and see attitude. Bowing to the will of the people, I am not going to draw any conclusions about the working group at this time. In the interim, if you want more information, or would like to share your opinions with me, drop me a line.

The USENIX Standards Watchdog Committee contact on 1003.7 is Mark Colburn. He can be reached at:
Mark Colburn
NAPS International
117 Mackubin St., Suite 1
St. Paul, MN 55102
(612) 224-9108
mark@naps.mn.org

1003.8 – Networking Extensions to POSIX

IEEE P1003.8’s charter (not yet formally approved by IEEE, but pending) is to help develop an IEEE POSIX networking standard. This was the committee’s first formal meeting, and it was devoted mostly to organizational matters, particularly on setting specific technical goals and how to divide the work into subcommittees.

This working group has emerged out of the work done by the /usr/group Technical Committee’s subcommittee on networking. Once this committee has been formally formed, the /usr/group networking committee will be considered to merge with the P1003.8 committee, and meet concurrently whenever P1003.8 does. Ultimately, the /usr/group committee is likely to disband completely in favor of P1003.8.

The charter (“project authorization request,” or PAR) was reviewed briefly:

SCOPE

1. Define Network Services required by portable applications consistent with existing and emerging standards such as OSI.

2. Define interfaces to the network services defined above, and where possible, language and protocol independent programming interfaces.

3. Identify the requirements for new network services & protocols and liaison with appropriate standards bodies (national and international) and interested organizations when appropriate.

PURPOSE

Define and/or adopt a set of paradigms to permit the implementation of portable applications in a network environment.

Areas to be addressed by this committee include:

1. Interoperability between POSIX applications and non-POSIX applications.

2. Bindings to OSI application layer services.

3. Identification of language requirements not appropriate to applications portability, and liaison with appropriate standards bodies to ensure that action is taken where appropriate.

4. The evaluation and definitions, where require, of binding to lower layer OSI services.

5. The requirements to ensure interoperability among POSIX-based distributed applications and services.

6. Network Services profile definitions for portable applications (POSIX).

Subcommittee Organization

A poll was held to find out what the most important topics were as far as the group was concerned. These turned out to be: process to communication, directory services, network management, transparent file access, and remote procedure call. Three main subcommittees were formed to look at some of these tasks. Roughly, these committees were “interprocess communication,” “remote procedure call,” and “transparent file access.”

Directory services and network management were recognized as important, but also as cutting across other functional areas. Also, it was noted that there were not enough people in attendance with sufficient expertise in these topics to form a useful body of opinion on proposals in these areas.

Transaction processing was generally felt to be within the domain of the committee, but as a special case of remote procedure call. It was noted that others who were not on the committee may feel otherwise.
The committee split up into subcommittees for a day to refine the definitions of the most important end products identified for the committee to concentrate on.

Specific Technical Goals

The following is a summary of what the committee as a whole agreed on as a result of the input from the individual subcommittees.

* Transparent File Access

It was decided that the products should be client-only interfaces. Three products were identified:

1. Full POSIX-semantic transparent file access interface. This would include previous /usr/group DFS Committee work on DFS (distributed file system).

2. Administrative interface to support (1).

3. Subset-semantic transparent file access interfaces. This could be vendor (e.g., MS-DOS, Apple, etc.) or protocol (e.g., FTAM) specific.

Work items identified so far include:

1. Definition of file operations

2. Liaison to system administration; definitions of transparent file access specific system administrative utilities and/or interfaces

3. Liaison with directory working group

4. “Appropriate approach” to the protocol invention problem

This group expects to finish relatively quickly (6 months or so was the estimate given), because it was felt that a significant amount of the work needed to produce standards in this area is already done by definition (the P1003.1 standard).

* Remote Procedure Call:

The RPC folks apparently did not define their charter so much as identify issues that need to be addressed. The following was their list of issues along with tentative resolutions (if any):

1. Level of service

2. POSIX-to-POSIX versus POSIX-to-other (address POSIX-to-other)

3. Language binding (initial target: C)

4. TP support

5. Connection re-use

6. Call-back/recursion

7. Compiler language

8. Data canonicalization

9. Authentication

10. Our scope versus X.500

11. Standard suite of services need to confer with X3T5 on possible charter issues

12. Idempotency – execute once only guaranteed

13. Long running processes – keepalive & timeouts probably needed

14. Crash recovery

15. Real Time issues – no real time interface

16. Directory services

17. Multiple protocol stacks

The subgroup chose not to identify the next step in the process (apparently meaning that they will wait for proposals).

* Interprocess Communication:

Four products were identified:

1. Simple Protocol-Independent Network Interface

Features:

- Bidirectional byte stream virtual circuits
- Connectionless message exchange
- Read/write support
- Protocol-independent naming
- Asynchronous communication services
- Support for both client and server processes
- POSIX-to-non-POSIX support

Issues:

- How to resolve names in a protocol-independent manner?
- What should the individual functions look like?
2. Simple Structured Data Network Interface Features:
All of (1), with extensions for data description and machine-independent representation.

- Description of the syntactic structure of the data; when you send the data, you reference the structure.

- Not all functions from (1) may work (such as, read/write)

Issues:
- Structure alternatives: ASN.1, ...
- C data structures (stub compilers)

3. Protocol-Option-Extended Network Interface Features:
- Provides the ability to access protocol dependent options
- Migration path to potential future protocols
- POSIX-to-any
- Virtual circuits, datagrams

Issues:
- Limited lifespan (?)
- Limited utility
- Usefulness as a migration tool
- Relationship to (1)

4. OSI application level interface Features:
- A family of interfaces with consistent style and syntax which provides OSI application level services, e.g. FTAM, VT, ACSE, ROSE.

Issues:
- Complexity
- Prioritization (which ones to focus on first)

One issue that surfaced very quickly in the network IPC discussions was the differences and relative merits of sockets and XTI. Some went as far as to say that the differences were significant enough to guarantee "religious wars" over the issue, and/or make any kind of progress impossible in the area of product (3).

Whatever the cause, a majority (8/0/3/3) of participants expressed interest in working on product (1), with products (3) and (4) having a relatively weak level of interest.

The committee will get down to serious business at the next meeting (in January; 5 days). For the next meeting, proposals are being solicited in all areas. The USENIX Standards Watchdog Committee contact on this committee is Stephen Head. He can be reached at:

Stephen Head
Hewlett Packard
263 Mackintosh St.
Fremont, CA 94539
+1 (408) 447-2740
smh@hpda.hp.com
uunet!hpda!smh

That's about it for this quarter. As always, if you have any comments or suggestions, please contact me at:

Shane P. McCarron
NAPS International
Suite 6
117 Mackubin St.
St. Paul, MN 55102
+1 (612) 224-9239
ahby@bungia.mn.org
uunet!bungia.mn.org!ahby
Letter to the Editor

21 December 1988

First, I'd like to make it clear that I support the Association's desire to publish periodically a report on UNIX-related standardization activities. I also respect Shane McCarron and value his opinions on controversial issues in this area. However, I feel that because of the increasingly editorial nature of the Updates, they are no longer serving their intended purpose.

The Updates should report on the accomplishments of the various national and international standards bodies and any other relevant standardization developments or issues that affect the UNIXs community. Controversial issues should be covered impartially, giving both sides equal time and refraining from the pedantic or preachy.

Mr. McCarron's most recent report on the National Institute of Standards and Technology (NIST, formerly National Bureau of Standards) begins with a short update on the POSIX FIPS but quickly becomes a criticism of NIST's practice of writing FIPS before the associated POSIX standard has been approved. He even goes so far as to say "This interesting turn of events proves that in their previous heinous acts they were just being nice." This is uncalled-for. NIST's actions are at worst misguided. To his credit, McCarron says NIST is being given the opportunity to rebut these comments. Unfortunately, McCarron presumes himself capable of predicting NIST's response and proceeds to attack the rebuttal before it has even been made.

There are other examples, but I won't bother listing them. My point is that I don't think the Association should continue to fund McCarron if he can't or won't cease using the Updates as a personal soapbox. If the Association values his opinions enough, he should be commissioned to write a separate editorial column.

David Sill
UNIX System Programmer/Analyst
Naval Surface Warfare Center
Dahlgren, VA 22448
dsill@relay-nswc.navy.mil

Further comments (not flames) concerning these reports may be sent to (usenix,uunet)!peter.
The EUUG

The EUUG and USENIX have agreed to give space in their respective newsletters so that their members can be made more aware of what is happening on the other side of the Atlantic.

Donnaly Frey has already started to fill her column in the EUUG Newsletter, informing us of what is happening with USENIX.

Since this is the first article to be published in the EUUG column of \textit{login}, it occurred to us that there are probably lots of people who don’t know exactly what the EUUG is (this is certainly true in Europe, so I think it’s a good guess that the same must be true in the US).

So, here is a quick history of the EUUG, and a description of what it is today, and what its objectives are.

\textit{History}

The EUUG began life not as an European organisation, but simply as a DECUS SIG in England. As time progressed, it became obvious that UNIX could (and did) run on machines other than PDP-11s.

Many of the people interested in UNIX had no relation to DECUS, and many did not want any such relationship. DEC was not entirely happy about supporting a SIG whose primary purpose was promoting the use of a non DEC operating system. It became obvious that the SIG could not continue in its present form, and that a split would be made from DECUS.

The result of this split was the formation of the UNIX User Group (UUG).

This was a radical decision, because without the logistical and financial support that DEC gives to its SIGs people had to start paying membership fees. There were many people who doubted that the UUG would last. In one way they were right, the UUG did not last very long. There were more and more people coming to the meetings from outside the United Kingdom – especially from Holland and Denmark, but also from Germany.

The Dutch members founded their own UNIX group, and it became obvious that other countries were interested in doing likewise. It also became obvious that the UUG could not remain a British organisation, so, at a meeting of the UUG in 1980 at Heriot Watt University (in Scotland) the UUG was transformed into the European Unix User Group (EUUG).

Sometime later, the EUUG had to again change name, following some not too subtle hints from you-know-who, to the European Unix systems User Group. Since the EUUG and its logo were reasonably well known by this time, it was decided to keep the name EUUG, and only mention “systems” on printed documents.

One of the first decisions taken by the newly formed EUUG was the promotion of what were at the time called “local groups.” These local groups were the beginnings of the national groups which exist today.

The national groups exist to promote UNIX in their respective countries. This is a task which would be difficult for the EUUG to do, given its limited resources. The national groups operate more or less independently of the EUUG, except that they are all members of the EUUG “federation.” The national groups provide services to their members which are specific to their national needs (national language meetings, national product catalogues, working groups, and exhibitions, for example).

The EUUG serves as a cohesive force for these groups, and organises things which are better tackled at a European level than at a national level. Examples are the EUUG conferences, and EUnet, of which more will be said later.

As other European countries formed national user groups, most affiliated with the EUUG; the only notable exceptions were the Swiss, who prefer to maintain their famous neutrality.

The current members are:
introductory courses to advanced and highly technical tutorials on some specific aspect of UNIX.

EUnet

The network (EUnet) is somewhat more formally organised than is the case in the USA.

Each country has one national backbone node handling almost all international traffic. There is one international backbone (mcvax) which connects this backbone network with other continents. This structure is very much determined by the high international telecommunications tariffs.

Because of the volume of traffic, the backbone links are in the process of being replaced with leased lines, to replace the existing X.25 links. Each country manages its own part of the network, some in close collaboration with the national user group, others with somewhat less contact between the two.

The only rules governing who can connect are:

1. They must be a member of the EUUG (or affiliated group).
2. They must refrain from commercial exploitation of the net.
3. They must pay their fair share of the running costs.

As you can imagine, much time and effort is spent working on policies to implement the third point!

The EUUG Newsletter

The EUUGN is published 4 times per year. At present, it seems to have reached a form which satisfies most people, acting as both a newsletter to inform members what each of the national groups is doing, giving reports on conferences (USENIX for example), and as a technical journal.

Although it currently seems to satisfy most people, we are still exploring new methods of improving its acceptability. For example, we are currently experimenting with bilingual articles. People can send articles for publication in their own language. We will translate these, and print the article in double column format,
one column being the original text, and the column opposite being the English translation.

We also carry (limited) advertising to help offset the costs of production.

Future input to ;login:

Rather than try to describe each national group, and what it is doing, future articles in this column will come from these groups, each taking its turn to introduce itself, and explain what it is doing.

Contact points

If you have any questions or suggestions, feel free to contact us. Some useful postal and e-mail addresses follow.

Subscriptions and general questions:

EUG
Owles Hall
Buntingford Herts. SG9 9PL ENGLAND
Tel. +44 763 73039
Fax. +44 763 73255
euug@inset.co.uk

Newsletter (article submission etc.):

Alain Williams (EUGN editor)
Parliament Hill Computers Ltd.
7 Prospect Street
Caversham Berkshire RG4 8JB ENGLAND
addw@phcomp.co.uk

EUnet – General information:

Daniel Karrenburg
dfk@cwi.nl

Philip Peake
EUG publications executive
philip@axis.fr

---

EUUG Spring ’89 Conference

Brussels, Belgium
April 3-7, 1989

The BUUG will host the Spring ’89 European UNIX systems User Group Technical Conference in Brussels. Technical tutorials on UNIX and closely related subjects will be held on Monday and Tuesday, followed by the three day conference with commercial exhibitions.

If you wish to receive a personal copy of further information about this, and future EUUG events, please contact the Secretariat.

Secretariat

EUG
Owles Hall
Owles Lane
Buntingford
Herts, SG9 9PL, UK

Phone: +44 763 73039
Fax: +44 763 73255
Telex: 928476 ISTECH G
Email: euug@inset.uucp

Tutorial Officer

Neil Todd
IST
60 Albert Court
Prince Consort Road
London, SW7 2BH, UK

+44 1 581 8155
+44 1 581 5147
neil@ist.co.uk

Programme Chair

Prof. Marc Nyssen
Medical Informaticas Dept.
Vrije Universiteit Brussel
Laarbeeklaan 103
B-1090 Jette Belgium
+32 2 477 44 24
+32 2 477 40 00
marc@minf.vub.uucp

Volume 14, Number 1 January/February 1989 47
Future Events

Workshop on Software Management  
New Orleans, Apr. 3-4, 1989  
See page 3.

EUUG Spring Conference  
Brussels, Apr. 3-7, 1989  
See page 47.

Workshop on UNIX Transaction  
Processing, Pittsburgh, May 1-2, 1989  
See page 4.

USENIX 1989 Summer Conference and  
Exhibition, Baltimore, Jun. 12-16, 1989  
See page 5.

Distributed Processing Workshop  
Fort Lauderdale, Oct., 5-6, 1989

Graphics Workshop V,  
Monterey, Nov. 16-17, 1989

Long-term USENIX & EUUG Schedule  
Sep 18-22 '89 Vienna, Austria  
Jan 22-26 '90 Omni Shoreham, Washington, DC  
Apr 23-27 '90 Munich, W. Germany  
Jun 11-15 '90 Marriott Hotel, Anaheim  
Jan 21-25 '91 Grand Kempinski, Dallas  
Jun 10-14 '91 Opryland, Nashville  
Jan 20-24 '92 Hilton Square, San Francisco  
Jun 8-12 '92 Marriott, San Antonio

Publications Available

The following publications are available from the Association Office. Prices and overseas postage charges are per copy. California residents please add applicable sales tax. Payment must be enclosed with the order and must be in US dollars payable on a US bank.

Conference and Workshop Proceedings

<table>
<thead>
<tr>
<th>Meeting</th>
<th>Location</th>
<th>Date</th>
<th>Price</th>
<th>Overseas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Installation Systems Admin. Workshop</td>
<td>Monterey</td>
<td>Nov. '88</td>
<td>$ 8</td>
<td>$ 7</td>
</tr>
<tr>
<td>C++ Conference</td>
<td>Denver</td>
<td>Oct. '88</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>UNIX and Supercomputers Workshop</td>
<td>Pittsburgh</td>
<td>Sep. '88</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>UNIX Security Workshop</td>
<td>Portland</td>
<td>Aug. '88</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>USENIX Conference</td>
<td>San Francisco</td>
<td>Jun. '88</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>C++ Workshop</td>
<td>Santa Fe</td>
<td>Nov. '87</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>Graphics Workshop IV</td>
<td>Cambridge</td>
<td>Oct. '87</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>USENIX Conference</td>
<td>Washington DC</td>
<td>Jan. '87</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Graphics Workshop III</td>
<td>Monterey</td>
<td>Dec. '86</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

The EUUG Newsletter, which is published four times a year, is available for $4 per copy or $16 for a full-year subscription.

We hope to have EUUG tapes and conference proceedings available shortly.

EUUG Proceedings for Spring 1988 (London) and Fall 1988 (Portugal) are available in limited numbers to North American customers at $40 per copy.
Long-Term Calendar of UNIX Events†

1989 Jan 9-13
1989 Jan 17
1989 Jan 30-Feb 3
1989 Feb
1989 Feb 28-Mar 3
1989 Feb 28-Mar 3
1989 Apr 3-4
1989 Apr 3-7
1989 Apr 10-11
1989 Apr 24-28
1989 May 1-2
1989 May 8-12
1989 May 14-16
1989 May 16
1989 May
1989 Jun
1989 Jun 12-16
1989 Jul
1989 Jul 10-14
1989 Sep
1989 Sep 18-22
1989 Oct 5-6
1989 Oct 16-20
1989 Nov 1-3
1989 Nov 6-10
1989 Nov 16-17
1989 Nov
1989 Dec
1990 Jan 22-26
1990 Jan 23-26
1990 Jan 29
1990 Feb
1990 Apr
1990 Apr 23-27
1990 May 7-11
1990 May
1990 Jun 11-15
1990 Autumn
1991 Jan 21-25
1991 Jan 22-25
1991 Jun 10-14

IEEE 1003
Terminal Int. Ext. and Net. Serv.
USENIX
UNIX in Government
UNIX Convention
UniForum
*Software Management Workshop
EUUG
ANSI X3J11
IEEE 1003
*Transaction Processing Workshop
DECUS
AMIX
POSIX Application Workshop
UNIX 8x/etc
NZSUGI
USENIX
JUS 13
IEEE 1003
*Large Systems Admin. Workshop
EUUG
*Distributed Systems Workshop
IEEE 1003
UNIX Expo
DECUS
*Graphics Workshop V
JUS 14
JUS UNIX Fair
USENIX
UniForum
IEEE 1003
UNIX in Government
IEEE 1003
EUUG
DECUS
UNIX 8x/etc
USENIX
EUUG
USENIX
USENIX
Embassy Suites, Ft. Lauderdale, FL
NIST; MD
Town and Country, San Diego, CA
Ottawa, Ont.
AFUU; Paris, France
Moscone Center, San Francisco, CA
New Orleans Hilton, New Orleans, LA
Palais des Congres, Brussels, Belgium
Phoenix, AZ
Minneapolis-St. Paul, MN
Pittsburgh Hilton, Pittsburgh, PA
Atlanta, GA
Israel
NIST; MD
/usr/group/cdn; Toronto, Ont.
New Zealand
Hyatt Regency, Baltimore, MD
Toyko, Japan
San Francisco, CA
Vienna, Austria
Marriott Marina, Ft. Lauderdale, FL
Brussels (or Amsterdam)?
New York, NY
Anaheim, CA
DoubleTree Inn, Monterey, CA
Osaka or Kobe, Japan
Toyko, Japan
Omni Shoreham, Washington, DC
Washington Hilton, Washington, DC
New Orleans, LA
Ottawa, Ont.
Montreal, Que.
Munich, Germany (tentative)
New Orleans, LA
/usr/group/cdn; Toronto, Ont.
Marriott Hotel, Anaheim, CA
south of France
Grand Kempinski, Dallas, TX
Infomart, Dallas, TX
Opryland, Nashville, TN

† Partially plagiarized from John S. Quarterman by PHS.
* USENIX Workshops
Large Systems Administration Workshop

There will be a third Large Systems Administration Workshop, most likely in early September. It will again be chaired by Alix Vasilatos, uunet/osf.org/alix. A full announcement will appear in the next .login: and on comp.org.usenix.

-PHS

New Release of 2.10 BSD Available

The second release of 2.10BSD is finally available! It has been designated 2.10.1. Although the changes are fairly simple to describe, they cover large portions of the distribution. Most will not be visible to either users or administrators; specifically, no recompilation is necessary. Administrators should be aware that the 4.3BSD disk quota system is now available. Due to address space considerations, however, it is expensive to run. Also, the source for the on-line manual pages has been rearranged as per the 4.3BSD-tahoe release.

The major change, and the reason for the second release, is an extensive reworking of the kernel to move the networking into supervisor space. This move eliminated most, if not all, of the instabilities seen in the original networking provided with 2.10BSD; it also doubled the speed of, for example, file transfer. As encouragement to sites that encountered difficulties in using the networking in the first release, or encounter difficulties in this release, we have beta sites that have been running for months without crashing, as well as sites with fifty nodes. We are, however, still suspicious of the DEQNA driver...

In application land, many missing pieces of the 4BSD distribution have been added, most notably the FORTRAN compiler and library and the line printer sub-system. Many other programs have had minor (and not-so-minor) fixes applied.

Keith Bostic
Casey Leedom

Because the changes to the kernel are major, no "upgrade" tape will be available. 2.10.1 BSD is only available as source, to appropriate licensees of V7, System III, System V, or 2.9BSD. The cost is $200, prepaid.

The release consists of two 2400 foot, 1600 BPI tapes (approximately 80Mb) and approximately 100 pages of documentation. If you require 800 BPI tapes, please contact USENIX for more information.

If you have questions about the distribution of the release, please contact USENIX at:

2.10BSD
USENIX Association
PO Box 2299
Berkeley, CA 94710
+1 415 528-8649
(uunet,ucbvax)!usenix!office

If you have technical questions about the release, please contact Keith Bostic at:

(ucbvax,seismo)!keith
keith@okeeffe.berkeley.edu
+1 415 642-4948

NOTE: There are a few copies of 2.9BSD available. If you do not have split l&d and want to run UNIX on your PDP-11/x, write the USENIX office.

- PHS
MEMBERSHIP RENEWAL FORM

USENIX Association
P.O. Box 2299
2560 Ninth St., Suite 215
Berkeley, CA 94710
415 528-8649
{ucbvax,unet}lrsenix|office

Name ___________________________________________

Address _________________________________________

______________________________________________

Your membership in the USENIX Association expires on December 31, 1988. To renew for 1989, please send your payment soon.

☐ Individual: $40.00  ☐ Corporate: $275

☐ Student: $15.00 (enclose a copy of ID)  ☐ Educational: $125

☐ Check enclosed payable to USENIX Association.

☐ Please charge my ☐ Visa ☐ Mastercard

Account # ___________________________ Exp. Date ____________

Signature _______________________________________

Please return this notice with your payment for prompt service. If you have already prepaid, please accept our thanks and disregard this notice.

Change of Address? Please indicate above.

Overseas? Please make your payment in U.S. currency by one of the following:
• Charge (Visa, MasterCard, or foreign equivalent)
• International postal money order
• Check - issued by a local branch of a U.S. Bank

Don't miss the first 1989 issue of Computing Systems!
Calls for Papers

Enhancing the 4.3 BSD UNIX Serial Line Interface

An Update on UNIX Standards Activities

New 2.10 BSD Release Available

Change of Address Form

Please fill out and send the following form through the U.S. mail to the Association Office at the address above.

Name: ___________________________ Member #: __________________

OLD: ______________________________

NEW: ______________________________

Phone: ____________________________

uucp: {uunet,ucbvax}!

______________________________

______________________________