EUUG

European UNIX† systems User Group

PROCEEDINGS

Autumn 1985 Conference

Copenhagen, Denmark
Bella Conference Centre
10 - 12 September 1985

Introduction

The EUUG autumn 1985 conference was a fair success, in most of its aspects. The technical programme was received very well, as were the tutorials and the industrial programme contents.

The conference proceedings contain extensive papers on topics covered by various authors, both of the technical and of one of the industrial sessions.

Thanks go to David Tilbrook (Imperial Software Technology) for his invaluable help in suggesting speakers and in pointing us into the right direction. Thanks go also to Kim Biel-Nielsen, who held the Industrial Programme Chair. Last but not least, we wish to thank UNIX Europe Ltd. for their support with respect to the several AT&T Bell Laboratories speakers. A conference program chairman can impossibly compose a good EUUG conference programme without being able to invite some speakers to tell us about the action near the very roots of the UNIX system.

For Nigel Martin (The Instruction Set, Ltd) and myself - we both shared the program chair of this conference - it was a success that all complaints went to the food.

Hendrik-Jan Thomassen
AT Computing b.v.
Nijmegen, the Netherlands

Table of Contents

no papers have been received of the following lectures;

An improved UNIX Kermit file transfer program
Johan Helsingius, Helsinki University of Technology,
Helsinki, SF.
co-author: Barbro Malm (Helsinki Univ. of Techn.)

Smalltalk on UNIX Systems
Georg A.M. Heeg, Universitaet Dortmund,
Dortmund, D.

Nottinghams experience of X.25 under 4.2BSD
William Armitage, University of Nottingham,
Nottingham, GB.

Standardised Art as a Vehicle for Enhancing Market Penetration
Mike O'Dell, Group L Corp.,
Herndon, Va., USA.

AWK as a General-Purpose Programming Language

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

Awk is a pattern-action language with convenient facilities for processing strings and numbers. It was originally intended for simple data validation, report generation and data transformation tasks; *awk* programs for such tasks can often be expressed in only a few lines of code.

Awk has often been pressed into service in ways far beyond what was originally intended. Recent changes to *awk* make it more suitable as a general-purpose programming language. This paper will discuss new features — new built-in functions, multi-file input, dynamic regular expressions and text substitution, and user-definable functions — and illustrate some interesting applications that they allow.

1. Introduction

The *awk* language was originally designed for specifying simple tasks in data validation, transformation and report generation. For example, this *awk* program prints each line of its input in which either the first or second field is negative:

```
$1 < 0 || $2 < 0 { print "Line", NR, "has a negative value", $0 }
```

This program replaces each second field by its logarithm:

```
{ $2 = log($2); print }
```

This program prints the sum and average of the values in the first field:

```
        { sum += $1 }
END     { if (NR > 0) print sum, NR, sum/NR }
```

An *awk* program consists of one or more pattern-action statements. The basic cycle is to scan each line of input, comparing it in turn to each pattern. If the pattern matches, the action is performed. If there is no pattern, the action is performed on all lines, as in the second and third examples above; if there is no action, the selected lines are printed.

Awk automatically scans each input file and breaks each input line into fields (referred to as `$1`, `$2`, etc.). `$0` is the whole record, and the variable `NR` is the input record number.

Awk programs may use variables, which take on numeric or string values according to context; there are also associative arrays in which subscripts may be arbitrary values. Variables may be combined in expressions using most of the same operators as in C, plus string concatenation. Patterns may include relational expressions on strings and numbers, and regular expressions.

Actions may include control flow statements like those in C (`if-else`, `while`, `for`), and calls of built-in functions for standard arithmetic operations (e.g., `log`), string length, substring, etc.

Awk programs are often substantially shorter than equivalent programs in a language like C, because most of the trappings of conventional languages — declarations, initialization, input scanning and parsing, type conversions, etc. — are handled implicitly. For example, this program uses

an associative array in which subscripts are entire input lines to count the distinct lines of input and print them, preceded by their count, in decreasing order of frequency.

```
        { x[$0]++ }
END     { for (i in x) print x[i], i | "sort -nr" }
```

(The lines are sorted by piping the output to the standard `sort` program.) Furthermore, *awk* is implemented as an interpreter; since there is no compiled form of an *awk* program, there are no object files and no loader, so housekeeping is simpler than it would be for a compiled language.

Although *awk* was intended for very short programs like those above, it has been used for much larger ones, often running well over one thousand lines. Examples known to the author include assemblers for several microprocessors and a variety of simple database management systems. Most of these programs are not described in the literature; one exception is Comer's Flat File Generator [1], which is a collection of *awk* and shell programs.

There are problems with using the original version [2] of *awk* for such large programs. One difficulty is efficiency; both compilation and execution can be quite slow. The other problem, and the subject of this paper, is the lack of certain crucial functionality: access to command-line arguments, control of input and output files, and user-definable functions. The current version [3] of *awk* addresses these issues, and the combination of new features and better understanding has led to some interesting applications. The next section describes new language components; the following section sketches applications.

2. New Facilities

Built-in functions

Trivial enhancements include the addition of built-in functions for `sin`, `cos`, `atan2`, `rand`, `srand` (to set the seed for `rand`), and `system(`*string*`)`, which executes the command given by *string*.

One significant complaint about the original *awk* was that it ran out of open file descriptors because there was no way to close a file once opened. That defect is remedied by the obvious `close` function.

Multiple simultaneous input files

The original *awk* processed each input file from beginning to end in order, one at a time. The new version provides a function `getline` that permits arbitrary input from arbitrary files. It comes in several flavors:

`getline`	sets `$0` from next input line
`getline x`	sets `x` from next input line
`getline <"file"`	sets `$0` from next line of `file`
`getline x <"file"`	sets `x` from next line of `file`
`"command" | getline`	is like `getline <"file"`
`"command" | getline x`	is like `getline x <"file"`

`getline` makes it possible to read several input files at once, and to write loops like this one for counting users:

```
BEGIN {
        while ("who" | getline)
                n++
        print n
        exit
}
```

`getline` returns 1 if there was data, 0 for end of file, and —1 for errors.

Command-line arguments

In the original version of *awk*, there was no access to the values of the command-line arguments, which were deemed to be filenames for input and nothing else. Naturally, this restricted the ways in which arguments could be used with *awk* programs, often forcing some remarkable evasive maneuvers with the shell and quoting, as in this example from Reference [4]:

```
awk '
        BEGIN { split("'"`date`"'", date) }
        $1 == date[2] && $2 == date[3]
' $HOME/calendar
```

In the current version, the variable **ARGC** contains the argument count and **ARGV** is a vector of arguments, just as in a C program. If the values of **ARGC** and **ARGV** are left untouched, the behavior is exactly as before. But if any argument is set to null, or if **ARGC** is changed, the corresponding arguments will not be treated as filenames. Thus, the program above could be re-written as

```
awk '
        BEGIN { mon = ARGV[3]; day = ARGV[4]
                for (i=2; i < ARGC; i++) ARGV[i] = "" }
        $1 == mon && $2 == day
' $HOME/calendar  date
```

Of course, an even better solution would be something like

```
awk '
        BEGIN { "date" | getline d; split(d, date) }
        $1 == date[2] && $2 == date[3]
' $HOME/calendar
```

Dynamic regular expressions

Regular expressions in the original version of *awk* were compile-time constants enclosed in slashes, as in

```
/regexp/
```

In the current version, any string, whether constant or variables, may be used as a regular expression in any context where static one can be used. Two functions, **sub** and **gsub**, perform text substitutions for patterns specified by regular expressions:

sub(*regexp, replacement, target***)**

replaces the first occurrence of *regexp* in *target* by *replacement*; **gsub** replaces all occurrences. For example, the following program generates form letters, using a template stored in a file called **form.letter**:

```
This is a form letter.
The first field is #1, the second #2, the third #3.
The third is #3, second is #2, and first is #1.
```

and replacement text of this form:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The **BEGIN** action stores the template in the array **template**; the remaining action cycles through the input data, using **gsub** to replace template fields of the form ***#n*** with the corresponding data fields.

```
BEGIN { FS = "|"
        while (getline <"form.letter")
                line[++n] = $0
}
{       for (i = 1; i <= n; i++) {
                s = line[i]
                for (j = 1; j <= NF; j++)
                        gsub("#"j, $j, s)
                print s
        }
}
```

The heart of the procedure is the first argument to **gsub**, which is a dynamic regular expression formed by concatenating a # and the field number.

One by-product of extending the regular expression mechanism is that the field separator, previously limited to a single character, can now be set to any regular expression. For example, the following assignment sets the field separator to match an optional comma followed by blanks or tabs:

```
FS = "(,[ \t]*)|([ \t]+)"
```

Although useful, these facilities are not enough to make *awk* a serious competitor for *sed* when efficient text editing is necessary.

User-definable functions

The major limitation in the original *awk* was the lack of any way for a user to define his or her own functions. The new version provides them with the following syntax:

`function` *name*`(`*argument list*`) {` *body* `}`

A function definition can occur anywhere a pattern-action statement can.

Scalar arguments are passed by value; array arguments are passed by reference. Arguments are local to the function, but *all* other variables have global scope. Any argument not explicitly provided by the caller is null, so in a rather *ad hoc* way, excess formal parameters can be used as local variables.

As in C, a `return` statement exits from a function, optionally returning a value.

The function mechanism is not terribly efficient, but it is at least acceptable, as shown in this table of run times for computing Ackermann's function Ack(3,3), which requires 2432 nested function calls:

```
func ack(m, n) {
        if (m == 0) return n+1
        if (n == 0) return ack(m-1, 1)
        return ack(m-1, ack(m, n-1))
}
```

C	<0.1 sec.
bas	1.3
hoc	5.5
awk	10.9
bc	39.7

All times are from a DEC VAX-11/750.

3. Applications

Algorithm and data structure presentation

Bentley has used *awk* extensively in his *Programming Pearls* columns for presenting algorithms and data structures. *Awk* programs are compact and typeless, so the presentation can focus on the essential ideas without getting buried in irrelevant details. For example, a column on testing programs [5] presents a variety of searching and sorting algorithms, plus a framework of testing routines, all written in *awk*.

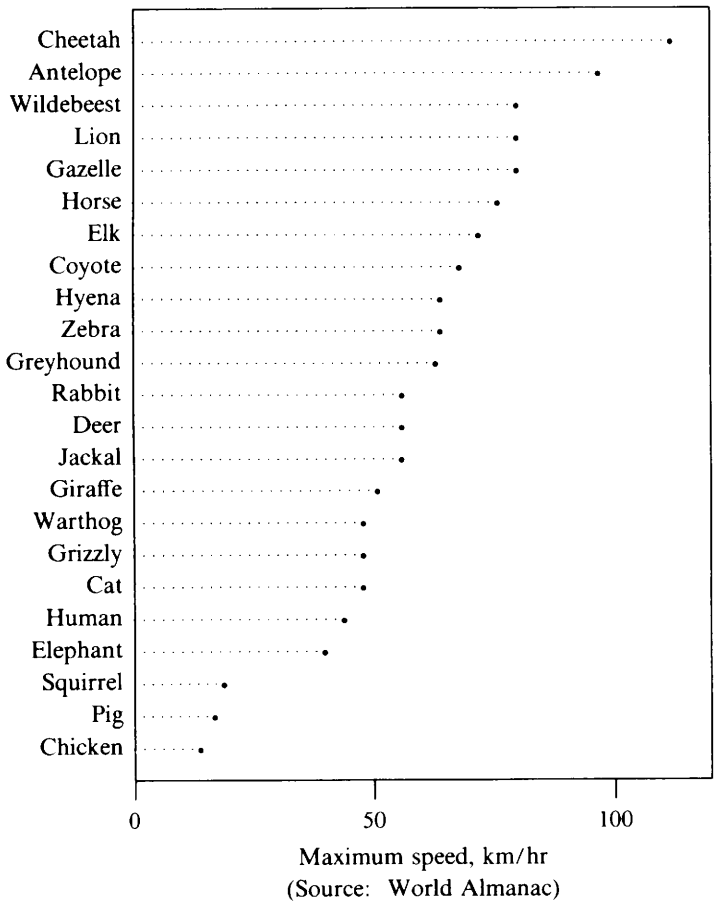
Another column [6] uses *awk* to demonstrate how certain algorithms can be expressed with remarkable brevity and clarity when associative arrays are used. (The line-frequency count program shown earlier in this paper is a simple example.) This code maintains a graph specified as predecessor-successor pairs, and upon request prints all nodes reachable from a specified one, using recursive depth-first search:

```
function visit(node,    i) {    # i is a local variable
        if (visited[node] == 0) {
                visited[node] = 1
                print "   " node
                for (i = 1; i <= succct[node]; i++)
                        visit(succlist[node "," i])
        }
}
$1 == "reach" {
        for (i in succct)
                visited[i] = 0
        visit($2)
}
$1 != "reach" {
        succlist[$1 "," ++succct[$1]] = $2
        if (!($2 in succct))
                succct[$2] = 0
}
```

This example uses associative arrays to simulate several data structures. The subscripts of the array `succct` are the nodes of the graph; the values are the number of successors. The array `succlist` contains the successors of each node, encoded as subscripts of the form `p,n`, where `succlist[p,n]` is the `n`-th successor of node `p`. In other words, a 1-dimensional associative array represents a sparse 2-dimensional array. Functions are also needed; by the nature of the depth-first search algorithm, they must be recursive, with local variables.

Compiling little languages

A dotchart is a form of statistical display for data that consists of a name and a value, as in this plot of animals and their maximum speed in kilometers per hour.



The dotchart is described with a very simple language; this picture comes from

```
file "animal.data"
label "Maximum speed, km/hr" "(Source:  World Almanac)"
coord x 0 to 120
```

where the file `animal.data` contains lines of the form

```
112 Cheetah
97 Antelope
...
```

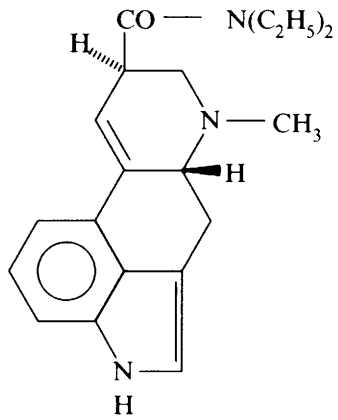
The dotchart "compiler" converts the specification into commands for the *grap* language [7], which in turn is converted into *pic* and then into *troff*.

The implementation of *dotchart* is about 25 lines of *awk*, 20 of which are `print` statements generating *grap* commands. Other statistical displays have been built with much the same flavor as the dotchart language; the most widely used is one for generating scatter-plot matrices [8].

A larger *awk* program (about 350 lines) is used to implement a specialized language for describing chemistry structures; the language was designed in collaboration with Jon Bentley and Lynn Jelinksi. The input language provides for describing a list of objects, each followed by attributes of size, direction, position, and substructure. For example, the LSD molecule is specified like this:

```
# Lysergic acid diethylamide
B:      benzene pointing right
F:      flatring5 pointing left put N at 5 double 3,4 with .V1 at B.V2
        H with .n at F.N.s
R:      ring pointing right with .V4 at B.V6
        front bond right from R.V6 ; H
R:      ring pointing right with .V2 at R.V6 put N at 1 double 3,4
        bond right from R.N.e ; CH3
        back bond -60 from R.V5 ; H
        bond up from R.V5 ; CO
        bond right ; N(C2H5)2
```

The *awk* program translates this description into *pic* commands, which, combined with some macro definitions and passed through *troff*, produce the following picture:



Awk has turned out to be a good implementation language for simple compilers. Implicit input, field-splitting and regular expressions handle parsing of input. Associative arrays provide both symbol tables and arbitrary data structures. Functions encapsulate basic operations. All storage management and type conversion is done implicitly. As a result, the programs are probably a factor of five smaller than they would be in C. (Honesty compels me to admit that they are at least an order of magnitude slower too.)

4. Conclusions

There have been some strains in keeping compatibility between old and new versions of *awk*, most notably the handling of command-line arguments, but most changes have gone in smoothly.

The new version is actually faster than the old one, largely because the detailed code is cleaner and simpler. Unfortunately, there does not seem to be any way to make the program a great deal faster without a complete revision, although one perennial option is to have *awk* generate C for subsequent compilation.

Larger programs require more emphasis on error reporting. The new version is much improved over the old in this area; the laconic "syntax error" has been replaced by reasonably accurate identification of the offending piece of code. For example, omitting the **1** from the line

```
for (i = 1; i <= succct[node]; i++)
```

in the program above produces the message

```
awk: syntax error at source line 5 in function visit
 context is
        for (i = >>>  ; <<<  i <= succct[node]; i++)
```

(Not all error messages are this precise, of course.)

The evolution of *awk* retains the useful attributes of the original while providing new features. In particular, it appears that the area of little languages is a fruitful one, and that *awk* is a good tool for prototyping and often for actual production use.

Acknowledgements

The new version of *awk* is a joint project with Al Aho and Peter Weinberger, the other members of the AWK Development Task Force. We are greatly indebted to Jon Bentley for many contributions, including interesting applications of *awk*, publicity, bug reports, and valuable suggestions on this paper.

References

References

1. D. E. Comer, "The Flat File System FFG: A Database System Consisting of Primitives," *Software—Practice and Experience* **12**(11), pp. 1069-80 (November 1982).
2. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "AWK — A Pattern Scanning and Processing Language," *Software—Practice and Experience* **9**, pp. 267-280 (April 1979).
3. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *AWK — A Pattern Scanning and Processing Language. Programmer's Manual,* AT&T Bell Laboratories CSTR 118 (May 1985).
4. B. W. Kernighan and R. Pike, *The Unix Programming Environment,* Prentice-Hall (1984).
5. J. L. Bentley, "Programming Pearls: Confessions of a Coder," *CACM* **28**(7), pp. 671-9 (July 1985).
6. J. L. Bentley, "Programming Pearls: Associative Arrays," *CACM* **28**(6), pp. 570-6 (June 1985).
7. J. L. Bentley and B. W. Kernighan, *GRAP — A Language for Typesetting Graphs*, To appear.
8. B. W. Kernighan, "Recent Work in Unix Document Preparation Tools," *Proc. EUUG*, Copenhagen (September, 1985).

The STREAMS Facility in System V

Bob Duncanson
Senior Software Consultant
Unix Europe Limited

STREAMS is a set of mechanisms within the UNIX system kernel that allow character I/O to be implemented in a modular way. As a result, the drivers for character-type devices have a structured interface to the UNIX system kernel. The STREAMS mechanisms also provide a flexible framework within which different networking architectures can easily be designed and implemented.

STREAMS includes a conceptual model of character I/O, a well-defined interface with the rest of UNIX System V, a library of utility programs, other utilities (such as an administration driver) and several new (or modified) system calls.

1. Introduction

UNIX * System V is designed to provide comprehensive support for networking services. This paper describes a major building block of that support. Although this paper describes STREAMS in some detail, no assurance is given that System V Release 3 conforms to this description.

The original streams concepts were developed in the Information Sciences Research Division of AT&T Bell Laboratories under the 7th and 8th Editions of the UNIX system. [DMR] †

The UNIX system was originally designed as a general-purpose, multi-user, interactive operating system for minicomputers. This design objective and hardware state of the art during the initial development resulted in data communications capabilites principally intended to support slow to medium speed asynchronous terminals. This environment did not require rigorous emphasis on performance and modularity as other parts of the system.

Since then, support for a broad range of devices, speeds, modes and protocols has been incorporated into the system. Development of this support has been impeded by overhead inherent in the original implementation, in which characters are processed one at a time.

The current generation of networking protocols, such as OSI, SNA, TCP/IP, X.25 and XNS, is characterized by diverse functionality, layered organizations and various feature options. Developers of these protocol suites have encountered problems arising from lack of mechanisms, structure, and software interface standards for the introduction of new components or modification of existing ones.

---

* UNIX is a Trademark of AT&T in the U.S.A. and other countries.

† D.M. Ritchie, "The UNIX System: A Stream Input-Output System", AT&T Bell Laboratories Technical Journal, p 1897, Vol 63 no 8 part 2

Attempts to compensate for performance and structural limitations have led to differing, *ad hoc* implementations. Different implementations of functionally equivalent protocol software will not mesh with layers above or below. Protocols and device drivers are often intertwined with the hardware configuration. This has limited portability, adaptability and reusebility of software parts, resulting in increased cost and confusion for system developer and customer. Various patchwork solutions have been tried with limited success.

In consideration of these problems, and of the intent to provide broad support for networking and internetworking in UNIX System V, the need to enhance character I/O was recognised.

The result is *STREAMS*. STREAMS will define interface standards for character input/output within the kernel, and between the kernel and the rest of UNIX System V. It implements an associated flexible, simple, open-ended mechanism, constructed from a set of system calls, kernel resources and library routines. The standard interface and mechanism will enable modular, portable development and easy integration of high performance network services and their components. STREAMS provides a development framework, rather than imposing any specific architecture. It coexists with the existing character i/o facilities and provides a user interface consistent with these facilities.

The power of STREAMS resides principally in its modularity. STREAMS design reflects the "layers" and "options" characteristics of contemporary networking architectures. The basic components in a STREAMS implementation are referred to as *modules*. Each module represents a set of processing functions. >From user level, modules can be dynamically selected and interconnected in the kernel to form processing sequences, with no kernel programming, assembly, or link editing required. As discussed later, this allows

- User level programs to be independent of underlying protocols and physical communication media.
- Network architectures and higher level protocols to be independent of lower level protocols, drivers and physical media.
- Protocol portability - As shown in figure 1, the same protocol modules can be used on different computers. The X.25 packet layer module interfaces with the higher Connection Oriented Network Service (CONS) and the lower Link Access Procedure (LAPB) driver.

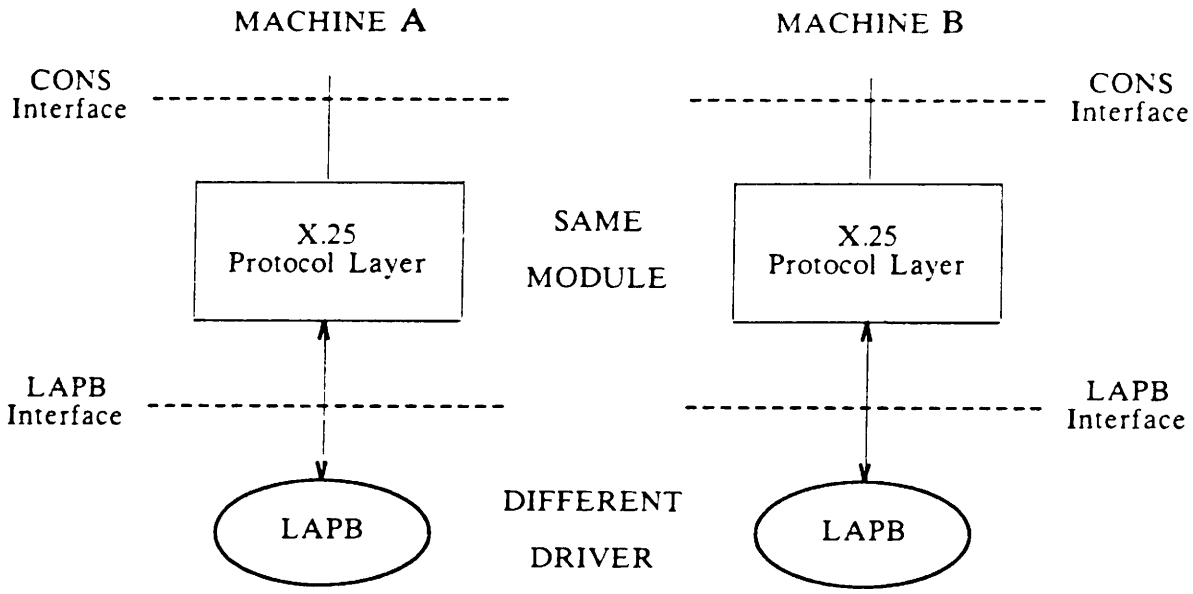


fig.1: Protocol Portability

- Protocol substitution - As shown in figure 2, alternative protocols (and device drivers) can be interchanged on the same computer.

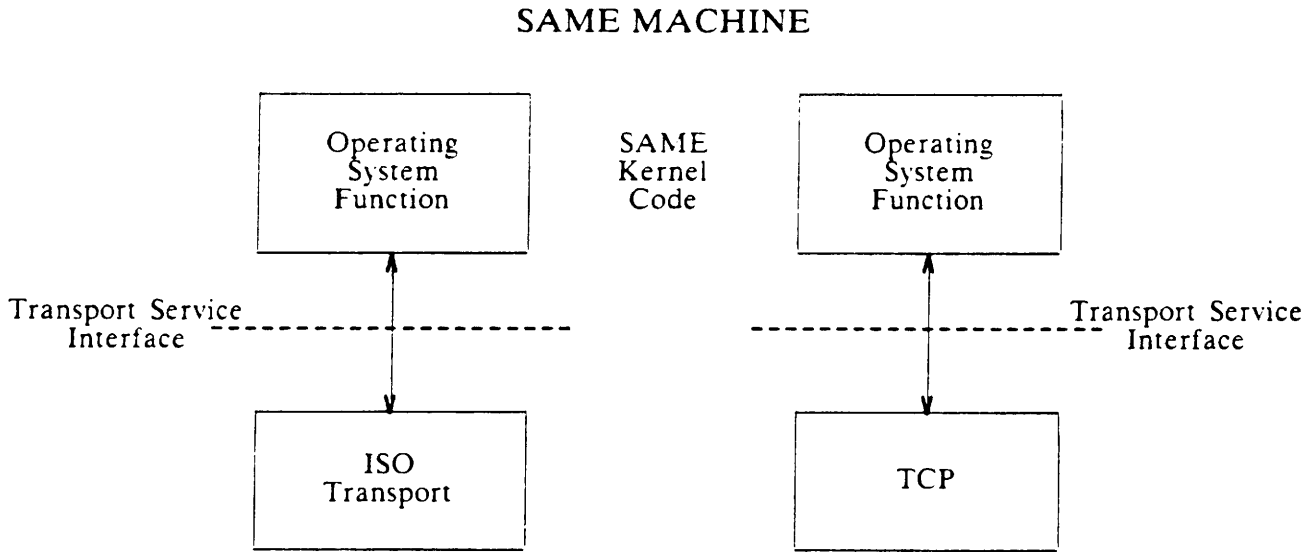


fig.2: Protocol Substitution

- Protocol migration - As shown in figure 3 - protocol functions can be transferred between kernel software and front-end hardware. The standard interface allows higher level modules to be transparent to the number of modules below. The ability to easily shift functions between software and firmware allows cost effective implementation of equivalent systems over a wide range of configurations, as well as rapid assimilation of technological advances.

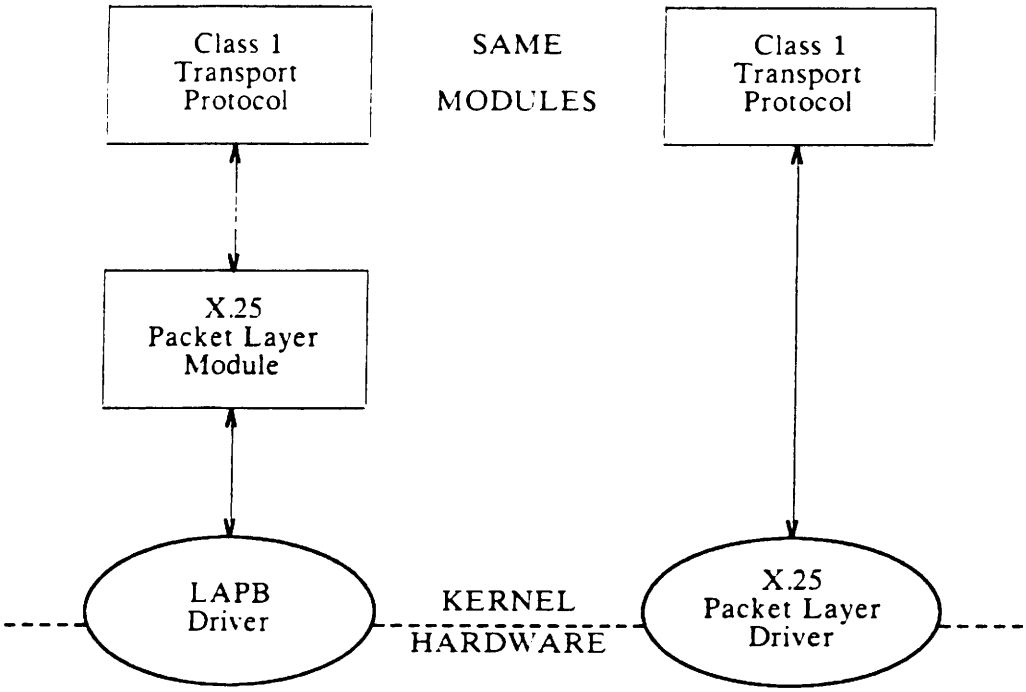


fig.3: Protocol Migration

- Higher level services to be created by selecting and connecting lower level services and protocols.

In addition to modularity and flexibility, STREAMS offers library routines and facilities that expedite design and development. The facilities include:

- Buffer management - STREAMS maintains its own independent buffer pool that can be shared by all STREAMS modules.
- Automatic flow control to conserve STREAMS buffer resources.
- Scheduling - STREAMS provides its own module/driver scheduling mechanism.
- Error and trace loggers exist for debugging and administrative functions.

2. *Internal Architecture*

stream is a full-duplex connection between a user process and an open device (or pseudo-device). Existing entirely within the kernel, it provides a general character I/O interface, plus the ability to interpose some intermediate processing modules between the user-process and the device-driver end.

The beginning of the stream, where it starts near the system call interface, is called the "stream head", and is *upstream* of the rest. The device-driver is at the *downstream* end.

System calls are converted by the stream head into *messages*, that are passed downstream through each processing module sequentially until they are consumed, and messages can be generated by processing modules and passed upstream, usually to the stream head.

2.1 queues

Each processing stage in a stream is represented by a pair of *queues*. The queue structures are allocated in even/odd pairs so that it is simple to locate any specific queue's backstream partner.

2.2 scheduling

A queue is scheduled for servicing when it has messages waiting, and when it is not blocked for flow control. All the queues ready for servicing are linked together on the run list. When the system is ready to run them, it calls the *service* procedure.

2.3 data structures

2.3.1 streamtab

```
cdevsw
_______
| ... |
| info|--|
|_____|  |
         |
         V
stream initialization table
___________________________
| upper read queue        |
| upper writequeue        |
| lower read queue        |
| lower writequeue        |
|_________________________|
```

fig.4: Streamtab

The *streamtab* table contains the linkage between the *cdevsw* for a particular device, and the queue initializations for a module's type of read and write queues.

2.3.2 queue initialization and module info

```
queue initialization structure
_____________________________
|  put procedure            |
| service procedure         |
| open procedure            |
| close procedure           |
| module statistics         |
| module information        |---|
|___________________________|   |
                                |
                                |
                                V
        module information structure
        ____________________________
        |    module id number      |
        |    module name           |
        |    min packet size       |
        |    max packet size       |
        |    hi_water mark         |
        |    lo_water mark         |
```

fig.5: Queue initialization

The *qinit* structure is a static part of the module. It contains pointers to the module-specific put, service, open and close routines.

2.3.3 queue

```
data queue
_____________________________________
| procedures and limits             |
| first data block in Q             |
| last data block in Q              |
| Q of next stream                  |
| next Q for scheduling             |
| private data pointer              |
| number of blocks on Q             |
| queue state                       |
| min packet size accepted          |
| max packet size accepted          |
| queue high water mark             |
| queue low water mark              |
|___________________________________|
```

fig.6: QUEUE information structure

A pair of *queue* structures is allocated for each stream head, driver and protocol module.

2.4 *Modules and Drivers*

A driver is a stream endpoint, it has an inode in the file system and can be open()-ed; there are media drivers and multiplexing drivers.

A module has no inode. It is referenced by a string "name" wholly kept within kernel. It can only be PUSHed or POPped on a STREAM.

2.5 *Opening a STREAM*

A stream is set up when a stream device is opened. A new info field has been added to the *cdevsw* table. If it is null then the device is an ordinary character device. But if it is non-null, the info field points to the *streamtab* table for the device. On first open, the system initializes two pairs of *queue* structures with linkages to each other and their queue initialization structures. Then the stream open routine calls the drivers open routine referenced in the qinit structure.

If the driver needs to maintain state information across calls, it must allocate its own storage area. There is a pointer in the queue structure for the driver's use in locating its driver-specific data.

Such a simple stream could be useful, for example: "raw" tty driver.

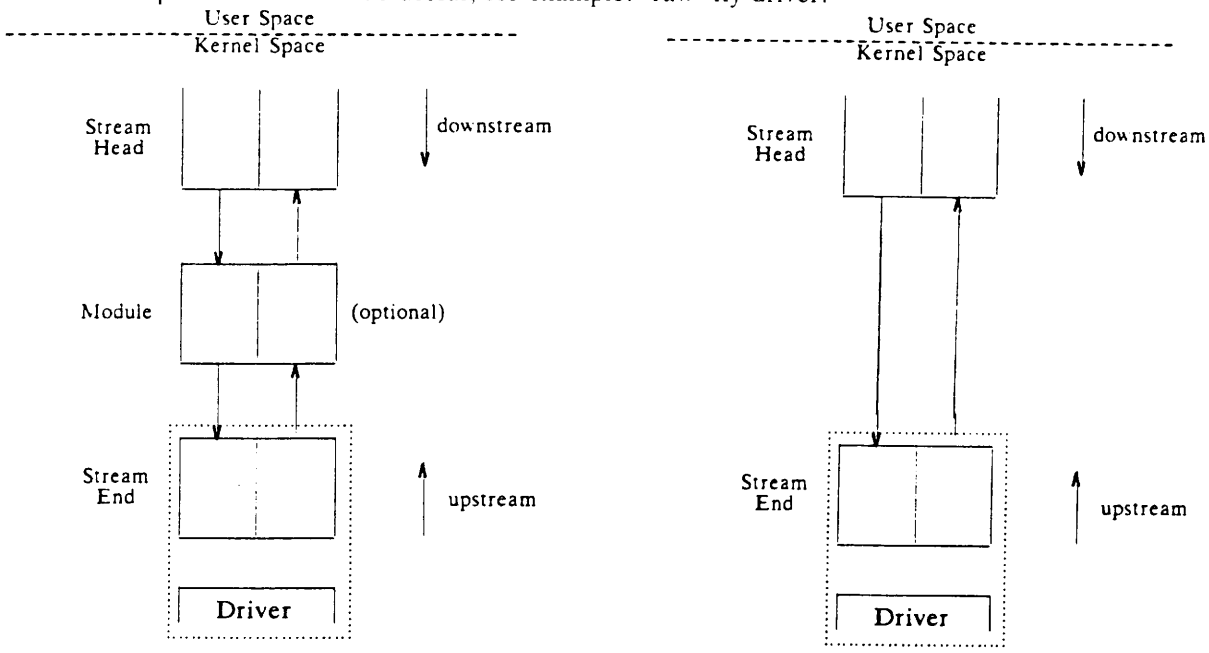


fig.7: Basic stream

2.5.1 *adding a protocol module*

Other protocol modules can be added at the top of the the stream by an ioctl call. Whenever a new module is pushed in, its module-specific open-routine is called to allow for initialization.

2.5.2 *removing a protocol module*

Similarly, protocol modules can be removed by an ioctl call. The system calls the modules close-routine. Then the modules queues are removed from the stream.

2.5.3 *Use of queue functions*

When a process sends data or control information to a stream, the streamhead converts the write, send *or* ioctl system call into a message. Each processing element passes the message along the stream by calling the *put()* procedure of the next processing element. The system provides a generic procedure *putq()* to enqueue messages. But a module developer may design a special put procedure to do some special processing on arriving messages.

Usually the put procedure will enqueue the message (using *putq()*) and this enables the queue for servicing. The system, some time later, unrelated to the process that may have caused the original message, calls the modules *service* procedure. It is the function of the service procedure to dequeue messages (using the system routine *getq()*).

It is possible to do without a service procedure if the processing can be done entirely by the put procedure. In that case the module's put procedure forwards processed messages by the system provided *putnext*.

The system routines implement flow control by checking the amount of data in a queue against the queue's limit. When this "high-water mark" is reached, queues that would attempt to pass a message to this one are disabled from servicing. When the amount of data then passes below a queue's low-water mark (having been serviced), the queue behind it is re-enabled.

3. *Messages*

3.1 *data structures that define a message*

```
Message block descriptor
_________
|  prev |-- message block *
|  next |-- message block *
|  rptr |- char * 1st byte to read ------|
|  wptr |- char * 1st byte to write ----||
| datap |-----|                         ||
|  cont |--|  |                         ||
|_______|  |  |                         ||
           |     Data block             ||
           |    _________               ||
           |    | base  |------ _________________
           |    | limit |--     |       ||        |
           |    | type  |  |    | data    buffer|
           |    | refs  |  |    |_________________|
           |    |_______|  |---
           |
           V
next message block of same message
```

fig.8: Message Structures

3.2 queueing priorities

Most messages are normal priority; they are placed at the end of a queue. A few messages are high-priority; they are placed before the normal priority messages in any queue, and are allowed to bypass the normal flow-control mechanism.

3.3 message types

* Data and protocol messages (regular priority)

DATA regular data
PROTO protocol control

* Control messages (regular priority)

BREAK line break
SIG generate process signal
DELAY real-time xmit delay (1 param)
CTL device-specific control message
IOCTL ioctl; set/get params
SETOPTS set various stream head options

* Control messages (high priority; go to head of queue)

IOCACK acknowledge ioctl
IOCNAK negative ioctl acknowledge
PCPROTO priority proto message
PCSIG generate process signal
FLUSH flush your queues
STOP stop transmission immediately
START restart transmission after stop
HANGUP line disconnect
ERROR fatal error used to set u.u_error

fig.9: Message Types

3.4 common routines for manipulating messages and data

allocate a message block
trim bytes in a message
get pointer to the queue behind a given queue
recover from failure of allocb()
test for room in a queue
make a copy of a data block (with data)
make a copy of a message (with data)
test whether message is a DATA message
duplicate a message block
duplicate a data block
flush a queue
free entire message
get next message from queue
concatenate two messages into one
remove message block from front of message
get number of data bytes in a message
prevent a queue from being enabled
concatenate data bytes in a protocol message
return a message to the front of the queue
forward message along to following queue
put a message onto a queue
put a control message onto a queue
put a control message with a 1-byte parameter
enable a queue (schedule for servicing)
send a message back up the other side of a stream
get the number of messages on a queue
remove a message block from a message
test if a buffer is available
returns address of partner queue structure

fig.10: Common Utility Routines

4. *System call interfaces*

4.1 open()

This has been discussed above in the section "Opening a stream".

4.2 close()

This has been discussed above in the section "Opening a stream".

4.3 write()

The stream head turns a write system call into a DATA message.

4.4 read()

The stream head turns a read system call into a fetch of the first message in the stream head read queue.

4.5 ioctl()

The stream head turns an ioctl system call into an IOCTL message. The following type are special for streams modules:

push named module onto top of stream
remove module just below stream head
retreive name of uppermost module
flush queues, input, output or both
register to be sent SIGSEL signal when an event occurs
returns bitmask of SIGSEL events enabled
tell whether named module is anywhere in the stream
look at first readable message without removing it from queue
set read mode: byte-stream, msg-discard, or msg-nondiscard
return current read mode setting
return size of next message to be read
send control message about another stream
send ioctl message downstream
connect a stream underneath a multiplexing driver
disconnect a stream from a multiplexor

fig.11: list of ioctl() options

4.6 send()

The purpose of send() is to allow high-level protocol information to be kept and passed with the data that it is associated to. The stream head turns a send() system call into a DATA or protocol message, depending on the presence of normal data, control information, and high-priority flags.

4.7 recv()

The purpose of recv() is to allow high-level protocol information to be kept and passed with the data that it is associated to. The stream head turns the recv() system call into a retrieve of the first message on the read queue of the stream head. Because of the expanded arguments, it is possible to receive protocol control information in association with data by use of recv().

4.8 select()

The select() system call provides users with a mechanism for multiplexing I/O over a set of file descriptors that reference open streams. *Select* identifies those streams for which a user can receive messages, send messages, or receive expedited messages. A user can receive messages through *read(2)* and *recv(2)*.

5. Some Features Built on STREAMS

Remote File Sharing (RFS)

The Remote File Sharing System feature, planned for a future release of System V, uses STREAMS to access its network transport service.

Transport Layer Interface (TLI)

A set of primitives has been defined to provide a transport service interface for user processes. This transport service interface enables two processes to transfer data between them over a communication channel. The user interface is defined as a set of library routines for the UNIX system. It is designed to be independent of any specific transport protocol, and any specific UNIX system implementation. The interface is called the Transport Layer Interface. The implementation provided for a forthcoming release of System V is implemented using STREAMS to access the services of an underlying Transport Provider.

UUCP over TLI

A new version of UUCP for system V has been enhanced to use TLI as a universal networking interface. Thus, UUCP can be run over any network for which a TLI transport provider exists, without having to alter or add special code to UUCP.

Higher level protocols

Implementations of TCP/IP, SNA and OSI Transport Protocol are being planned using STREAMS.

6. *CONCLUSION*

The STREAMS architecture provides

- structured intermodule interfaces
- efficient message passing
- automatic flow control
- scheduling of service procedures
- multiplexing
- new system calls send(), recv() *and* select.

Use of the STREAMS architecture will make networking in System V modular, allowing

- media and protocol independent user services
- media independent protocol servers, and
- simplified device drivers

through standard interfaces between kernel drivers, kernel-resident protocol modules, and user processes.

Bob Duncanson
UNIX Europe Limited,
27A Carlton Drive,
London SW15 2BS, UK

Technical Support, UEL
phone: +44 1 785-6972
fax: +44 1 785-6916
telex: 914054 UNIXTM G

PORTABLE UNIX* MEMORY MANAGEMENT SYSTEM

Hans van Someren (mcvax!ace!hvs)
Martijn de Lange (mcvax!ace!martijn)

ACE Associated Computer Experts bv
Nieuwezijds Voorburgwal 314
1012 RV Amsterdam
the Netherlands

ABSTRACT

The ACE tailored UNIX System V kernel for MC680X0 based systems accepts an extended version of the System V COFF format as standard load format. New are the support for more segments, flexible attributes, shared segments, and shared libraries. Fundamental has been to get the semantics of the segment attributes as explicit as possible, and to strive for a memory management implementation of the highest level of portability.

The advantages are numerous: increased functionality, reduced memory usage, reduced disk usage, and an improved system performance and maintainability. The extended COFF is implemented for all MMUs supported by the ACE-UNIX kernels for single and multiprocessor Motorola MC680X0 based systems.

The paper starts with an overview of UNIX object formats and memory management techniques.

1. Introduction

The ACE tailored UNIX System V kernel for MC680X0 based systems accepts an extended version of the System V COFF format [AT&T83] as standard load format [VanSomeren85]. The extension consists of a new set of section types, among which the **indirect section type** stands out, and a set of binary attributes, such as **shared** and **writable**. These are described in detail in section 3.

UNIX is a trademark of AT&T Bell Laboratories
PDP11, ULTRIX, and VAX are trademarks of Digital Equipment Corporation

September 3, 1985

Section 1 puts the ACE-COFF UNIX kernel into a **historic perspective** by highlighting the evolution of its memory management support, showing what led to COFF, and indicating which developments will be next.

Section 2 gives an overview of **object formats** (a.out and COFF) and **memory management techniques** (swapping and paging) in UNIX systems. It can be skipped by readers familiar with the topic.

Section 3 introduces **ACE-COFF** with particular emphasis on its extensions. After a treatment of the use of **shared libraries**, section 4 deals with the **implementation** of ACE-COFF in the kernel. As an example the execution of /bin/ls is highlighted, using a shared C-library.

Compatibility with System V, and future developments conclude the paper.

1.1. History of memory management implementation in the ACE kernel

The ACE-UNIX kernel is functionally a System V kernel. Its implementation has gone through an evolution parallel to the Bell UNIX efforts from Version 7 [Thompson78], through System III to System V [AT&T83]. Its evolution has been guided by the following rules:

- reconfiguration and portability to other hardware: different MMUs and peripherals;
- easy capacity tuning: tuning the code and the amount of data for a specified capacity;
- easy adaptability: new functionality, home-grown as well as from the outside should be easy to incorporate;
- efficient maintenance: testing, profiling, debugging, and updating the kernel should be easy;
- concentration of functionality: never two implementations of the same functionality.

Performance improvements have also been guided by these rules. In retrospect, most improvements have been small, localised changes, to central and heavily used code and were implemented in a few days.

In 1980 ACE started adapting the UNIX kernel to different hardware architectures. Larger applications (>64k, cf. PDP11), different "strange" MMUs, 32-bit address space, all forced revision of the restricted assumptions as they occur in the original V7 code.
As a first phase the code was:

1. cleaned in respect to the use of "standard" C,
2. made independent of the integer size (16 vs. 32 bits),
3. rigorously split in architecture dependent and architecture independent code.

This had its impact on a large portion of the code. For memory management it induced a reconsideration and redesign of the physical allocation of memory, the process-notion, swap-handling and the accessibility of memory spaces from the kernel.
The following layers of abstraction were recognised and implemented:

1. The process' address space, range and attributes. This led to a per process administration organised per (user-) segment with its attributes. To this layer belongs a set of functions that manipulate the segments, validate addresses and address ranges and manipulate mapped data.

2. The address and memory space as seen from the kernel. With this layer comes a set of functions that allocate and manipulate kernel data, where user-segments after de-mapping are considered to be kernel data. The ACE-UNIX kernel supports various memory management schemes.

3. The machine dependent memory handling. This layer deals with the actual hardware characteristics, e.g. the MMU. There are in fact three hardware dependent functions that must be written when ACE-UNIX is adapted to a new hardware architecture. This layer also deals with the (bus-dependent) organisation of the "real" physical memory, gaps, and other system peculiarities, through table driven generalised functionality.

This portable generalised kernel was used to incorporate the System III characteristics. Due to its division in system dependent and system independent parts, the integration of functionality only had its influence on the machine independent parts.

System V Release 1 as it was distributed implied more serious adaptations to the ACE-UNIX kernel; however since it still preserved the old a.out scheme functionally, no architecture dependent modifications were required. Only now that the additional functionality of multi segmentation has been added to the ACE kernel has it become necessary to provide some extra interface facilities to handle the functionality System V does not offer.

1.2. COFF and future developments

In the search for an object file format suitable for the definition of multiple segments in a UNIX environment, after some consideration, COFF was accepted. COFF is general enough to allow extension and experimentation. Furthermore software is available with System V, such as the link editor, debugger, archiver, and utilities such as size(1), nm(1), etc., although System V makes limited use of it.

However it was felt that the multiple segment aspect of COFF in a UNIX environment is not mature. The segment description is not powerful enough: how to describe that a segment can be expanded (such as bss and the stack) either upward or downward, or how to describe that a segment is read-only, or that a segment has to be shared, also when it is writable? To this end the flags definition of a COFF section header was

upward compatibly extended with a set of binary attributes describing these characteristics.

Since the attributes of segments are made explicit, segments can be treated orthogonally, enhancing the structure of the kernel. This improvement will pay off in the future when new functionality is added.

The source level implementation of virtual memory in System V has not been adopted in ACE-UNIX since it would reduce the functionality and the power of the ACE-COFF kernel. However the paging algorithms in System V served as a start for the design of paging in the ACE-COFF kernel since it is one of the cleanest implementations today.

2. Overview of object formats and memory management techniques in UNIX systems

After a brief description of the two major directions in current UNIX systems some terminology is introduced. This is then followed by the main discussion on object formats and memory management techniques employed by most current systems.
ACE-COFF and its implementation will be discussed in the next sections.

2.1. Major directions in UNIX systems

Among current UNIX systems two major directions can be recognised:

- The official and marketed AT&T Bell kernels, which have evolved from Versions 6 and 7, side-stepping to the Programmers Work Bench PWB/UNIX, to System III and System V.
  Internally within Bell Edition 8 is widespread. Through AT&T's aggressive marketing policy, System V now has evolved into a standard: the "Purple Book" [AT&T85] has been published and the European companies Bull, ICL, Nixdorf, Olivetti, Philips, and Siemens have publically announced they will support System V as part of their X/OPEN efforts [X/OPEN85].
  Until the end of 1984, the UNIX kernels available from Bell lacked virtual memory support. They used swapping as their main means of sharing the physical memory resource. The new virtual memory version, described for the VAX in [McCormick84] and here abbreviated to SV2.0p, looks promising.
  Many of the ports to micro systems are in this direction but still do not support paging.
  Most UNIX implementations claim to be compatible with System V. But especially its semaphore, ipc, and shared memory support are not universally accepted. Not without reason in our opinion.

- The virtual memory UNIX kernel that was developed by the University of California at Berkeley at the end of 1979 from the VAX port of UNIX Version 7, is the ancestor of the second direction. The development of this Berkeley Systems Distribution is discussed in [BabaogluJoy81]; [VanSomeren84] describes it in detail. Since 1979 it has gone through various incarnations, resulting in the current version Bsd4.3*, but the basic memory management functionality has

---

September 3, 1985

remained the same [McKusick85]. It was written closely around the VAX architecture. Later it was ported to a SUN (with MC68010) using comparable memory management hardware.
The kernel supports paging of user space; to make the paging implementation cost-effective quite some effort went into tuning [BabaogluJoy81]. A minor deficiency of VAX memory management hardware, the absence of a page referenced bit, proved to be costly in a UNIX environment.
With respect to the system call level functionality, Bsd4.x has gone its own way, diverging from Bell with System III and System V. Several UNIX implementations for larger machines such as Digital's VAX-UNIX implementation, Hewlett Packard's implementation for the HP9000 and that of Gould for the PN9000 stem from Bsd4.x.

2.2. Terminology

A **program** is built up from **sections**. Sections are the prototypes for **segments** and are prepared by a link editor. When a program is **loaded** by the kernel its sections are used to create or attach to segments. Processes may request the loading of a particular program by means of the exec(2) system call of UNIX.

A **process** is a program in execution; its image consists of **segments**. A segment implements a contiguous logical address space starting at a **logical (base) address** and of a certain **size**; furthermore a segment has **attributes** such as sharability, writability, and extensibility. Segments can be **private** to a process or be **shared** between processes.

Note that segments are software notions and throughout this paper they do not indicate so called hardware segments. They exist as the live incarnation of a program's section, even if the underlying MMU does not support segmentation at all.

2.3. Object formats

The object formats currently in use in most UNIX systems are derived from the UNIX/V6 PDP11 a.out format and the System V COFF format.

2.3.1. A.out

The a.out format stems from UNIX systems running on the PDP11. Figure 1 shows that it is composed of a small header describing the format and the sections, the .text section, the .data section, relocation information and a symbol table. Every individual UNIX kernel adapted the format to fit its own need: for 32-bit machines the relocation information was not adequate; for paging systems there was the need for starting sections on block boundaries; to allow external names longer than 8 characters the symbol table format was rearranged; furthermore the header often was not defined such that the different formats could be

---

this paper uses "Bsd4.x" to refer to these versions to emphasize the similarity

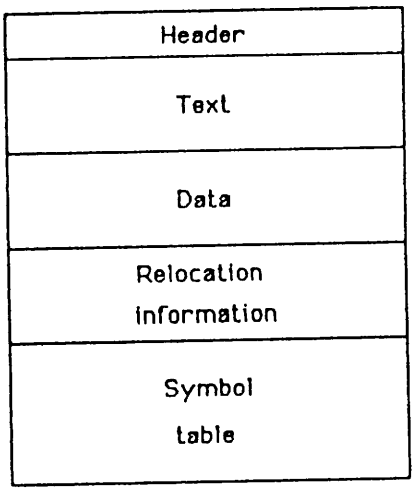


Figure 1. A.out format

recognised. But common to all a.out formats is the definition of 3 sections: a program section called ".text", a data section called ".data", and a bss section called ".bss". For these sections only the size and the initial image are defined; other attributes such as logical base address and sharing are implicit in the format. The .bss section results in a segment that is initially cleared.

2.3.2. COFF

With the introduction of System V [AT&T83], COFF was introduced into the UNIX world. Basically it recognised that the process of link editing and archiving is largely a machine and operating system independent operation. COFF, the Common Object File Format, defines the object file in such general terms that it should be easy to use in a variety of environments.

The format (see figure 2) is self-describing: it starts with a file header defining the format globally; the file header is optionally followed by system specific information (in UNIX often the a.out header); then the descriptions of the multiple sections follow (called section headers); after these section headers, for each section there is the initial image (called the raw data); then for each section relocation information; for each section line number information; the last part is a symbol table. The file header consists of a magic word, the number of sections, a time and date stamp, a byte offset to the symbol table, the number of entries in the symbol table and the number of bytes in the optional header, and finally some flags indicating whether some of this information is missing (is stripped from the file), or whether all external references are resolved, etc. The section header contains an 8 byte name for the section, a physical and logical start address, a size, byte offsets to the raw data part, the relocation information, and the line number information of the section, the number of line number

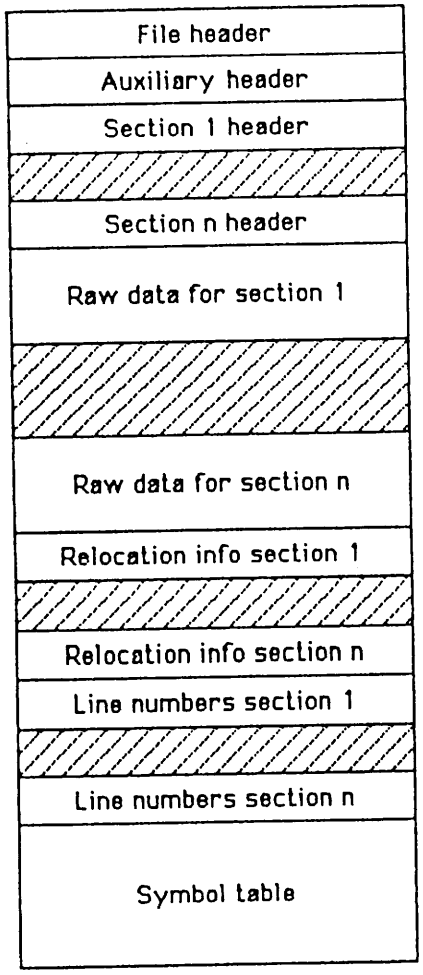


Figure 2. Common Object File Format

entries and some flags defining the type and some attributes of the section.

2.3.3. Object format peculiarities

Although the raw data parts of the sections are machine specific, it is in the definition of COFF that the headers shouldn't be. As an example it suffices to note that the byte offset of the start of a raw data part is a long, and that e.g. VAX and MC68000 disagree about the correspondence of most significant and lowest addressed byte in a long. The file header contains provisions for this: it shows in its flags whether the file was created on a machine with bytes numbered from right to left or left to right in a long, and what the word length of that machine was. But a certain source of trouble is that these flags are specified in a short! Also note that the magic number is a short, and not two bytes!

For systems employing paging and especially loading pages on demand from files it is advantageous to align the section's raw data part on a block boundary. Then transfers of pages from the file correspond to an exact number of blocks starting at a block boundary. When preparing an object file for another system, one must know the size of blocks it uses for the file system, which makes the COFF object file non-portable.

2.3.4. What UNIX does not expect to be supported by object formats

Once the UNIX link editor has made its final pass over the object file the logical addresses and sizes of sections are frozen. UNIX will not attempt to relocate on loading (during exec). Also, the object file will contain a complete description of the program; there is no linking during loading, let alone dynamically during execution.

UNIX does not understand about dynamic segment management. System V contains a scaled-down facility for it. In SV.2 the shared segments are not swapped; in SV2.0p, the paging version, they can be.

Files and segments are different concepts, contrary to systems like Multics [Organick72]. Authorisation and access in UNIX are different for segments and files. But support for treating files as segments is needed from the machine architecture, and is therefore less portable.

2.4. Memory Management techniques

Swapping and paging are used in UNIX systems to manage main memory. After an introduction to the essentials of swapping and paging some typical implementations are examined. Finally performance is discussed.

2.4.1. Swapping vs. Paging

Most UNIX systems employ **swapping** as their main technique for managing the physical memory resource. With swapping, processes are typically either completely present in main memory or are swapped out to secondary storage.

This is an essential difference with **paging** systems (or more exactly, systems to which paging has been added). In these systems usually only the most essential segments are always present in main memory; of the other segments, pages will be brought in when a process makes a reference to them. In this context a **page** is a small part of the logical address space of a process: in fact this space is composed of a set of pages, all equal in size. Logical pages are mapped onto physical pages by the paging MMU. Usually the mapping is specified to this MMU by means of **page tables**: tables of **page table entries** each of which specifies the mapping of one page. With small pages, a large logical address space, and the possibility of extending the addressable logical address space, page tables tend to be large too, and segments on their own. Page table segments are only accessible to the UNIX kernel.

Important in paging systems is that there is a supply of empty physical pages. Since user processes try to grab these pages and do not release them voluntarily, there must be a mechanism to bring them to the free

pool again. In the two virtual systems described here this function is performed by a second kernel process, the **page daemon.** It regularly scans all pages in use, determines referencing frequency, and detaches the least referenced ones from the process. To obtain "referenced" information for a page, some minimal hardware support, e.g. setting of a bit in a page table entry when the corresponding page is referenced, certainly is cost effective [BabaogluJoy81].

From theory and practice it is known that a process requires a minimal set of its pages to be present to run comfortably: its **working set.** When the sum of the working sets of the processes exceeds the amount of physical memory available, a swapper still has to exercise control to swap processes out. So paging is not an alternative for swapping, it is a refinement to postpone swapping as long as possible.

2.4.2. Pure swap based systems

Many UNIX systems are pure swap based. To this set belong the original UNIX systems Version 6, Version 7, and System III, and until this year System V. There are however degrees in how much memory is swapped each time. If we assume that a process should be swapped out because another process must be swapped in but cannot be allocated memory, it can be seen that if the chance that memory allocation is successful can be increased, swapping will be reduced. This chance can be increased through allocating in smaller portions, making better use of fragmented memory.

The following implementations can be discerned:

- All segments of a process are allocated in one chunk of physical memory, and are swapped in one operation. The chance of allocation failure is very high. Also problems arise when a segment is extended and the boundary is somewhere in the middle of the chunk: the complete chunk has to be extended.

- One memory chunk is allocated for each segment, and segments are the unit of swapping. On a segment extension only the corresponding chunk has to be reallocated.

- More physical memory chunks may be allocated for each segment. Each segment then may require more swap I/O actions. This allocation method optimises physical memory usage, at the cost of requiring more but shorter swap actions.

2.4.3. Virtual swap based systems

A variant of the swap based systems are the systems that allow chunks of segments to be swapped in on demand. Usually the size of the chunks is much larger than any paging system uses for pages. Again systems can be categorised as done for pure swap based systems above. But <u>when</u> swapping is done is another criterion:

- The simplest virtual swapping systems always swap out all segments, and swap in all segments on demand.

- More advanced systems also employ "load on demand", "zero on demand", and "copy on write" (see below) on segment level.

- When segments of a process are also swapped out while the process is running, it has more of the characteristics, including the overhead, of a paging system.

Especially in the case of a large number of segments, a virtual swapping system performs well.

2.4.4. Paging systems

The utmost limit in allocation and page I/O actions is reached when each page is individually allocated and swapped. The two most important improvements to this behaviour that are found in Bsd4.x are:

- Increasing the page size of the system to a multiple of the hardware supported page size. Typically the optimal page size is in the order of 2k bytes.

- Clustering pages on I/O transfers: when one of a set of (for the I/O controller) contiguous pages has to be transferred, check whether it is advantageous to transfer the neighbouring pages too, e.g. when they have been modified, to update the swap image. Clustering also has the effect of prefetching pages.

Extensions, related to paging on demand, are:

lod Allocate and load the page from the object file on the first reference, instead of loading all pages from the object file at once as result of the exec(2) system call. The obvious advantage is that the working set is allowed to grow gradually, and not forced to a maximum immediately.

zod Allocate and zero the page on the first reference: the companion of lod in the case of a page that only has to be cleared upon loading.

cow Allocate and copy the page on the first write access. The original is shared and set temporarily read-only. Copy on write is used to reduce the overhead of forking.

Paging is not for free; the effort of implementing paging is not comparable to implementing allocation of segments in more than one chunk. Additional effort should be put in:

- The page daemon, referred to above, to steal pages from processes on behalf of the free page pool. This introduces additional concurrency which has always been a weak point of the UNIX kernel. Also hardware support for obtaining referenced information is extremely useful.

- The page table management. Due to their importance page tables and the user structure play a special role in the kernel. In Bsd4.x they are swapped instead of paged. They form an additional layer of memory space that has to be managed logically as well as physically.

- Sharing management. With Bsd4.x and SV2.0p we see different solutions for mapping shared segments. Bsd4.x keeps page tables per process; the page table entries of a segment are not shared, only the physical pages are shared. SV2.0p for the VAX shares page table entries of a shared segment, at the expense of fragmenting the logical address space of processes in pieces of 64k bytes. This is done by keeping blocks with page table entries associated with the segments, and using the fact that page tables are mapped in system logical address space, to map these page table blocks into the page tables of the sharing processes.

- The swapping algorithm. In Bsd4.x the swapper bases its actions on averages of the number of free pages, and the paging rate; these have to be monitored. It has two modes of operation: if memory becomes very tight, it switches to desperate mode in which it only swaps out processes.

In summary, paging not only interferes with memory allocation but also with many process operations such as forking, execing, and exiting; it adds an additional level of management to the implementation of the kernel; and it requires the introduction of advanced monitoring and dedicated processes.

2.4.5. Performance of memory management techniques

Extensive measurements and simulations of swapping behaviour in loaded (non-paged) UNIX systems have shown that:

> The amounts of process swap-outs initiated because allocation of a segment in physical memory fails due to lack of free contiguous memory (e.g. as a result of fragmentation) is reduced by more than 20 percent if segments are not allocated in one contiguous chunk of physical memory but in several smaller chunks of memory.

This result implies that scattered allocation of physical memory in several chunks per segment gives a considerable improvement of loaded systems behaviour. To allow this, the hardware MMU should allow logical contiguous mapping of physical scattered memory.

The often claimed disadvantage of a scattered scheme, is the supposed overhead of multiple IO (DMA) operations per segment-swap. That this is an example of penny-wise/pound-foolish thinking is supported by the following considerations:

1. 20 percent reduction of process(!) swaps certainly compensates for a few more (interrupt-) IO operations;

2. the majority of disk controllers do not allow for large (e.g. more than 32 or 64k bytes) DMA ranges anyway; so there will often be multiple IO actions for a swap, even if a segment is allocated contiguously.

For a large collection of measurement results see the report [Van-Konijnenburg85].

3. ACE-COFF

3.1. Object format

Very regrettably System V only makes little use of the potentials of COFF: object files in System V are just a COFF representation of the old a.out format. Only 3 sections (.text, .data, and .bss) are accepted by the System V kernels and their arrangement in memory is the same as with the a.out format. Only for debugging the power gained is exploited (sdb). To discriminate between the many segments in ACE-COFF the section header of System V COFF has been adapted: the flags field is replaced by an attribute-type combination, and the header has been extended with a version stamp; see figure 3. The following binary attributes are defined:

Shared: desired and visible sharing of segment
Read: read allowed
Write: write allowed
Exec: instruction fetches allowed
Upext: upward extension allowed
Downext: downward extension allowed
Group: combine segments after loading
Bonded: on a re-link-edit, do not reallocate this segment
Stack: this segment may be grown by the kernel (up or down!)

And the following types of sections are recognised:

Regular: (already in SV-COFF) initial image in raw data part
Clear: (generalisation of bss type) initial image is cleared
Indir: use section from other COFF object file

Next the **indirect** section type, and the **shared** and **group** section attributes will be discussed in more detail.

3.1.1. Indirect section type

Due to the introduction of the **indirect section** it is no longer necessary for all raw data parts to be present in one COFF object file. An indirect section refers to a section in another COFF object file. The (absolute) path name of this file forms the raw data part of the section; the section size indicates the length of the name. The target section will be the section in the target file with the same section name as the invoking (indirect) section. The target section may be an indirect section itself thus forming a chain of sections starting in the program file specified by the user and ending at a non-indirect section,

section name (8 bytes)
physical address (not used)
virtual address (base)
section size in bytes
file ptr to raw data
file ptr to reloc entries
file ptr to lineno entries
no of reloc entries
no of lineno entries
section attributes
section type
version stamp

Figure 3. Section header in ACE-COFF

called the **effective** section. Figure 4 shows such a chain.

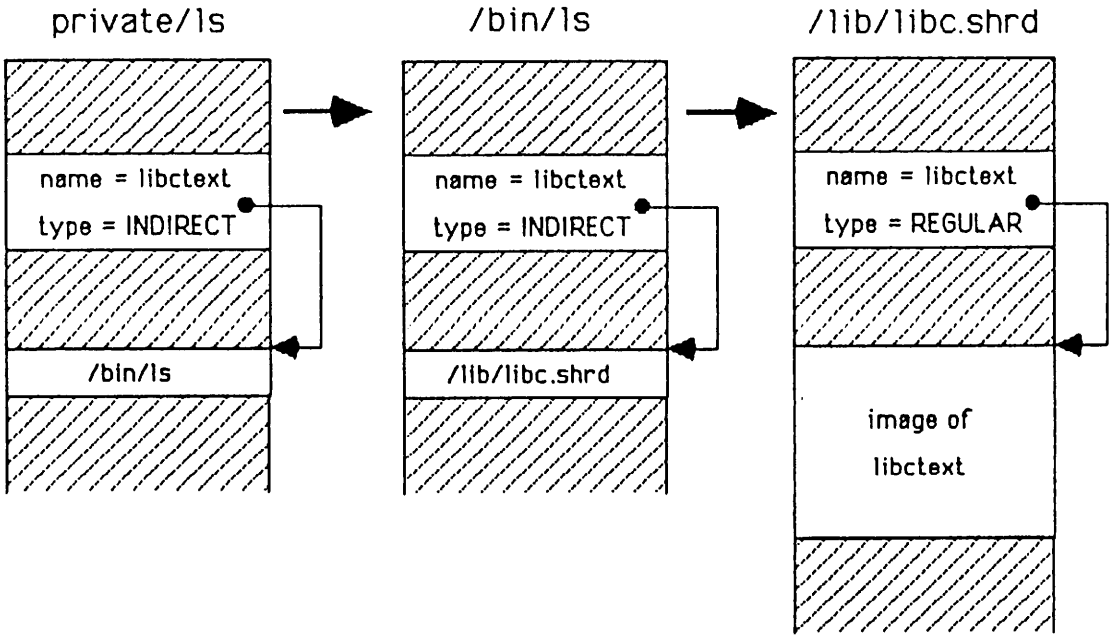


Figure 4. Chain of indirect sections

September 3, 1985

Each such effective section will usually be of the **regular** or **clear** type. In the former case, the effective section's raw data part will be used as initial image for the segment; otherwise it will be cleared initially.

Generation of the **effective attributes** that will be used to create or attach to a segment is discussed below. These effective attributes also determine whether the effective section will be part of a group or will be shared.

Of all files in the chain it is required that they are executable and regular files, that they not be open for writing, and that they are in COFF format. A new magic word is defined to disallow execution of a COFF object file present only as target for indirect sections. To prevent running through a cycle in the chain of indirect sections, a maximum on the number of indirect sections in the chain has been defined (currently 8). For all sections in the chain, the version stamp should be the same.

One may use indirect sections to impose one's own attributes on the attributes of the sections of an existing COFF object file, e.g. to make a section writable to set break-points. But more important, when indirect sections are made to refer to the sections of a large library (in which all routines are linked together), the raw data of these library sections do not have to be physically included in the invoking COFF object file, saving disk space.

3.1.2. Shared section attribute

Regardless of other section attributes the **shared** section attribute indicates desired and visible sharing of the resulting segment. The conventional sharing semantics of text segments is visible when the attribute's value is **true**: writing to the COFF object file is not allowed; the kernel considers the COFF object file opened, etc. When its value is **false**, any sharing or association with a file is not detectable. The ACE-UNIX kernel then also shares the segment, if it is read-only, but detaches on a relevant open for write or a dismount.

If a segment is shared, it cannot be extended. For grouping shared segments, see grouping below.

3.1.3. Section attributes effective during loading

Stepping through the chain of indirect sections leading to the effective section the effective attributes are derived to meet the following requirements:

1. the segment corresponding to the effective section will only be shared if all sections in the chain of indirect sections specify a **true** shared attribute value, and

2. additional indirect sections in a chain to an effective section cannot increase the capabilities derived from the effective attributes, when the corresponding segment will be shared.

Expressed as an algorithm, the **effective binary attribute values** for a segment are derived as follows. For a section S1 with the chain

```
S1 -> ... Si .. -> Sn
```

E, the set of effective values, is computed as:

```
for attr in {binary attributes}
do
        E.attr := S1.attr
od
for i in [2..n]
do
        E.Shared := E.Shared & Si.Shared
        if E.Shared
        then
                for attr in {binary attributes}
                do
                        E.attr := E.attr & Si.attr
                od
        fi
od
if E.Shared
then
        for attr in {Upext, Downext, Stack}
        do
                E.attr := false
        od
fi
```

Now, if the effective shared attribute is **true**, sharing of the segment is desired; otherwise sharing must not be externally detectable, and private instances of the segment appear to exist. The initial size of the segment is taken from the size field of the effective section; the logical base address is taken from the vaddr field of the section in the program file.

3.1.4. Group section attribute

Loading sections together in one contiguous logical address space can be achieved through the **group attribute.** This group attribute indicates that the next section belongs to the same group as the current section.

Use of groups:

- gives the user control over the grouping of segments in fewer hardware segments, especially if the number of these for a particular system is small compared to the number of supplied sections;

- provides downward compatibility to the old a.out format, where in the 0407 format text, data, and bss must be contiguous in logical address space, and in the 0410 format data and bss must be contiguous in logical address space.

Limited by the fact that the size of a segment may not be a multiple of the MMU granularity of the underlying hardware, segments of the same group may have to be mapped by the same MMU descriptor. The effective attributes for all sections in a group from the set {shared, read, write, exec} must be the same. Only the section in a group with the lowest addresses may specify downward extensibility, or the section with the highest addresses upward extensibility. Section types may differ in a group. Note that the effective value of the group attribute is computed just as the other attributes, so grouping is only done if all intermediate sections agree.

3.1.5. Example

The program "/bin/ls" will be used in three examples below: first an ordinary 0410 equivalent COFF object file will be shown; the second example shows the C-library in COFF format; in the third example /bin/ls will use indirect sections to refer to the C-library.
The examples will abbreviate the section attributes by their first letter, and the section types by the first 3 letters.

Ordinary 0410 a.out equivalent:
/bin/ls:

```
        filehdr
        aouthdr
        scnhdr          .text           Reg     SR-X-----
        scnhdr          .data           Reg     -RW---G--
        scnhdr          .bss            Cle     -RW-U----
        scnhdr          .stack          Cle     -RW--D--S
        raw data        .text
        raw data        .data
        ...
```

C-library as target for indirect sections:
/lib/libc.shrd:

```
        filehdr
        aouthdr
        scnhdr          libctext        Reg     SR-X-----
        scnhdr          libcdata        Reg     -RW---G--
        scnhdr          libcbss         Cle     -RW------
        raw data        libctext
        raw data        libcdata
        ...
```

/bin/ls using the shared C-library:

```
/bin/ls:
        filehdr
        aouthdr
        scnhdr          libctext        Ind     SR-X---B-
        scnhdr          libcdata        Ind     -RW---GB-
        scnhdr          libcbss         Ind     -RW----B-
        scnhdr          .text           Reg     SR-X-----
        scnhdr          .data           Reg     -RW---G--
        scnhdr          .bss            Cle     -RW-U----
        scnhdr          .stack          Cle     -RW--D--S
        raw data        libctext        /lib/libc.shrd
        raw data        libcdata        /lib/libc.shrd
        raw data        libcbss         /lib/libc.shrd
        raw data        .text
        raw data        .data
        ...
```

3.2. Using shared libraries

The System V linkage editor accepts and generates COFF object files. Unlike the old UNIX linkage editors this linkage editor is more conveniently controlled by means of a command file. This new linkage editor has been adapted to accept and generate the ACE-COFF format. Because of indirect section types in ACE-COFF, conventions have to be established where to find the libraries. And because of the shared section attribute in conjunction with the indirect section type, one has to agree on the placement of library segments in the logical address space. Having indirect section types saves file system space; sharing segments saves physical memory space, and speeds up the loading of programs.

3.2.1. COFF link-edit command file

With the System V linkage editor, individual sections of the output file can be composed of specified sections from input files. Output sections can be located at a logical address that can be expressed in a powerful notation; the default is to locate them next to each other starting at the lowest addresses. Files that should always be incorporated in the output file can be specified, as well as which directories should be used for the library search. With the ACE extension, attributes of output sections can be specified.

The following COFF link-edit command files generate the COFF object files of the example in the previous chapter.

The ordinary 0410 a.out equivalent is generated by:

```
SECTIONS
{
        .text   ALIGN(0x10000) (REGULAR: READ, EXEC):   {},
        GROUP   ALIGN(0x10000):
        {
                .data   (REGULAR: READ, WRITE): {},
                .bss    (CLEAR: READ, WRITE, UPEXT):    {}
        },
        .stack  0xfff000 (CLEAR: READ, WRITE, DOWNEXT, STACK):
        {
                .=.+0x1000
        }
}
/*
 * link with crt0.o
 */
/lib/crt0.o
```

The C-library in /lib/libc.shrd is generated from the ordinary object module libc.o (with all routines linked together in the sections .text, .data, and .bss) by:

```
-o /lib/libc.shrd
SECTIONS
{
        libctext 0xb00000 (REGULAR: SHARED, READ, EXEC):
        {
                libc.o  (.text)
        },
        GROUP   0xb10000:
        {
                libcdata (REGULAR: READ, WRITE):
                {
                        libc.o  (.data)
                },
                libcbss (CLEAR: READ, WRITE):
                {
                        libc.o  (.bss)
                }
        }
}
```

And /bin/ls using the shared C-library is generated by:

```
SECTIONS
{
        libctext (INDIR: READ, EXEC, SHARED, BONDED):
        {
                /lib/libc.shrd  (libctext)
        },
        GROUP:
        {
                libcdata (INDIR: READ, WRITE, BONDED):
                {
                        /lib/libc.shrd  (libcdata)
                },
                libcbss (INDIR: READ, WRITE, BONDED):
                {
                        /lib/libc.shrd  (libcbss)
                }
        },
        .text   ALIGN(0x10000) (REGULAR: READ, EXEC):   {},
        GROUP   ALIGN(0x10000):
        {
                .data   (REGULAR: READ, WRITE): {},
                .bss    (CLEAR: READ, WRITE, UPEXT):    {}
        },
        .stack  0xfff000 (CLEAR: READ, WRITE, DOWNEXT, STACK):
        {
                .=.+0x1000
        }
}
/*
 * link with crt0.o
 */
/lib/crt0.o
```

Note that the bss segment of a shared library is not upward extensible; one upward extensible segment for mallocs seems enough. The alignment clauses in the examples cause placement of the section in the logical space at the next address that is a multiple of the ALIGN parameter, and are needed for a particular MMU that does not allow two segments with different attributes within a certain logical area. Note the specification of the .stack section: the initial stack size is 4k bytes; the stack will grow downward as usual during execution as indicated by the STACK and DOWNEXT attribute.

COFF link-edit command files will only be used for special purposes. The compiler will pass a command file at a default location to the COFF link editor when no command file is explicitly passed to it, so that ordinary users will not be confronted with such files.

September 3, 1985

3.2.2. Conventions about logical location of shared segments

In UNIX systems the load image is completely prepared in advance, to make loading by the kernel into the user's memory simple; there is no load-time relocation or whatsoever.

There is thus the problem that the logical locations of a program and of a shared library may conflict. Having a large amount of logical address space allows a partioning of that space into "segments": it is thus acceptable to make a decision for a particular machine-MMU combination about a fixed position of the segments of a shared library in the logical address space of every process, using the COFF command language for customising purposes.

To limit the number of relinks when a change is made in a shared library, ACE-COFF uses vectoring.
The text segment of the shared library has been fixed at the logical location 0xb00000, and its data segments at 0xb10000.

3.2.3. Conventions about location of libraries in the file system

Since absolute path names of libraries are incorporated into object files, their location in the file system has to be decided upon. At ACE "/lib" has been chosen.

Libraries should always be reachable:

- When present on a standard but mountable file system, they cannot be accessed during boot-time.

- When a shared library file has to be replaced by a new version, there is a moment in time that it is absent. When it is done by executing two mv(1) commands, the second mv cannot use the shared library itself.

- If a disk block becomes bad and it happens to be in the shared library, it cannot be repaired by programs using the shared library.

Programs needed to boot, to maintain, or to upgrade a shared library should not use the same shared library. ACE uses a separate system with a private file system with programs not using shared libraries, or, when a system does not have a removable file system, a small set of permanently available commands.

4. ACE-UNIX kernel

The ACE-UNIX kernel accepts ACE-COFF as object file format [Van-Someren85], and supports it on a variety of MMU types. Functionally it is an upward extension of a standard System V kernel.

4.1. COFF Memory Management support

The memory management support for COFF has been designed following the rules given in the introduction.

4.1.1. Overview

Special treatment of particular and known segments in the UNIX kernel has been replaced by operations based on segment attributes. The text segment now essentially is a shared, non-writable, non-extensible segment, to be initialised from a regular COFF section. Likewise the stack is a non-shared, readable, writable, downward extensible, automatically growable segment, initialised from a COFF section of type clear. Similar characteristics can be given for the data and bss segments.

Each individual attribute of a segment is reflected in private code in the kernel to implement it. The management of sharing segments has been redesigned, as have the extension of segments, loading of segments, allocation of segments, mapping of segments, swapping of segments, and the management of which of the images is valid in case of replicated segments.

Based on a generic queue package and a flexible dynamic memory pool package (called "chip pool"), a memory management database has been designed. This database has an interface to the processes having segments, files (actually inodes) being the association for shared segments, physical memory and swap space implementing segment images, and the MMU dependent part implementing mapping of segments.

4.1.2. Views, segments, maps and chunks

For the single processor swap based kernel, an overview will be given of the memory management database. The important structures are **view**, **segment**, **map** and **chunk**:

view: **Views** implement the process private effective attributes of an individual segment. They contain the logical base and size of the segment, and the binary effective attributes (shared, etc.). Each logical address domain of a process is described by a list of views rooted in a queue in the process structure.

segment: The **segment** structures represent the logical segments as the system sees them: the size and maximal effective attributes, some hidden flags, and sharing and loading information. The loading information consists of the section's name, version stamp, type, pointer to the raw data part, and the section's size.
Sharing is by association with the internal representation of a file, an inode, that contains the head of a list of segments that are associated with that file. The section's name in a segment is used to discriminate between the segments in this list.
Furthermore each segment is the head of a list of the views using the segment.

map: **Maps** implement the logical but mappable image of a (group of) segment(s). Each map is the head of a list of segments belonging to the group using the same map.
The map heads the lists implementing the replicas of the map's contents: in the example the blocks on swap space and the chunks of physical memory. In the map structure the real size, the map's state, and the in-core count are kept. The map's state records whether the in-core or on-swap image is valid, or allocated but invalid, or not allocated. Dirty map management and swapping are completely guided by this state information.

chunk: A **chunk** describes a range in an address space, in particular the physical memory. The attributes kept are the base and size, and some hidden flags.

The queues that are used to implement the relations in the database provide a doubly linked list between the queue's elements, a mechanism to get from each element to the queue's head and from there to the structure physically containing the queue's head, and (particularly relevant for the multi processor implementation) a synchronisation element. Queues are very carefully typed to get the utmost profit from the compiler and lint. This typing is done as interface between the queue implementation and a particular type of queue; the queue implementation has been written generically for these different types.

The database can be very concisely described in the database relation notation in which "N <--->> M" indicates a one-to-many relation between N and M; see figure 5.

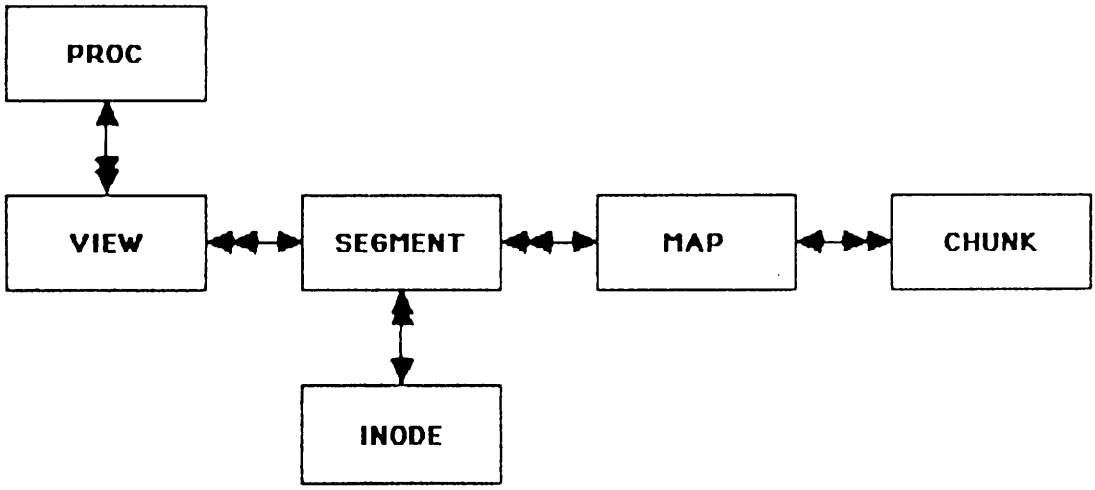


Figure 5. ACE-UNIX memory management database relations

September 3, 1985

4.1.3. Example

Figure 6 shows the database structure for the case that /bin/ls is loaded using the shared C-library. The text segment is linked in the association queue rooted at the inode of /bin/ls, and the libctext segment linked in a queue rooted at the inode of /lib/libc.shrd. Also if there are more users of .text of /bin/ls, their views are linked in a queue rooted in the text segment, and the views using libctext are linked in a queue rooted in the libctext segment.

When the segments had not been loaded, the data segment would also have been linked in /bin/ls's association queue; as would libcdata have been in the library's association queue. Segments of type clear do not need an association because of loading.

4.2. MMU Overview

With the number of segments, their sizes and relative positions depending on the user's specification, the kernel's algorithms to map the segments in memory have to be very general.

In a swap based kernel it is important to guarantee that the process can be completely allocated and mapped in memory. Generally it is required to check that the maps of a process do not overlap. The ease of these checks heavily depends on the type of MMU. Obvious is that the operations to manipulate the MMU are MMU dependent. Furthermore, whether the processor employed allows recovery from bus errors, influences MMU handling; especially for the MC68451 MMU handling can be much simpler with the MC68010 processor.

The ACE-UNIX kernel has to map ACE-COFF with all its generality with the help of many different types of MMUs. A classification of MMUs is difficult; rather by focussing on some peculiarities the mapping problems may be appreciated:

- The MMU may define one logical address space for user and kernel (e.g. VAX), or have different logical address spaces. In one space data transfers between user and kernel space are just copy loops; with two spaces special machine instructions have to be used, or mapping must be simulated.

- When a DMA IO controller has to do a transfer such as for swap or physical IO to user logical space, it is important in which address space it generates addresses. It may employ a private address space, the kernel's address space, or just the physical address space. In case of a private address space, it must set its mapping to physical space equal to the user's mapping. In case of the physical address space, it has to split its transfer in parts corresponding to physically contiguous parts mapped to from the user address space. With kernel address space, either this space should contain the complete physical space, or it should be possible to change the mapping of a part of it.

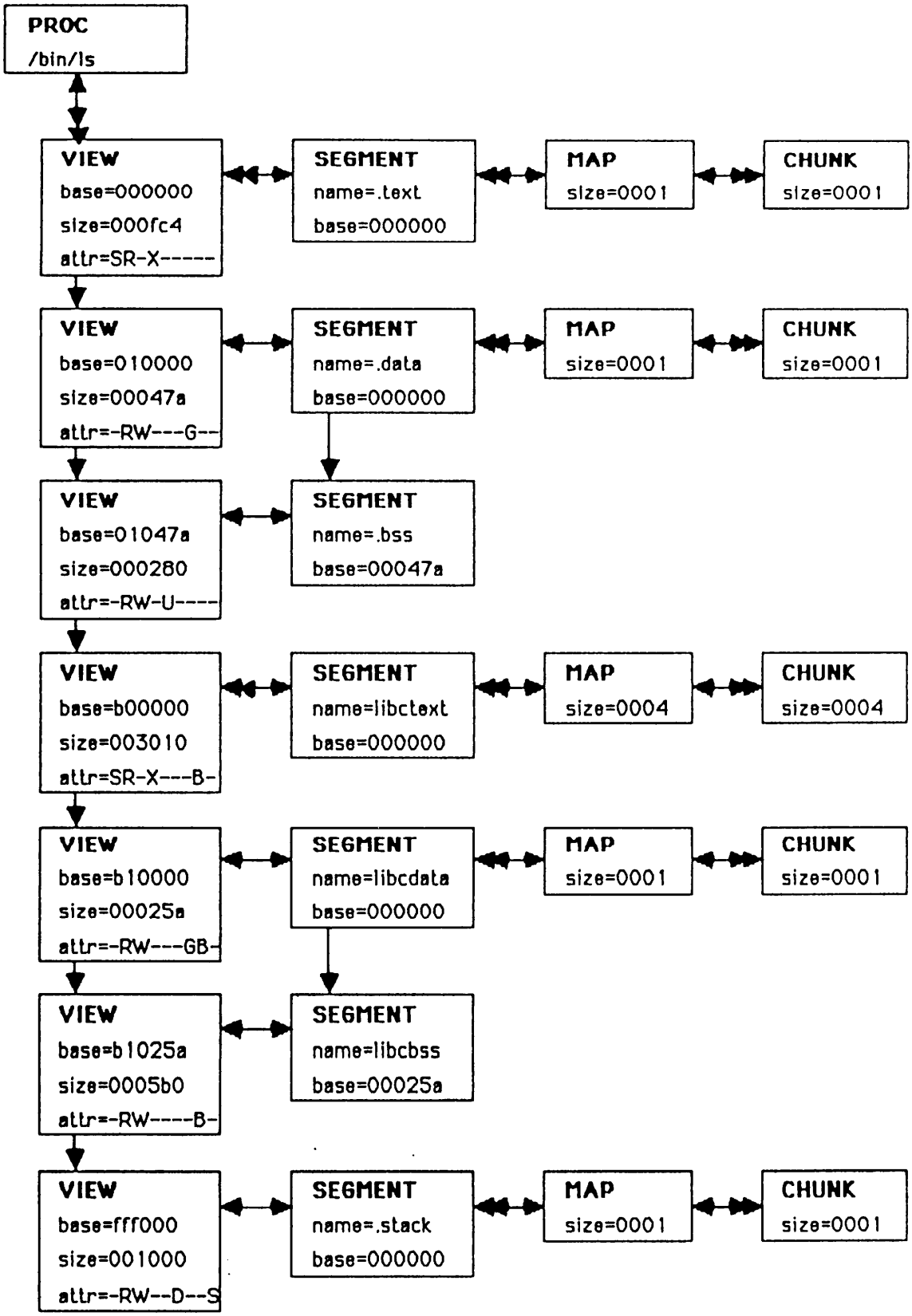


Figure 6. Example of the /bin/ls database

September 3, 1985

- The MMU may partition the logical address space in a number of **hardware segments** of fixed size. Each of these may be individually mapped on a contiguous range of physical memory. Two segments with different attributes within one hardware segment cannot be mapped. See the discussion on groups above. Also the number of partitions may be too small to accommodate the number of segments.

- The ranges of logical addresses in which the MMU partitions the logical address space may be arbitrary except for a small granularity. As before, the number of ranges may be restricted.

- There may be a relation between the logical range and physical range onto which it is mapped. The MC68451 is a well-known example of this category: for a given size of a hardware segment, the logical base and the physical base must be a multiple of that size. Especially large segments and segments not beginning on a nice logical boundary cause trouble. Frequently more MMU segment descriptors are used to map one segment.

- On every context switch the MMU may have to be fully reset, it may be able to remember a (too) small set of settings, or may fetch its new setting from memory itself.

- Referenced information or modified information may be kept or not. If not, it may have to be simulated: e.g. a writable segment will always be modified, or segment faults are used to catch accesses to the segment.

4.3. Core format and debugging

For COFF, core dumps usually contain more than the ordinary three segments. The ACE kernel's core format is COFF again. In an effort to decouple the kernel implementation and specification, the ptrace(2) system call has been respecified. As a result it is possible to continue "correctly" aborted programs.

4.3.1. Core format

Having the access routines for COFF object files at hand, dumping in COFF format is an obvious choice. All segments are dumped as regular typed sections, the logical address and size reflecting the segment's base address and size, and the attributes equal to the segment's effective attributes. The sections are named after their originals. The correspondence between the sections in the object file and the core file is now simple.

For the debugger this means a tremendous relief from core format details.

The process information saved from the processor is dumped in an additional section called ".state": it contains the register contents, status register, and an indication whether the image is restartable with these values.

The kernel's private process information, stored in the user structure, is dumped in a section called ".envunix", and is a structure of structures each corresponding to the type of a ptrace(2) packet.

4.3.2. Ptrace

Instead of allowing ptrace access to the process' user structure by specifying offsets from its base, the information in the user structure and proc structure has been collected and regrouped in logical packets: process id, signals, timing, accounting, user limits, global file information, profiling, and open file information. For open files as much information as possible is saved; the name of the files, if any, however is not part of the immediate knowledge of the UNIX kernel.

Access to the process' segments is by logical address; the maximum amount of bytes to be transferred is now 512 instead of the original 2. New are a counter for single steps, the possibility to catch every system call issued by the debuggee, and execution of a system call on behalf of the debuggee. These facilities are used by the ACE versions of adb(1) and sdb(1).

4.3.3. Continuing an aborted process

Because the core file is much like an object file, little effort is needed to use the core file as object file. By using the .state section for its register values and processor status value the process is effectively continued after the point causing the dump. Of course, it only makes sense to continue if the process did not dump because of an irrecoverable error but because of a voluntary abort, or a quit signal. Restoring the UNIX environment state of the process is not done by the kernel; usually the shell is used to open files again. In extreme cases, a separate program is used. Restoring a process completely, including its open file state is impossible: these files may be deleted after dumping core, even worse, their identifying <dev, inumber> may have been occupied by another file.

4.4. System V compatibility

4.4.1. COFF format

The ACE-COFF format very closely resembles the System V COFF format. There is a translation program for conversion between the two formats. The kernel accepts both. The System V COFF format is only accepted as far as System V2.0 kernels accept COFF, i.e. at the level of a.out. In fact the ACE-COFF implementation can handle both section header formats. In the same way the old a.out format is accepted; the kernel contains prototype section headers for a.out sections.

4.4.2. System calls

The break(2) system call to extend one's data segment, extends the segment created from a .bss section. There is a new system call to set, get, or change the attributes of particular views and segments.

4.4.3. Shared segment handling

System V shared segments are simulated with the segments of the kernel; only the functionality for matching and attaching is special for System V compatibility. ACE-UNIX uses another, more elegant, mechanism for dynamically creating and sharing segments.

4.4.4. Paging (SV2.0p)

The System V2.0p kernel running on 3B20 and VAX, described in [McCormick84], allows four "regions": a text, data, stack, and shared memory region. Regions correspond to combinations of segment and map in the ACE-UNIX kernel. As can be deduced from the description, only data, located in the VAX's P0 space, may grow upward, and the stack, located in the P1 space, may grow downward. User specifiable shared memory is only allowed to be in the shared memory region. This paging implementation uses a new object file format (0413). It presumably is COFF with a 0413 a.out header as optional header. This a.out header has been updated to allow starting text and data on separate segment boundaries so that they do not fall in the same region. Apart from these remarks, the paging implementation does not show in the external kernel specification.

5. Benefits

The gains of **indirect** sections are immense.

Rough calculations on the programs in "/bin" and "/usr/bin" show the following about disk space. The disk space occupied by the 118 stripped programs in "/bin" on a particular system running the standard ACE-UNIX distribution was 1395k bytes; that is about 12k bytes per program. For "/usr/bin" these counts were 149 programs using 2195k bytes, or 15k bytes per program. Inspections of the symbol tables of the original non-stripped versions showed that about 4.5k bytes per program was occupied by the text segment of the C-library with standard IO. Not taking the small .data sections of the libraries into account, a reasonable estimate is a disk space reduction of 30 percent.

Loading of programs is faster too: text segments of shared libraries often already are loaded by other programs running concurrently. The amount of bytes to be copied from the file system to user memory because of loading thus also will be reduced by about 30 percent.

Once programs are running and paged or swapped, shared segments are treated independently. Their images are allocated and swapped out only once. That reduces memory usage as well as swap IO traffic.

On the negative side of the balance is that for each program load more files have to be opened by the kernel causing more lookup operations and more scattered file IO.

September 3, 1985

6. Future developments

Starting from the given specification of ACE-COFF new applications are being developed and new features are added:

- The user segment interface looks like the file interface. Segments must be opened first; a (second) set of file descriptors is used for identification. A segment create creates a segment from the section header passed as parameter. System calls such as open, close, fstat, etc. are used for other operations. Where with the file interface a path name is used, now a section header is passed. The interface is compatible with the memory mapped file interface.

- Memory mapped files will be implemented as segments having an association with a regular file. The corresponding ACE-COFF section will be of a new type, **associated section**, similar to an indirect section but without the requirement that the target file is a COFF object file.

- Inter process communication may use message boxes: these will be implemented as shared segments; a new attribute, **position independent**, indicating indifference about the logical location of the segment by the process, is then needed.

- Binding between images and segments must be refined: the sticky bit trick of UNIX can be generalised by letting the kernel keep segment images in memory or on swap space as long as their space is not needed for other reasons. That gives the effect of **segment caching**. Several layers of binding strength may be used: the sticky bit is just a hint then.

- With paging the ACE-UNIX map structures will be associated with sets of page table entries mapping a contiguous set of logical addresses. The unit of maintaining referenced and modified information then is the page.

7. Conclusions

The benefits of using indirect sections have been elaborated upon in a preceding section: a reduced disk space usage, reduced main memory usage, faster loading of programs, and less swapping.

By the generalisation of segment handling, a very sound base has been developed for future extensions: MMU handling has been more formalised, porting to other MMUs will be easier, and adding new functionality (as mentioned in the previous section) is easier.

Acknowledgements

ACE-COFF is the result of many fruitful discussions both within ACE and within Philips Data Systems in Apeldoorn. Willem Wakker of ACE is responsible for the ACE-COFF compilers and link editor; together with Anton Vriendts of Philips, he did the tedious job of exploring the odd

September 3, 1985

edges of the System V COFF linkage editor. The ACE people were always ready to provide the required support on many fronts. Especially Hans Schipper working on the ACE-UNIX kernel was of great help when implementing the COFF functionality into the kernel. Henk Hesselink's comments improved the paper considerably. Marius Schoorel and his colleagues of NIKHEF-H in Amsterdam were very helpful when making the figures.

References

[AT&T83] AT&T Bell Laboratories,
UNIX System V Release 2.0 User Reference Manual;
AT&T Bell Laboratories, December 1983.

[AT&T85] AT&T Bell Laboratories,
System V Interface Definition;
AT&T Bell Laboratories, Spring 1985.

[BabaogluJoy81] O. Babaoglu and W. Joy,
Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits;
Operating Systems Review Vol. 15, No. 5, December 1981, pp. 78-86.

[McCormick84] J.M. McCormick,
UNIX System Paging Design for a VAX Processor;
AT&T Bell Laboratories, November 14, 1984, File Case 40125-100, 45422-841114.01IM

[McKusick85] M. Kirk McKusick, Mike Karels, Sam Leffler
Performance Improvements and Functional Enhancements in 4.3BSD;
USENIX Association Summer Conference Proceedings Portland 1985, June 11-14, 1985, pp. 519-531.

[Organick72] E.I. Organick,
The Multics System;
The Massachusetts Institute of Technology, 1972.

[VanKonijnenburg85] E. van Konijnenburg,
Benchmarking UNIX Systems;
ACE Associated Computer Experts bv, to appear.

[VanSomeren84] Hans van Someren,
Paging in Berkeley UNIX;
Delft University of Technology, Afdeling der Elektrotechniek, Lab. voor Schakeltechniek en techniek van de informatieverwerkende machines, February 29, 1984.

[VanSomeren85] Hans van Someren,
COFF Loader User Reference Manual;
ACE Associated Computer Experts bv, September 6, 1984.

September 3, 1985

[Thompson78] K. Thompson,
UNIX Implementation;
The Bell System Technical Journal, Vol. 57, No.6, Part 2, July-August 1978, pp. 1931-1946.

[X/OPEN85] The X/OPEN group, Bull, ICL, Nixdorf, Olivetti, Philips, Siemens,
The X/OPEN Portability Guide, Part II: The X/OPEN System V Specification;
Elsevier Science Publishers bv, Amsterdam, July 1985.

September 3, 1985

NETWORKING WITH ISO AND UNIX (*)

Sylvain Langlois, Alain Gauteron

ROSE Project
BULL, PC 33/05
68 Route de Versailles,
78430 Louveciennes, FRANCE

mcvax!vmucnam!lvbull!sylvain
mcvax!vmucnam!lvbull!alain

ABSTRACT

During the last 5 years, operating systems have been evolving towards new concepts and design in order to integrate inter-systems communications. The work done by international standardization bodies within ISO has been conceptualized into the Basic Reference Model for Open Systems Interconnection. The protocols on which this Model is based have now reached a level where they are quite well specified (state of International Standards), and thus can be incorporated into operating systems.

This paper presents an application of the OSI Model to the case of a Local Area Network environment. It mostly concerns the four lower layers of the Model. All the protocols used here are those specified by ISO. The Class 4 Transport Protocol has been reworked and simplified to operate in a Local Area networking context, but the resulting implementation enforces ISO conformance rules. It covers integration of this set of protocols into the 4.2BSD UNIX kernel, and details changes made to the standard Berkeley code. A brief overview of the current development of an X25 environment, and a LAN/WAN gateway is also given.

This work has been developed for the ROSE/ESPRIT Project of which a presentation has been made at Cambridge EUUG Conference (September 1984). The implementation has been done on a BULL Mini6 with Berkeley networking code added to a V7 kernel.

September 11, 1985

(*) UNIX is a trademark of AT&T, Bell Laboratories

NETWORKING WITH ISO AND UNIX (*)

Sylvain Langlois, Alain Gauteron

ROSE Project
BULL, PC 33/05
68 Route de Versailles,
78430 Louveciennes, FRANCE

mcvax!vmucnam!lvbull!sylvain
mcvax!vmucnam!lvbull!alain

1. Introduction

Until now, computer communications within the UNIX world is mainly following two directions:

- the *UUCP* software [NOWITZ et al. - 78] is using asynchronous telephone lines to perform file transfer and some limited remote execution commands between UNIX machines only
- a more sophisticated set of protocols, available since 4.2BSD and known as the DoD Internet Architecture, built on top of TCP/IP protocols, allows communications between UNIX and non-UNIX machines, in a very reliable way.

On the other hand, international standardization bodies have come now to a stable communication architecture, by developing an *Open System Interconnection Reference Model* [ISO/7498 - 83] defined by a set of protocols which, for the most of them, have now reached the state of International Standards. It is not going too far to say that in the very near future most of the systems, whatever the hardware they may be implemented on, should be reachable whether through Wide Area Networks (such as the numerous specialized data networks already existing in many countries), or Local Area Networks (within a private company, university campus, ...) using a basic common set of internationaly approved protocols.

In 1984, the European Economic Commission launched the ROSE Project (*Research Open System for Europe*, ex EIES), within the *European Strategic Program in Information Technology* (ESPRIT) with the aim of implementing an operational network based on the ISO protocols under the UNIX Operating System, and offering to all the different ESPRIT contractors services such as: message passing, file transfer, remote login and execution, document transfer, ... [BITTLESTONE et al. - 84]. Starting with standard UNIX applications (*mail, news*, ...) using *uucp* and *cu* facilities over asynchronous lines or PAD to PAD connection, developments are evolving gradually in accordance with international standardization, to finally allow distribution of applications over a set of UNIX and non-UNIX interworking systems, as shown on figure 1.

(*) UNIX is a trademark of AT&T, Bell Laboratories

MHS(*)	FTAM(**)	
Session (BAS + BCS + BSS)		
Transport CI3/2	Transport CI4	Gateways
X25	CSMA/CD LAN	

(*) Message Handling System
(**) File Transfer Protocol

Fig. 1: ROSE Architecture (end of year 1)

2. The OSI Reference Model and UNIX kernel

The main problem to solve in this development was to clearly define which part of the OSI Architecture should be integrated into the UNIX kernel, and, symmetrically, what can be developed as user level software. Mainly, three constraints were to be taken in account:

1) *guaranty kernel homogeneousness*: due to its monolithic nature, maintenance of a UNIX kernel is not trivial, and this especially applies to the V7 and System III versions. This underlined the need of clear kernel structuration,

2) *make it portable*: as shown on figure 2, the OSI Model can be split into two subsets: protocols used for "pure" data transfer and those which actually process data in order to offer the expected service to the user application. Obviously, hardware dependent (or related) protocols had to be implemented inside the kernel,

3) *take care of performance*: optimal memory management to avoid multiple copies from one protocol space to the other, management of physical communication resources, such as X25 circuits, or CPU load are strong arguments which influence the kernel additions.

Data processing	Application Layer
	Presentation Layer
	Session Layer
Data Transfer	Transport Layer
	Network Layer
	Link Layer
	Physical Layer

Fig. 2: OSI Reference Model

As described in the above remark 2), it is quite natural to think that Transport layer Service is the obvious bridge between these two abstract subsets of the Model. The connection multiplexing facility was also leading to integration of Transport Protocol in the kernel. Some experiments made with Transport Protocol at user level have strongly underlined that most of the UNIX versions were not appropriate to such a design (no shared memory among processes, lack of interprocess communication means) [MARTIN et al. 84].

The *driver* approach, and the lack of flexibility of the standard *open(2), read(2), write(2), ioctl(2), close(2)* interface provided by the V7 and SIII versions were not exactly adapted to what we wanted to do. The OSI Model assumes a clear hierarchy among all layers, and a strong cooperation between protocols. The Service

concept covers many features which cannot be ''cleanly'' implemented in these versions.

On the contrary, the 4.2BSD offers a more flexible and superior system environment: the new interprocess mechanism (*sockets*) and the memory management features (*mbufs*) [LEFFLER et al. - 82] made it more attractive. The DoD TCP/IP protocols implementation was also a good example to start from.

However, the Rose Project was committed to first deliver an implementation in a V7 Unix environment and a first implementation of the ISO protocols has been done in such a way. As it has been thought that all versions of the system should be impacted, we did it using the 4.2BSD version and some work has also begun in a SV environment. In any case, all the user level software (Session, Message Handling System) will be adapted for all UNIX versions.

3. Architecture

3.1. Protocols

Our work began with the development of a Local Area Network environment, following ISO Architecture. We are now extending this by incorporating a WAN environment as well.

The LAN protocols are based on a CSMA/CD technology, and consists of those specified by [ECMA/82 - 83] and [ECMA/81 - 83], which represent the ECMA version of the ETHERNET protocol.

In order to facilitate interconnection of different LANs to build up a global network, we used the ISO Internet Protocol [ISO/8473 - 84] at the network layer level. It offers a sub-network independent Network Service, operating in a datagram mode, to the Transport Layer and provides a fragmentation facility which makes use of different underlying LANs possible. This protocol is very similar to the DoD IP Protocol, though they are totaly incompatible. Nevertheless, they provide approximatively the same set of options, so that the Network Service can be customized to what is exactly required by the upper layer entity (source routing, route recording, security, ...). Another feature of this protocol is variable length addresses. This can be extremly useful when the protocol operates over multiple interworking LANs and encourages multi-vendor architectures. It may be all the more interesting as there is no addressing standard currently approved; it allows use of this protocol with different addressing schemes for now, without waiting for the standardization bodies to agree on one.

The Transport Layer gathers two different protocols. The Connectionless Transport Protocol [ISO/8602 - 84] provides the user with the means of sending a single TSDU (*Transport Service Data Unit*) to another in a datagram fashion: no logical relationship is maintained among multiple TSDUs. It is primarily intended to serve applications which do not require a high reliability, using simpler mechanisms than connection-oriented protocols (mainly transactional-like applications). This protocol has not reached its final specifications within ISO, but it has been taken into account here because of its simplicity: we thought it was a good, and easy, way of testing the lower layers implementations.
At the time we started this work, the Connection Oriented Transport Protocol specifications [ISO/8073a - 83] were not satisfactory with respect to the use of this Protocol on top of a connectionless Network Service. It is obvious that the LAN environment we wanted to build was requiring use of a Class 4-like Transport Protocol. The provided Network Service needed a Transport Protocol able to detect and recover from transmission errors, possible data loss, duplication or misordering. It was not in our mind to re-write our own Transport Protocol, but rather re-work the ISO specifications and adapt this protocol to a local area networking context. Overall, we wanted to have an implementation which enforced ISO conformance rules. So we better thought about establishing some simplifications in the existing protocol which can be seen as implementation choices for optimization in a LAN context (see 3.3) [LANGLOIS et al. - 84].

3.2. Addressing

According to the protocol architecture, and with the aim of facilitating network interconnections, the Rose Project chose a hierarchical type for Transport level addresses (TSAPs), in a human readable form (this latter point allows easy mapping of names in some applications which could be directly interfaced to the Transport protocol, such as network management facilities). These addresses are built on a 19 bytes character string, as follows:

<site-name> <host-name> <transport selector>

where the site length is 8 bytes, the host also 8 bytes, and the selector 2 bytes. The transport selector is used as an application switch. An extra header byte (value 40 in hex) has been added for ECMA compatibility reasons [ECMA/20 - 84]).

Things are less simple for determining the network address structure. This problem is currently discussed within ISO. The only published recommendation [ISO/DAD2 - 84] still needs some work to be done and this problem hasn't been solved among ROSE contractors. For the first implementation, we are using an ARPA IP Class A type address, specified on a 4 byte quantity, divided in two fields:

<site-number> <host-number>

(site-number is defined on 7 bits, host-number on 24 bits, with a null bit header). This scheme assumes that only one Transport Protocol sits on top of the network layer, and thus doesn't fit to our architecture. We think about switching to the following address structure:

<site-number> <host-number> <network-selector>

as soon as this is agreed by all ROSE partners. This last structure can have a variable size.

3.3. Transport Protocol

3.3.1. Protocol Options

We decided not to use the "extended format" for Transport Protocol Data Units (TPDUs) numbering, and the "checksum" facility. The ETHERNET is already using an FCS field to detect data corruption, and the CNP Protocol is also checksumming the header of received datagrams. There was no need to check again TPDUs validity, and ,in doing so, this was saving processing time.

However, for compatibility with other implementations, the "checksum" option had to be supported if required by a remote entity initiating a Transport connection. At the same time, many TPDUs parameters (such as "version number", "security" parameters, "reassignement time", ...) are not used, and simply skipped and ignored if present in a received TPDU.

3.4. Multiplexing

As the Network Service was datagram based, talking about multiplexing multiple Transport connections onto a single Network connection was irrelevant. However, multiple Transport connections can be established between the same pairs of NSAPs.

3.5. Connection / Disconnection Phase

When initiating a Transport connection, our implementation should propose:

- non use of checksum,
- exclusive use of Class 4 (i.e. the "Alternative Protocol Class" is never sent),
- use of normal format,
- a maximum TPDU size of 1024 bytes (thus, a TPDU with the network protocol header can fit into an ETHERNET frame data area),
- use of expedited data.

3.6. Protocol error and disconnection phase

The disconnection phase has not been changed. But the protocol error handler has been slightly modified. When the local Transport entity detects some misfunctionning in the remote entity behaviour, it will initiate a disconnection rather than reporting an error via an ER TPDU. On the other hand, when an ER TPDU is received, the disconnection phase is entered.

3.7. Data Transfer

To allow high speed data transfer, our implementation never reduces the upper edge of a transmit window: if the protocol is getting short of space, it will let the remote credit fall down to zero, but once a credit has been sent, the protocol makes sure that enough space is available to handle incoming data. However, upper window edge reduction shall be supported if received from a remote entity.

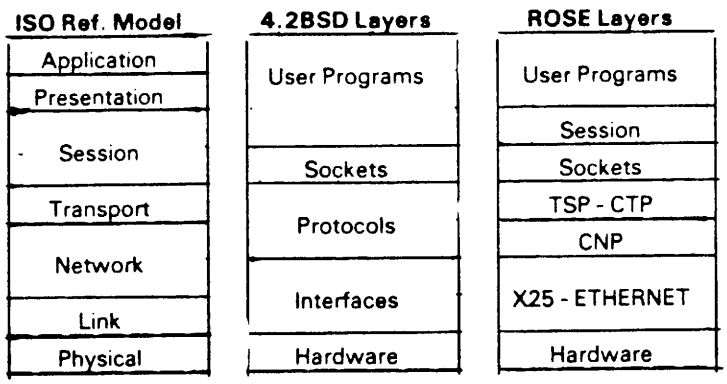
For the previous reason, there is no need to use the TPDU sub-sequence numbering scheme. But, on the other hand, the "Flow Control Confirmation" parameter has to be supported. If it is not used under "normal" conditions operating, it has to be sent when the remote Transport entity reduces, and increases the upper window edge back again, to make sure that both entities update their emission and reception credits in a correct manner and then synchronize the data transfer.

3.8. Protocol validation

The resulting protocol still enforces ISO conformance rules: as we said before, the protocol has not been changed in itself, we just optimized its implementation according to the environment we had. The re-written state tables have been automaticaly checked, using the LISE formal validator [ANSART et al. 83]. Some of the fixes we made have been reported to the ISO Transport Comittee and incorporated in the last version of the protocol specifications [ISO/8073b - 84].

4. Implementation in a 4.2BSD kernel

The LAN protocols organization was easy to integrate as its design is similar to the one used for TCP/IP implementation. Things were getting more difficult when integrating the WAN environment. X25 boards are now appearing on the market place, and we thought it would be more useful to code the X25 protocol at the 4.2BSD interface level, so that, one could easily remove this part of software and change it to a hardware X25, when available. Figure 3 details kernel organization.



TSP: Transport Protocol (Cl 2/3/4)
CTP: Connectionless Transport Protocol
CNP: Connectionless Network Protocol (over ETHERNET only)

Fig 3: OSI Model and kernel layering

4.1. Integration of a new Protocol Family

If it can't be denied that the networking facilities incorparated in the Berkeley UNIX system [LEFFLER et al. - 82] are a major enhancement to the UNIX system, it is also true that this particular part of kernel code (*socket, protocols and network interface* code) was designed around TCP/IP protocols and is making strong assumptions about network addresses, some data structures and protocols functionnalities. On the other hand, most of the network interface code assumes that the Service provided by the network is connectionless, which is not the case when using the X25 protocol. We didn't want to introduce too many modifications in the kernel code, to stay compatible with the standard Berkeley distribution. Nevertheless, integration of an *AF_ISO* family required some changes. One can also ask about the long term utility of having multiple families inside the same kernel: the size of the resulting kernel makes it hardly portable on small workstations, which was not our objective and one can hope that everybody is going to adopt ISO standards soon.

Besides declaration of new queues, bringing up the *AF_ISO* family obliged us to extend the *sockaddr* structure size to be able to pass all the addressing information down to the kernel. Because we wanted to take advantage of the already existing address manipulation code, we defined two compatible *sockaddr* structures within one! Berkeley had all these user library routines to format an *in_addr* and we had nearly the same network level address format, so we thought it worth keeping these utilities, so that */etc/hosts, /etc/networks,* ...could be used without any modification. We had to have two addressing structures to be passed to the kernel, the TSAP and the NSAP. The use of TSAP is limited to the Transport Protocol and is not used at all below this layer: under such conditions, it is possible to have a *sockaddr* structure known at user level which includes the TSAP, and another *sockaddr_iso* structure known by the kernel code. In doing so, we had to increase the data area of the *sockaddr* structure up to 32 bytes (was 14 bytes before). This doesn't have any consequence on the other families code (*AF_UNIX, AF_INET*). The new structures are detailed in figure 4.

```
struct sockaddr_ul {
        struct sockaddr_iso             ul_saddr;/* kernel sockaddr */
        char                            ul_tsapid [19] ;/* TSAPID as a string */
};

struct sockaddr_iso {
        short       siso_family;        /* protocol family */
        u_short     siso_tsel;          /* TSP selector */
        struct      iso_addr;           /* kernel NSAP */
        char        siso_zero [8] ;     /* padding area */
};
```

Fig 4: Address structures

4.2. Transport Connection Parameters

Transport Service makes extensive use of "user" defined parameters, such as Service options (expedited data) and Quality Of Service parameters (QOS), (maximum transit delay, ...). These parameters are initialized by the connection initiator: we modified the *setsockopt(2)* and *getsockopt(2)* to operate directly at Transport level (*). They are negociated between the two users involved in the communication. The receiving user level application needs to indicate to the Transport Protocol that negociation has been performed (after possible changes of these values) for the protocol to complete its three way handshake. To do so, we have introduced a new system call, *confirm(2)*, which is to be invoked after the listening process does an *accept(2)*. If *confirm( )* is not issued, the Transport Protocol will assume, after a certain amount of time, that the user process is refusing the connection and will automatically disconnect the peer.

4.3. Data transfer

When using CTP, data transfer takes place quite easily in the standard Berkeley code, and is done in the same way as UDP. ISO Transport Service doesn't specify any maximum size to TSDUs. It assumes that data received from the upper layer in a single shot forms a single TSDU (i.e. the upper level process does not segment its data). It is up to the Transport Protocol to segment it according to its current TPDUs maximum length. The existing socket buffer code was making the assumption that the underlying protocols were either connection and byte stream oriented, in which case the upper level application is segmenting data itself (data passed to TCP are up to 2K bytes), or connectionless and then, data exchange between user space and kernel space was made on an atomic basis (*PR_ATOMIC* and *PR_ADDR* flags set): for example UDP has been implemented such that its maximum send and receive queue sizes are 2K bytes each. These constraints were not acceptable when implementing the ISO Transport data transfer phase, which is block oriented and delivers EOT indications (in contrast, TCP is stream oriented and uses a "push" mechanism). However flexible the implementation had to be, we had to keep the code compatible with the existing protocols and applications. Supporting applications transmitting huge amount of data (voice or image transfer) has to be done in such a way as to respect user required options (non blocking I/Os).

We then modified the *send(2)* system call (via the *sosend* routine) to allow Transport Service to exercize flow control against the user in the following way:

- the protocol satisfies user process requests up to the space it has available in its retransmission queue. When functionning in a non-blocking mode, the protocol will block the user if the data can not be handled in a single shot up to the time some space becomes available (when some previous data are acknowledged and then removed from the retransmit queue). It mustn't be forgotten that the Session Protocol is supposed to negociate the size of the messages it is going to transfer; thus, it seemed reasonable to fix the length as an implementation dependant constant (which is up to 8K bytes in our current code)

(*) These routines are also used to pass user data during the connection establishment phase.

depending on the machine ressources.

In the reception mode, the upper level process is supposed to be passed a whole TSDU also. In this case, we can't do anything else than segmenting the TSDU if a non-blocking *recv( )* is invoked, asking for a length of data which is longer than that which can be stored in the receive queue. We introduced a new socket buffer flag in the *recv* call, *MSG_MOREDATA* indicating to the user process that the remaining TSDU has not been received yet. That's why, we made *recv* return a *long* instead of an *int*: this flag is represented in the high bytes of the *long*, while the count of actually received bytes stands on the low order bytes. This trick is still compatible with the standard *recv* call, and thus, some 4.2BSD applications can be used with this modified system call without knowing what happened.

The expedited data transfer also differs from that used by TCP, but both modes can be solved in the same way. A problem of synchronization between normal and expedited flows may appear, but this part of the code is still under test.

4.4. X25 Interface

The design of the *interface* code supposes that all exchange between this level and the hardware level are using the same "packet" type, which is directly passed to IP. In our architecture, an incoming packet can be either a CNP datagram, in which case it is passed to the CNP protocol on software interrupt, using an *isointrq*, or an X25 packet, in which case it is analysed by the X25 driver. We had to add some "glue" to the transport Class3 interface to the network layer to perform specific X25 processing, namely:

- generate X25 requests (connection and disconnection requests),
- respond to specific X25 events, such as connection, disconnection, and reset indications,
- maintain X25 protocol control blocks (basically, virtual circuit management). All these operation are made transparent to the transport entity. In fact, this part is not fully satisfying as it should be handled by a proper X25 *ifnet* structure, but would have resulted into major modifications of the existing code.

4.5. LAN/WAN Gateway

A LAN/WAN gateway is now being developed in order to enable communications between ETHERNET and X25 networks. This bridge simply consists of a Transport relay function, using a Transport Class4 - Transport Class 3 back to back facility. As both of these protocols provide the same Service, the gateway mainly translates indication primitives of one Transport Class into requests of the other Class. It also exercizes flow control between the two networks.

5. Conclusion

Our work introduced some modifications in the standard 4.2BSD. The implementation of the ISO Transport Protocol could be seen as a first step in implementing the *sequenced packet socket* facility, but it might need to be done in a slightly different way which can result in major changes in the 4.2 BSD existing code. We also think that all the address related code needs to be reviewed to be more Protocol Family independant than it actually is. This last point would surely make integration of new families much easier. The resulting additions consists in about 90K bytes of new code inside the kernel. The code is being ported on a VAX: this is hoped to reduce its size a little bit, because of some machine dependant adaptation we had to make on the Level 6 (related to some memory alignement problems). The following steps of our work is now to interface the ISO Session to this new kernel. The Session code is already working in a V7 environment and shouldn't be too difficult to adapt. We would also like to use some standard Berkeley applications (*TELNET* for example) with the *AF_ISO* family and make sure that the modifications don't have any side effects on the existing code. We already have *TFTP* working over both CTP and UDP, with about 30 modified lines in TFTP sources only.

6. Acknowledgements

We would like to hereby thank all the people who helped us in our work. Jim Loveluck, Jacques Bernadat, Gerard Vandome (BULL) for the discussions we had while establishing this architecture, Kevin Porter (Ferranti Infographics Ltd) who implemented an ECMA version of Transport, for accepting to discuss about his work, Jean-Pierre Ansart (ADI) and Tomasso Ricci (OLIVETTI) for the help when re-writing the Transport Protocol state tables, and all the people involved in the ROSE Project who paid attention to what we were doing.

7. References

[ANSART et al. - 83] : *Validation, Description and Testing Tools*, Jean-Pierre Ansart, Vijaya Chari, Didier Simon, Omar Rafiq, RHIN Project, Agence de l'Informatique, Paris, December 1983,

- [BITTLESTONE et al. - 84] : *Implementation of OSI Protocols in the ESPRIT Information Exchange System", R. Bittleston, R. Cadwallader, A. Diedew, M. Elie, J. Loveluck, F. Lung, S. Miess, S. Pozzana, ESPRIT'84: Status Report of Ongoing Work, Bruxelles, September 1984,*
- [ECMA/20 - 84] : *Layer 1 to 4 Addressing* , ECMA Technical Report TR/20, March 1984;
- [ECMA/82 - 83] : *Local Area Networks CSMA / CD Baseband: Link Level*, ECMA 82 Standard, August 1983;
- [ISO/7498 - 83] : *Data Processing, Open System Interconnection - Basic Reference Model*, ISO/DIS 7498, November 1983;
- [ISO/8073a - 84] : *Connection Oriented Transport Protocol Specification* ISO/DIS 8073 (Ottawa version), October 1983;
- [ISO/8073b - 84] : *Connection Oriented Transport Protocol Specification* ISO/IS 8073 (Copenhagen version), June 1984;
- [ISO/8473 - 84] : *Information Processing Systems, Data Communications Protocol for Providing the Connectionless Mode Network Service*, ISO DIS 8473, September 1984;
- [ISO/8602 - 84] : *Protocol for Providing the Connectionless Mode Transport Service*, ISO DP 8602, September 1984;

[ISO/DAD2 - 84] : *Addendum to the Network Service Definition Covering Network Layer Addressing*, ISO Second DP 8348/DAD 2, October 1984;

- [JOY et al. - 82] : *4.2BSD System Manual*, W.N. Joy, E. Cooper, R. Fabry, S. Leffler, M. Mc Kusik, Technical Report 5, University of California, Berkeley, September 1982;
- [LEFFLER - 83] : *A 4.2BSD Interprocess Communication Primer*, S.J. Leffler, University of California, Berkeley, February 1983;
- [LEFFLER et al. - 82] : *4.2BSD Networking Implementation Notes*, S.J. Leffler, W.N. Joy, R.S. Fabry, University of California, Berkeley, September 1982;
- [LANGLOIS et al. - 84] : *Implémentation du protocole de Transport ISO Classe 4 sur Réseau Local de type CSMA / CD*, S. Langlois, J.P. Ansart, Proceedings of the *De Nouvelles Architectures Pour les Communications* Conference, Ed. Eyrolles, Paris, September 1984;
- [MARTIN et al. - 84] : *Problémes posés par le développement de logiciels réseaux sous UNIX*, B. Martin, H.C. Lucas, V. Merrien, Proceedings of the "De Nouvelles Architectures Pour Les Communications", Conference, Ed. Eyrolles, Paris, Septembre 1984;
- [NOWITZ et al. - 78] : *A Dial-Up Network of UNIX Systems*, D.A. Nowitz, M.E. Lesk, UNIX Programmer's Manual vol. 2c, Seventh Edition, Bell Laboratories, August 1978;
- [ROSE - 85] : *ROSE Technical Specifications*, Version 2, January 1985;

Sendmail Now and Its Next Generation

Miriam Amos
Digital Equipment Corporation
Computer Systems Research Group
University of California
Berkeley CA 94720

The University of California at Berkeley released *Sendmail,* an internetwork mail routing facility, with its 4.2BSD release almost two years ago. Since then *sendmail* has become the comparative yardstick for UNIX mail systems. However, because *sendmail's* size, complexity, and flexibility, some consider it to be overkill. This paper looks at *sendmail* as it is now and discusses the points of consideration for the next generation of the Berkeley mail system.

The first section describes *sendmail's* features and design considerations. The second section describes the design goals and considerations in the evolving Berkeley mail system.

1. Sendmail Today

Sendmail is an internetwork mail routing facility. Features include aliasing and forwarding, queueing, an implementation of SMTP (the Simple Mail Transfer Protocol [RFC821]), automatic routing to network gateways, and flexible configuration [Allman85]. *Sendmail* does not interact with the user nor perform the actual delivery of the message. It simply collects the message generated by either a user interface, such as, Berkeley's Mail [Shoens83] or MH [Borden79], or an intermediate mailing channel. Once *sendmail* collects the message, it manipulates the header as needed for validity at the destination, and then passes the message on in the next step of delivery.

Sendmail interacts with many diverse networks. Some networks provide a point-to-point routing as in UUCP [Nowitz78], while others provide only end-to-end addressing as in DECnet. Some use a left-associative syntax, while others use a right-associative syntax. When both are mixed, ambiguity in the proper interpretation of the address arises. These differences between networks are why *sendmail* exists and why it performs the functions it does. *Sendmail* as an internetwork mail router bridges these networks with their different syntaxes and semantics.

1.1. Original Design Goals

Sendmail was designed with a number of goals in mind [Allman83]. In particular:

- *Sendmail* was to be compatible with existing mail facilities.
- No message was to be lost. Once a message was accepted for delivery, it was to be delivered, returned, or passed to a human for some appropriate action.
- Existing software was to be used when possible.
- *Sendmail* was to be an expandable system so it could handle the changing mail environment.
- The configuration information was not to be compiled into the code to simplify installation.

- Individual forwarding and access to mailing lists were to be provided such that both could be performed without modification to a system-wide alias file.
- The selection of a mailer was to be placed at the individual's choice, allowing the user to create his/her desired environment.
- Mail for a given destination host was to be batched when possible to reduce network traffic.

1.2. Features

The actual implementation of *sendmail* meets its designer's goals [Allman85]. *Sendmail* successfully provides a reliable and flexible mail system. What follows is a description of *sendmail's* features and functionality.

1.2.1. Message Delivery

When a message is to be sent, the generator of the message communicates with *sendmail* in one of three ways. It can call *sendmail* via the standard UNIX method for communication with a process, or by invoking SMTP over pipes, or by invoking SMTP over the 4.2BSD IPC mechanism. By whatever means it is called, *sendmail* first does a preliminary verification of the recipient list. This includes syntax checking, alias expansion, and verification of local addresses. It should be noted that files and programs are valid recipients. The actual verification on non-local recipients is done at delivery.

Next *sendmail* collects the message. The message consists of two sections — a header section and a body section. The header section is made up of ASCII lines in specific formats with information relating to the delivery of the message. The body section is ASCII text. The header is parsed and stored within *sendmail,* while the body of the message is stored in a temporary file.

Once the message is collected, *sendmail* attempts to deliver the message. Message delivery is optimized by batching mail according to the destination site and mailer. *Sendmail* then calls the mailer by one of the above methods used for its own invocation. If the mailer is invoked by the first method, then the recipients are passed as arguments while the body of the message is passed via standard input. The latter methods pass the recipients one at a time according to the Simple Mail Transfer Protocol to the mailer's standard input. Once all the recipients have been transferred, the body of the message is then sent to the mailer's standard input. *Sendmail* does compensate for mailers that can only deliver to one recipient at a time.

Sendmail checks the mailer exit status upon completion and reports to the sender if an error occurred. Additional error information can be sent by the mailer over it standard output to *sendmail. Sendmail* will then return the undelivered message, complete with the error information, to the originator. The contents of the error message will usually contain the host and recipients it was unable to deliver to, as well as why it was unable to deliver the message.

1.2.2. SMTP

SMTP (as defined in [RFC821]) was included as the "backbone" *sendmail* protocol. SMTP controls mail message transfers over the ARPAnet. The incorporation of SMTP into *sendmail* resulted in the inclusion of some desirable features to the actual mail delivery, such as, multiple recipients per message, control over the status of each individual recipient, and requeueing of an undelivered message. The inclusion of SMTP precipitated the need for queueing.

1.2.3. Queueing

Queueing improved the reliability and performance. Once a message is place in the queue, it is preserved and can survive most causes of message loss. Queueing allows *sendmail* to be responsive to user agents, because it can return as soon as the message is "checkpointed" into the queue.

Sendmail provides some interaction with the queue. The decision of when to queue a message can be controlled by the configuration file /usr/lib/sendmail.cf. In the configuration, file the option **x** can be set to a decimal value. The value of **x** is used for comparison against the system load average. If the load average is greater than the value of **x**, *sendmail* will queue a message for later delivery. Otherwise *sendmail* will attempt immediate delivery of the message. If the delivery fails because of a temporary condition, the message can be requeued for later delivery.

1.2.4. Aliasing

Sendmail provides aliasing in multiple forms. First, system-wide aliases are provided in the /usr/lib/aliases file. This file gives system-wide consistency for mapping names to mailing lists.

The next level of aliasing is provided as a subset function within the /usr/lib/aliases file. By using inclusion a mailing list can specify a file that is contained outside of /usr/lib/aliases. This provides the consistent system-wide name of the mailing list, but it also allows the list to be maintained by an unprivileged user. This obviates the need to rebuild alias database when simply updating a group list.

At the lowest level of aliasing, an individual can set up forwarding. If a .forward file exists in the user's home directory, *sendmail* will read the file and forward the message to the indicated recipient list. This same mechanism can be used to select a private incoming mailer. To select a mailer one would create a .forward file with the contents:

"|/usr/local/newmail myname"

With all these levels *sendmail* furnishes a rich complement of aliasing capabilities.

1.2.5. Configuration File

The configuration file is an ASCII file that is read each time *sendmail* is executed. This file contains information for the interpretation of addresses, manipulation of headers, selection of a mailer for delivery, specifications for calling the mailer, and the controlling parameters for *sendmail's* execution. Within the configuration file running options can be manipulated without recompiling the program.

The configuration file provides *sendmail's* flexibility. However, this file is also the most difficult aspect of *sendmail* to understand. The rewrite rules (found in the configuration file) are what give *sendmail* its reputation for black magic. They are an ordered list of pattern-replacement rules, which are applied to each address. Each rule is divided into two parts — the pattern to be matched, and the replacement format it is to be rewritten as. Once the pattern is matched and rewritten, the rule is applied until it fails. To terminate a ruleset, either the address matches a rule where termination is indicated, or the address falls off the end. The difficulty is not in the interpretation of the individual rule, but how the rules tie together to perform their rewritting tasks.

2. The Next Generation of Sendmail

Sendmail was the successor to *delivermail,* Berkeley's first solution the the internetwork mail problem. While it has improved on *delivermail's* shortcomings, it has introduced its own set of flaws. The successor to *sendmail* will try to correct these flaws, hopefully without introducing new ones.

2.1. Design Goals

The majority of the design goals of the next Berkeley mail system are the same as those of *sendmail.* These goals are general design goals which should be adhered to when developing a mail system.

- Reliable delivery is a must: no message should be lost.
- Network use should be optimized by batching mail for a given destination host.
- The new system should be compatible with existing mail facilities.
- Aliasing should be provided.
- The user should be able to tailor his/her own mail environment.

Those goals that are specific to the next Berkeley mail system are:

- To provide a group of programs that together will provide the same functionality as *sendmail.* Breaking *sendmail* into several programs should ease the maintenance cost of the mail system.
- To utilize existing code when possible.
- To simplify the configuration file scheme.
- To provide a mail system that is specific to the Berkeley release.

2.1.1. Maintenance Issue

The goals of the next Berkeley mail system do not differ that much from the original goals of *sendmail.* What is different is the emphasis on maintainability. Many large sites must have a "sendmail wizard". This person must spend time in understanding the configuration file and internetwork addressing. A cost of software packages usually considers the initial expense, but also considers the expense of continual use of the package. This expense includes the manpower involve in maintaining the package. Taking this into account, *sendmail* has not been cheap.

This cost in manpower caused some sites to rewrite *sendmail* into a smaller, less complex, system such as SM, the mail system at Lucasfilm Ltd. [Ostby85]. By applying the KISS (Keep It Simple, Stupid) philosophy to *sendmail,* the next Berkeley mail system will be easier to understand and maintain.

2.1.2. Utilization of Existing Code

The reusing of existing software is an engineering decision. It is far more efficient and timely to reuse code. Because one the main goals of the project is to decompose *sendmail* into smaller programs, it only makes sense to use the existing code when feasible.

2.1.3. Configuration File

Parts of the configuration file which appear to be mostly static will be pulled back into the code. The rewrite rules for handling the addressing problem of an internetwork environment will still be handled outside of the code. Consideration is being given to following the Upas [Presotto85] concept of filters. A filter could exist for each network mailer. This would isolate the particular header munging needed when transferring mail

from one network type to another.

2.2. Components of the Next Generation

The new mail system will take the specific functions of *sendmail* and provide them in separate programs. Together these programs will provide an internetwork mail routing system.

Breaking *sendmail* into separate programs should simplify the mail system. It is realized that by forcing more programs to interact performance could be affected. A message will now have to be passed to the central control point. If the controller is unable to pass the message to the delivery agent or if the delivery agent cannot delivery the message at that time, the message will have to be passed to a queueing program for insertion into the queue. When the same message is later delivered, it will be copied again to the delivery agent. This scheme has two major performance considerations — the number of times a message will be copied before delivery and the number of forks or execs that are required to carry out the interaction between the different parts of the mail system. Great care will have to be taken to minimize the performance overhead.

2.2.1. Mail Crossbar

Mail crossbar (for lack of a better name) will be the main controller of the delivery procedure. This is the program that is called by the user agents and in turn calls the delivery agents. By keeping the concept of a central focus point, the mail crossbar can preserve the standard UNIX method for communications with a process used by *sendmail.* However, the mail crossbar will not implement SMTP as *sendmail* did, nor any specific mail protocol. The mail crossbar will act only as an mail exchange point.

The mail crossbar will be the decision point at which mail is passed on to a delivery agent. Depending on the status it returns, the message will be passed to a queueing program for insertion into a queue. Certain delivery agents (such as UUCP) have their own queueing, so it is not necessary to submit the message into the mail queue.

2.2.2. Queueing Routines

Breaking the queueing related functions into separate programs allows flexibility in how each of the programs can be implemented. Cleanqueue mentioned below could be implemented as a shell script that is run by cron. The queueing functions of *sendmail* and some new functions will be provided by five programs.

- *Queueup* to insert messages into the mail queue.
- *Runqueue* to process entries in the queue.
- *Printqueue* to print out the mail queue.
- *Cleanqueue* to remove expired messages and perform basic consistency checking.
- *Rmqueue* to remove jobs from the queue.

Queueing has the greatest potential for experimentation. Consideration has been given to using separate queue directories per delivery agent or even by host, as some UUCP implementations and MMDF (Multi-channel Memo Distribution Facility [Kingston]) have done. However, because the major mail delivery agents used are UUCP, local, and internet it is not clear that the a more complex queueing scheme would be beneficial. UUCP provides its own queueing thus eliminates itself from the queueing question. Local mail is almost always delivered immediately, so the current scheme of a single mail queue appears to fit the needs of the mail system.

2.2.3. Aliasing Routines

The manipulation of the alias file will be provided by two programs: *newaliases* to rebuild the alias file and *editaliases* to modify the alias database, providing access control and locking.

A past problem with *sendmail* has been the mutilation of the alias database. By providing separate utilities for the control of the database, access to who can modify the database and when the database is to be rebuilt can be more efficiently handled.

2.2.4. SMTP Server

A SMTP server will be provided as a separate mail delivery agent like /bin/mail the local mail delivery agent. Consideration was given to separating the SMTP server into two separate programs: one to handle incoming connections and another to handle outgoing connections. This scheme was use in the DECnet-Ultrix mailers and proved to simplify the implementation. However, this does not take advantage of the SMTP ability to turn a SMTP session around. SMTP has the ability to turn a connection around much the same way UUCP does. Because connections would not be taken down and put back up for the exchange of mail between two sites, the implementation of a single server with the turn around functionality would be a performance enhancement.

3. Summary

As has been discussed in this paper, it is the goal of the next generation of *sendmail* to continue to implement the best points of *sendmail* and to rectify its weaknesses.

REFERENCES

[Allman83] Allman, E., "Sendmail — An Internetwork Mail Router." In [UCB83], volume 2C. July 1983.

[Allman85] Allman, E., and Amos, M., "Sendmail Revisited." In *Proc. of Summer 1985 USENIX Conference.* Portland, Oregon. June 1985.

[Borden79] Borden, S., Gaines, R. S., and Shapior, N. Z., *The MH Message Handling System: Users' Manual.* R-2367-PAF. Rand Corporation. October 1979.

[Kingston] Kingston, D. P. *"MMDFII: A Technical Review"* Ballistic Research Laboratory. May, 1985.

[Nowitz78] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Ostby85] Ostby, E., and Kaplan, A., "SM: A SMALL MAILER" In *Proc. of Summer 1985 USENIX Conference.* Portland, Oregon. June 1985.

[Presotto85] Presotto, D., "Upas — a simpler approach to network mail" In *Proc. of Summer 1985 USENIX Conference.* Portland, Oregon. June 1985.

[RFC821] Postel, J. B., *Simple Mail Transfer Protocol.* RFC 821. Network Information Center, SRI International, Menlo Park, California. August 1982.

[RFC822] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Shoens83] Shoens, K. (revised by Leres, C.), "Mail Reference Manual (revised)." In [UCB83], volume 2C. July 1983.

[UCB83] *UNIX Programmer's Manual,* 4.2 Berkeley Software Distribution. University of California, Berkeley. August 1983. Derived from *UNIX Programmer's Manual, Seventh Edition.* Bell Laboratories, August 1978.

Remote File Systems on UNIX

Douglas P. Kingston III

Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam
The Netherlands

ABSTRACT

In the past few years, the nature of computing has changed dramatically as the technology has made it possible to provide computers for small groups or individual users, while sharing more expensive resources via networking. Unfortunately this has also created problems since it is still desirable to easily access data belonging to others that may now reside on another system. When this capability is provided across a network in such a way that the remote files retain the basic attributes of a local file, it is referred to a remote file system (RFS). While not in widespread use, there have been a large number of attempts at developing an RFS capability for the UNIX operating system. The scope of such work has varied greatly between different attempts. This reflects changes in the design criteria, the underlying capabilities expected in the UNIX system, and the extent to which people were willing to alter UNIX itself. The implementations have varied greatly in their transparency and efficiency, the two most important qualities of an RFS. In addition, the proprietary nature of many of the RFS implementations has greatly hampered their widespread acceptance by the UNIX community. The ideal RFS is totally transparent to all UNIX user processes, has no noticeable effect on operating system performance, and is available to be implemented on a variety of systems. The paper will survey many of the remote file systems that have been implemented on UNIX, explain their basic designs, and comment on how well they approximate the ideal RFS, at what cost and with what disadvantages.

Based on the preceding review, and the implementations that we have been able to obtain or test, we have undertaken to implement an RFS for 4.3BSD that will be generally available to the UNIX community. The RFS is implemented in the kernel for transparency and efficiency. Implementation is underway and a test system was completed two months ago. We will detail the design and implementation choices and report on current progress.

Introduction

The design and implementation of remote file systems is still an area of active research and experimentation in the UNIX community. This paper has three major sections. In the first we will try to outline the major issues in remote file system design. In the second we will describe a number of past and present remote file system implementations, and finally we will describe the system in development at BRL and CWI for 4.3BSD. The paper assumes some familiarity with the UNIX file system and the UNIX kernel layering.

Part 1: Remote File System Design Issues

Types of Remote File Systems

Remote file systems can be implemented at several levels in the UNIX system. The lowest level at which one can be implemented is the device driver level. The device driver interface consists of routines to open and close the device, and to read and write individual blocks on the device. Read and writes can be done with or without using the UNIX block cache (cooked vs. raw devices). A number of remote file systems have been done that used this interface to give remote access to disk segments on another machine. Generally a pseudo device driver is installed that packages block read and write requests and sends them via a network link to another host for processing. This is the simplest of the interfaces and is consequently very fast. Problems with this interface include lack of concurrency control and the required homogeneity of the systems involved. This implies identical directory format, byte ordering, word size, and user ids.

The next logical level at which to implement remote file access is at the inode access level. The operations here operate on an inode or use them as references to access the contents of files. Inodes are read and written with iget/iput or operations like chmod and stat, and reading and writing of data is done through rdwri(). In these implementations, inodes for remote files are marked, usually by using a new inode type ("remote"). Data is read or written from a logical location in a file and is not constrained to be block aligned or block sized. Concurrency control is less of a problem at this level since inodes are locked before modification. When inodes are manipulated remotely one must solve deadlock and race conditions on locking and freeing inodes. In particular, one must solve the problem of letting a remote machine lock your inodes and then fail to unlock them.

The next logical level to place a remote file system interface is at the system call level, and here there are two choices, just inside the kernel, or just outside. In both cases, system calls that refer to remote files are packaged and sent to a remote machine for interpretation, but the semantics of the two implementations are quite different. An implementation just outside the kernel must manage the assignment of file descriptors itself, and arrange for connections to remote sites using "hidden file descriptors". An external implementation also requires that all user programs be recompiled to include the networking code. The added overhead of doing multiple system calls for even local operations adversely impacts the performance of these implementations. An implementation just inside kernel can be more effective at emulating the exact semantics of the system calls, and requires little or no changes to any user-mode programs. It is often easier for kernel level implementations to share network connections, one of the bottlenecks for remote file systems. Kernel implementations also avoid the added system call overhead and context switching of user mode implementations. Unfortunately, kernel implementations are harder to implement and test.

Naming: Super roots vs. remote mounts

One of the most important aspects of a remote file system is how remote files are named. Everybody agrees that remote files should look just like regular files to users and programs, but there seem to be two schools of thought on how to actually reference the root of a remote directory hierarchy. The first is to have a special directory above the root of your file system which contains entries for each remote system. This is generally referred to as the "super root" technique. Accesses look like /../host/remotefile. This has the disadvantage that semantics of ".." in / are changed, but the advantage that a given file will always have the same name since /../host/remotefile names the same file on all systems including the local system regardless of the host interpreting the path.

The second technique is to mount remote directory hierarchies much like a local file system would be mounted, except that the file system mounted there is marked as being remote. Subsequent actions on the remote file system are handled by remote file system code instead of the regular file system code. In the case of remote disks, this comes naturally, since the file system code can not generally tell the difference between a local disk drive and a disk accessed over the network (except it is generally slower). As implied by the last paragraph, allowing files to have the same name on all machines is more difficult in this case since you can not be sure that the mount point is the same on all machines, but with cooperation this can be possible, and if symbolic links are available, even files on the local machine can have the same name. In particular if the special case of /common-mount-point/thishost/file is caught and treated as /file, then you can have the advantages of the super root without the disadvantages, but it is more difficult.

Whatever the naming scheme, one must be careful to avoid the possibility of infinite loops while evaluating paths. This is more likely in the case of remote mounts than with super roots. Symbolic links just make the problem worse.

Authentication Techniques

There have been two approaches to user identification and access abilities, the easy way and the hard way. Those that chose the easy way decided to rely on the standard UNIX access controls and did not bother to do any mapping of user ids across network connections. As a result, these systems require that a global /etc/passwd and /etc/group file be maintained identically on all systems. The hard way is to pass userids across the network symbolically and map them into equivalent userids on the remote machine. This method is required if you cannot arrange for globally maintained accounting files, and reduces the maintenance effort on the network at a small cost for id mapping. One of the problems with this method is that there are files that when listed on the remote system by a local process, will have no equivalent local user. These files are often listed as belonging to "nouser" and "nogroup" or some equivalent notation for an unknown remote user. In order to avoid mapping every remote userid as it is encountered on the remote system, this scheme is often used for every remote file that is not in the user or group of the user making the remote request.

Transparency of Implementations

The key quality by which to judge a remote file system implementation is the transparency of the implementation to users and more importantly, programs. In general this means that you must continue to support the existing UNIX syntax and semantics of file system operations. One of the best tests of this is the UNIX command **pwd**. This serves as a very good test that the system has correctly implemented the semantics of joining a file system to a mount point. This includes getting the entry into /etc/mtab, properly chdir'ing from the root of the remote system into a directory on the local system, and that the entry for the remote system can be found in the directory on the local system. This is non-trivial.

Another good test is to check the evaluation of long paths that when evaluated, traverse to another machine and back again (via chdir .. or symbolic link) and then on to a third machine. This forces proper evaluation of chdir .. from the root of the remote machine, which may not be the same for local and remote processes.

Yet another telling test is to cause a program that has chdir'ed to a remote system to dump core. The core file should get dumped on the remote system if the remote file system is truly transparent, but less transparent implementations may cause core dumps on the local system. This can be one of the most difficult parts to get right because it is so tightly coupled with the kernel.

The meaning of chdir ".." from the root of your local machine on a regular UNIX is that you are still in the root. It would be nice to preserve this feature, since there are programs that use this feature to discover they have reached the root directory.

Finally there is the issue of portability. The days of one vendor computing shops is slowly coming to an end, particularly in the micro to super-mini segment of the market. With more advanced programming interfaces and networking, it is now common to have a network of heterogeneous machines with different word lengths, byte ordering, and operating systems. To a remote file system implementor, this implies the need for machine independent protocols and canonical formats for the data passed between machines such as directories. In particular, there are now several different file system types in existence. These arise from the fact that directories contain binary information in the form of inode numbers and there are two basic directory formats, Version 6 and Berkeley 4.2BSD. The Berkeley 4.2BSD file system has even more binary data than Version 6. This leads to at least 4 to 6 different formats when byte ordering differences are combined with format differences, and there is no reason to believe that there won't be further changes or new directory formats now that there is a format independent interface. This almost dictates a canonical interface to the directory information rather than having each program know the format of directories. Unfortunately this issue was largely ignored until 4.2 was released. This canonicalization comes at some cost, and can be a significant performance factor, especially on CPU starved machines.

Part 2: Remote File Systems Past and Present

BRL Remote Disk

The BRL remote disk facility was designed to give remote access at the device driver level to block structured special files on a remote system. The system was built on top of DEC PCL11-B network hardware, which gave 8 to 16 megabits per second transfer rate over a 16bit DMA Unibus parallel link which was time division multiplexed.

Assignment of major/minor device codes was unique across all machines, and the devsw table had the host id of the machine on which the device was resident. Each kernel knew where to send a read or write request if the device in question was not on its machine. A static routing table told which network device and address to send the request to. This meant that multiple network devices could be used and routing would happen at little cost. For devices not resident on your system, a pseudo disk driver was entered in the devsw table. This device drive took the request and put it in a network message and shipped it off to the remote system. Raw disk access was not provided over the network.

The server was a kernel process much like the swapper. The server was awakened to handle any incoming network packets, and would enqueue the request with the appropriate device driver. Note that this procedure could recurse on a different network device if the requested device was not resident locally. Since the entire implementation was in the kernel, it was very fast. Speeds in excess of 100K bytes/second were not uncommon when going disk to disk.

Because the network worked at the block transaction level, it was not possible to have the same disk mounted write enabled on multiple systems simultaneously since the kernels did not cooperate. It was also unwise to mount a disk write protected which was mounted write enabled by another system, since the disk contents might change underneath the client system. Block cacheing was done by the client through the normal block cacheing mechanism. Due to the speed of the network (very fast for its day) and the simplicity of the implementation, the remote disks were virtually as fast as local ones, with some minor additional latency. Network packets looked a lot like buf structs but with a network source address. The destination was specified by the b_dev field!

The typical use of this system was to mount write protected disks on multiple systems so that only one copy of the information was required. Occasionally write enabled disks were mounted if it was known that the sections desired were not going to change. Sometimes people were surprised.

The MIT RVD

One of the earliest remote disk protocols was the "remote virtual disk" protocol done by Mike Greenwald and Larry Allen at the MIT Laboratory for Computer Science. As with the BRL remote disk facility, sharing of remote file systems was only possible if the disks were write protected, since there was no high level interprocessor cooperation. MIT used multiple VAX/750s running RVD as file farms for remote systems.

Sun's Network Disk

SUN Microsystems has been responsible for two different remote file system implementations. The first is called the Network Disk protocol (or ND for short) and is basically a simple protocol for providing access to remotely resident disk segments. The interface to the client is at the device driver level, where a standard block style device driver packages disk I/O operations into packets to be serviced on the server machine. Sun has run diskless workstations using this protocol for several years. Each system thinks of the remote disk segment as though it were local. There is no coordination between systems, so unless the segment is mounted write protected, on all systems, there is no joint access possible.

This protocol was designed to be simple to implement and debug and as fast as possible given that it is treated as though it were a local disk segment. The local disk buffer cache can help to reduce network accesses unless raw disk accesses are made. Disadvantages of this protocol were the low level of the interface which made simultaneous access unwise, and the inability to work on other than a single Ethernet since it was not built on top of an existing internet protocol capable of routing across multiple networks. Recently the University of Maryland has reverse engineered this facility and can provide it to those with Sun source licenses. Maryland is now using VAXen as file servers for Suns using this code.

The Version 8 Remote File System

Peter Weinberger has a remote file system for the research version of UNIX used by Dennis Ritchie and friends at Bell Labs. Peter has an interesting philosophy about how a remote file system should function which he explained at the Utah Usenix conference in 1984. To paraphrase him: "I want **my** system to be able to get at **your** files." The client is in the Version 8 kernel, which removed the need for any user mode changes. The server runs as a user mode daemon which services requests over the network link. One of his design criteria was that he could do as much as he wanted to his machine, but others would probably only let him run a user-mode program. Having a user mode daemon to give to other people to run makes it easier for him to get other people to participate as remote servers. This also makes debugging the server much easier. The server normally runs as root and handles authentication on a per-user basis, but can be run totally unprivileged with degraded abilities.

The Version 8 remote file system is done at the inode interface level. Where there had been calls to the inode level routines, there are now switch statements based on the file system type such as local, network, process, and faces. A generalized mount system call was added to the Version 8 kernel:

```
gmount(fd, path, id)
int   fd;
char  *path;      /* Typically "/n/host" */
int   id;   /* file system type */
```

Remote file systems are mounted by first opening a connection to the remote host and then issuing the gmount system call. The connection is periodically checked for connectivity and if the connection fails, an automatic umount occurs. The network is only assumed to give virtual circuits.

Every file accessible through the remote file system has essentially two names. First it has its native name on the remote host, like "/usr/lib/crontab". It also has a different name on the client system, e.g. "/n/remotehost/usr/lib/crontab". For convenience, it is possible to create a link so references to local files can be made by the name /n/localhost/*filename*. If this is done /n/host/filename becomes a valid reference for *filename* on all machines. Files that are read or written on a remote machine get the modification times of the remote machine. This can cause problems with some programs such as make. A throughput number of 16K bytes/per/second was given at the Summer 1984 Usenix conference in Utah, but this may be out of date.

NETIX File System

NETIX was an attempt at a distributed UNIX system, including a remote file system capability that was done at the Bell Telephone Manufacturing Company in Antwerp, Belgium around 1982. Transparent access to all files in the NETIX system was an important goal. The NETIX implementation was entirely within the kernel and at the inode level. The naming convention used by NETIX was the super-root. Remote files were accessed by using names such as /../hostname/remotefile. The UNIX system was expanded to include a "network inode" which would trigger remote request processing. These may have allowed the /../hostname naming to be bypassed, the paper was unclear on this.

On every host was kept a list of the other hosts participating in the NETIX system and a mapping from that hostname to a unique *machine id.* Each user had a *base machine* on which his login was established. User authentication was handled by generating a network user id from the concatenation of the machine id and the user's id on his base machine. Superusers did not have remote privileges unless explicitly granted by the remote machine.

The Masscomp Extended File System

In 1984, Masscomp implemented a remote file system for their high performance workstation. Their implementation has many similarities with the Version 8 remote file system done by Peter Weinberger. Their design criteria had two major components. They could not change the semantics or syntax of the UNIX system call interface, and they did not want users to have to recompile or relink their programs to operate under the new UNIX that supported remote file access. Portability of the implementation to other architectures was not a major design consideration. They chose to implement both the client and the server in the kernel. While there was no reason forcing the use of a kernel mode server, the performance gain from using a kernel mode server was felt to be sufficient

reason for the additional effort it involved. The implementation was done at the inode level and the notion of a remote inode was introduced. No changes to the UNIX file structure were required since the remote inode only existed in core. Consequently, the current file position was stored by the client and passed to the server with each read or write request.

Masscomp chose to implement an "rmount" system call to add remote file systems to the namespace of the local host. An "rumount" was also added. The network communication was based on a 4.2BSD style implementation of the Internet Protocol family as defined by the United States Department of Defense (DOD). A new high level protocol called the Reliable Datagram Protocol was implemented and added on top of the DOD IP layer for use by the EFS. (This is not the same as the forthcoming DOD RDP).

The server was implemented as a collection of lightweight kernel processes. This "process pool" had the ability to set up its user structure as necessary to fulfill each request. There was a queue of processes and a queue of requests. Multiple server processes were used so that operations could continue if one of the servers blocked on an I/O operation. A single "lifeguard" process managed the process pool, dispatching incoming requests to servers and handling error recovery.

The Newcastle Connection (aka UNIX United)

The Newcastle Connection was probably the first and certainly the most complete implementation of a remote file system in user mode. This system was developed at the University of Newcastle-upon-Tyne, and subsequently maintained and marketed by MARI.

The interface for the Newcastle Connection is just above the system call interface. The Newcastle Connection client is embodied in a special version of the C library which provides replacements for all the standard file system routines. If a file is local, the local system call is used. If the file is remote, then the system call parameters are sent to the remote host for execution, and the results are returned in a response message. The server is a standard user-mode daemon.

Until recently, the Newcastle code used its own low level networking protocols, but it has subsequently been implemented on top of more standard protocols such at UDP/IP.

There are several problems with this approach to implementation. The worst problem is that with much of the code to support remote file system operations in the C library, every program one desires to use across machines has to be recompiled to include the network client code, and subsequently relinked every time there is an update to the client code. This is probably the single greatest failing of the system. Since the code is in the C library, assembler programs, or programs written in languages which do not use the C library are also unusable across system boundaries. Information is passed between processes in a special environment variable, so processes which play with the environment may break the remote file system operation.

The naming convention chosen by Newcastle was the super-root. Remote files are accessed by referencing /../hostname/remotefile. There can in fact be a hierarchy of naming "above" the root, for example, /../../ee/vax1/remotefile from the host cs/vax2. The naming hierarchy is independent of the actual connections which may be present. Actual routing of packets is handled at a lower level.

Authentication in the Newcastle Connection is symbolic. No assumptions are made about globally unique userids or usernames. Each client is expected to verify userids before sending the information to the remote server. The system administration on the server system can exert as much or as little access control as they like to files on their system based on user/host pairings.

There is not a great deal of information available on the performance of the Newcastle connection, but Randall made the following observations in his talk at the April 1983 European UNIX Users Group (EUUG) conference:

- Local file descriptor syscalls: check for remote FD, negligible.
- Local pathname syscalls: 1 stat syscall (to determine file is local).
- Remote file descriptor syscalls: check for remote FD, network I/O
- Remote pathname syscalls: 1 stat syscall (to determine file is remote), some number of additional stats to determine the local portion of the pathname, network I/O.

The summary ignores the costs associated with setting up server processes, which are used quite generously by the Newcastle Connection. To the last case I would add that it is necessary for the remote server to essentially duplicate the code in the kernel routine namei() so that it can

properly determine when a path leads off the root and on to another machine, or back to the original. This essentially involves a stat per path component. This becomes particularly important when symbolic links are available.

Purdue IBIS, Phase 1

Over a year ago, Walter Tichy from Purdue University made available some user mode code to provide a remote file system capability on 4.2BSD UNIX systems in a manner analogous to the Newcastle Connection. The code was made available as the first step in a more ambitious project called IBIS. This user-mode remote file system came to be referred to by the larger project name, IBIS.

The client portion consisted of a library of replacement routines for all of the common file system related system calls and a special startup program. The real system call interfaces were renamed to an unlikely name (e.g. open would become l_open). To use the system, all that was normally required was to recompile all the programs you wanted to have remote file access with this new library. The special startup program was used to establish trusted connections with the remote hosts and ran as a privileged process. All connections were carried out on top of TCP/IP connections, giving reliable data transfer and sequencing. Remote files were named by preceding the name of file with "host:", for example "purdue-merlin:/etc/passwd" would access the passwd file on the host Purdue-Merlin. This notation for remote files eventually proved to be a significant problem. It simply did not fit well enough with the standard UNIX naming style. Programs which checked to see a path was absolute by looking at path[0] and comparing that to a slash were quite confused.

The server for this system was a standard user-mode daemon process under 4.2BSD. Authentication was carried out between the startup process and the daemon after initializing the connection. The daemon used the same authentication facilities used by the 4.2BSD programs rsh and rlogin. This meant that authentication was completely symbolic and controllable by the user through his .rhosts file.

There were numerous problems with this system, including many of the same ones found in the Newcastle Connection. Several items were unimplemented or incompletely done. It was not machine independent. There was no attempt to take care of byte ordering or word alignment in structures. Every client process had one or more remote server processes shadowing it. If a process forked, the daemons forked. Remote files were not maintained across execs, even if both processes were compiled with the remote file access library. Remote file names that did not specify an absolute path, were taken to be relative to the users home directory by default. Recompiling every program is a massive undertaking, and in some cases programs would not work remotely due to the problems already mentioned.

Harvard Remote File System

The Harvard University Science Center did a remote file system back around 1979 called RFS. Steve Dyer was one of the principals in this effort. Their RFS was installed in PDP-11 systems running Harvard's own version of V6 UNIX. Under the Harvard RFS, network access was completely transparent to any program or user. The Harvard system was implemented just below the system call level so effectively all operations on remote files caused the request to be sent to the remote host for completion.

The Harvard RFS is a generalization of the mount facility, but the sub-hierarchies are found on different processors, rather than on different disk packs. When a program operates upon a file which is remotely mounted, the kernel sends packets over the link requesting an operation be performed. The program is blocked pending a response. At the remote end, a "server" process reads the commands, performs the operation if it can, and writes back any data expected, along with a success/failure indication. A particular machine may be both server and requestor, in a symmetric relationship with another machine. However, a processor may be connected to any number of machines in either capacity.

Harvard RFS: Client

To attach to another processor's file system, the "r-mount" system call takes three arguments: a "machine number", a small integer which selects a particular communications line to another machine, the "local name", an existing file which will later refer to the other processor's files, and the "remote name", which is a pathname string that refers to a hierarchy on the remote machine.

Unlike the "mount" system call which incorporates the entire tree structure of a disk pack, "r-mount" may select a sub-portion of the remote machine's files by specifying the pathname of a directory other than "/" as the remote name. The "remote name" is mapped into a unique integer, the string number, and the server is notified of the correspondence between this small number and the longer remote path name. Only the string number is passed to the server in subsequent requests. "r-mount" constructs a device name from the machine number and the string number, and a flag identifying it as an RFS device. It inserts this into an available slot in the mounted file system table, along with the i-number of the local file.

When a program issues a system call to access the file system, the UNIX kernel calls a subroutine to translate the pathname of a file into a device name and i-number. If an error is detected as it descends the pathname, the routine returns with an code describing the problem (e.g., no such file, not a directory), and the system call passes this back to the user's program. When this procedure reaches a component of a pathname that is flagged as a remote mount in the mount table, it returns a special error code, EREMFS. The device name found in the mount table specifies the machine number and sub-hierarchy. It is the responsibility of every system call which searches a pathname to recognize the EREMFS condition, package the remainder of the path name and any other necessary data, and hand them to the subroutine responsible for sending requests across the link. If a system call does not handle the EREMFS condition, it is treated like any other error, generating an message of the form: "Unimplemented Remote File System Operation". Because of the existing behavior of the UNIX system calls, the RFS system could be debugged piece-by-piece, without exhaustively incorporating large amounts of untried code.

Harvard RFS: File I/O

Opening or creating a file and reading or writing from that file, were more difficult to implement under RFS. Unlike once-only operations such as deleting files, or changing ownership or protection codes, "open" and "create" return small integers called file descriptors, which are subsequently used by the "read" and "write" system calls. These integers are indices into per-process tables, which ultimately access copies of i-node structures describing the opened files. There are two problems: first, "read" and "write" no longer have access to a pathname; hence, they cannot obtain the error, EREMFS, and perform the appropriate operation. Second, because the files are located on a remote machine, it is difficult to imagine what data are stored in the i-node on the local machine. Both problems are solved by having "open" and "create" construct a local i-node which is a "special file". Special files on UNIX do not access any data on a file system. Rather, when opened, read, written, or closed, they invoke device-specific subroutines. Usually, this is a clever way to allow hardware devices such as magtapes or disks to be accessed through the normal pathname convention. Here, by providing a set of subroutines to handle reading, writing, and closing of RFS files, and representing the open file as a special device, the main system calls were not modified. These special i-nodes are unusual in that they are constructed as needed by "open" and "create" and never appear on the disk. Their subfields have been filled in with data needed by the subroutines to route requests to the appropriate machine.

Harvard RFS: Server Process

The Harvard RFS server is a privileged program running in user mode. It is not part of the UNIX kernel. It reads from a special file that directly accesses the communications line, performs the requested operation, and writes its reply back on the same file. The length of request packets varies, depending on the system call which sent it. In general, a packet has the following byte-string format:

HDR1 HDR2 CHAN# MSGBYTECOUNT OPCODE
REPLYCHAN# USERID.HIBYTE USERID.LOBYTE
GROUPID.HIBYTE GROUPID.LOBYTE DATA ...

The server changes its effective privileges to the user with the specified UID and GID. This ensures that the server executes the system call with the same permissions as the program on the local machine. It reassembles the remaining data from the packet, performs the system call specified in the OPCODE, packages any data returned and sends it back to the requestor. It then resets itself to a privileged state. HDR1 and HDR2 are bytes to indicate the start of a requestor packet; they are included to ensure synchronization between the two machines. REPLYCHAN# is an index into the processes that are blocked waiting for the server to respond. The server includes this byte in the

packets it writes back to the requestor, which demultiplexes its data stream, sending the data to the appropriate process.

When the server receives a request to open or create a file, it spawns a new instance of itself, called the "sub-server", using the UNIX primitive "fork". This sub-server opens the file and pauses indefinitely, expecting read or write requests to arrive later from the requestor. The main server uses the CHAN# byte to distinguish between once-only requests that it can satisfy and read/write requests directed to the sub-servers. CHAN# 0 indicates a packet which is directed to the main server; other channels are used as indices into the currently existing sub-servers. If the server begins to read a request with a non-zero CHAN#, it wakes up the particular sub-server indexed by the byte, and the sub-server reads the remainder of the packet, completing the operation. During the time the sub-server reads and writes from the communications link, the main server suspends itself. When the sub-server has completed a request, it signals the main server to resume reading. At any instant, only one server process has control of the link.

When a program on the requestor closes an RFS file, a packet is sent to its associated sub-server on the remote machine. The sub-server reads the request, closes the file, sends an affirmative acknowledgement that the file has been closed, and dies.

Because the server program runs in user mode as a separate process, it was very easy to debug. It reads a stream of bytes from its standard input and writes response packets onto its standard output. By writing small, interactive front and back ends for the program which generated the requestor packets and interpreted the response packets, a fully debugged server existed well before any modifications were made to the requestor side in the UNIX kernel.

Harvard RFS: Reflections on the Design

An RFS operation will never be serviced quite as quickly as a normal file system request. An ordinary read operation of a UNIX file causes data to be copied from the file into the user's program. At the same time, UNIX enqueues the next logical block of the file to be read into a system buffer. If the program requests this block, it should already have been read in from the disk. A write operation causes data to be copied to a system buffer which is enqueued to be written out later. The program does not wait for the actual I/O operation to be performed. These read-ahead/write-behind buffering algorithms greatly improve the throughput of the file system.

The Harvard RFS does not use this scheme because of the danger of keeping local copies of a remote machine's data, which could possibly be invalidated by another program running on the remote machine. All I/O operations on an RFS file are synchronous--when a program executes a system call, even a write operation, it must send the request packet and wait for a response. This is a significant penalty. Because open RFS files are represented as "special files", not reflecting the actual state of the file, errors cannot be forseen on the local machine before a request is made.

A single accounting system is maintained between the two machines. This ensures that UID and GID numbers are not duplicated between the two machines. If a user on one machine has the same user-id as a different user on another machine, he IS that user as far as RFS is concerned. Therefore, unless some mechanism is implemented to insure that user-ids are not duplicated, UNIX protection mechanisms are worthless across RFS. For this reason, a system like RFS is not optimal for communication between unrelated sites.

On the HRSTS systems there are 8 separate file systems accessible from either machine. These are named "/rfs", "/fs/a", "/fs/b", through "/fs/g". The file system named "/rfs" is, by convention, the other 11/70's root file system. It is remotely mounted. On one of the machines, the file systems "/fs/a" through "/fs/d" are physically mounted on the machine; the remainder, "/fs/e" through "/fs/g" are remotely mounted. On the other, it is the converse: "/fs/e", "/fs/f", and "/fs/g" are physically present, "/fs/a" through "/fs/d" are remotely mounted. However, programs on both machines see the same directory structure. This makes the transition to a single machine trivial in the case of a failure, assuming that all disk drives are dual-ported. All file systems could then be physically mounted under the same names while the other machine was being repaired.

COCANET

Around 1981, the Department of Electrical Engineering and Computer Science at UC Berkeley developed a network system called COCANET. This system was in a sense somewhat more than just a remote file system since it also has facilities for remote execution, but the basic facility was remote file access and manipulation. The system took 18 months to implement. The time was spent equally between kernel modifications, the network manager process, and the server processes. The main change to the UNIX operating system was to add a message oriented IPC facility for network applications such as the remote file system. The other major change was to modify namei() and other kernel routines to recognize references to remote files. If a system call failed because of a reference to a network special file, the kernel would pass the system call to the network manager process for interpretation by the remote machine. The network manager was a special memory resident process analogous to the swapper that passed requests to the network and processed incoming responses.

Two types of servers were employed in COCANET. The shared server process handled stateless requests such as stat, chmod, and unlink. Whenever a file was opened on the remote machine, the network manager and the remote server would cooperate to create a private server that would shadow the original process. File descriptor operations were subsequently handled by the private server for that process. Whenever the original process forked, the server would fork twice. The one server would become a shared server, maintaining files open to both processes. In addition, each process also has a private server to maintain process specific data such as current working directory.

Execs of remote files caused the program to be run on the remote host by a private server. The local process would exec a ghost server in place of the originally requested program. The ghost would then handle I/O requests back to home machine for the now remote process. Most often the ghost would be needed to handle STDIN, STDOUT, and STDERR back to the original login terminal.

User authentication in COCANET was handled by static and dynamic mapping tables. These tables contained mappings from host/userid to local-userid. Remote accesses as root were not permitted.

The naming strategy used by COCANET was to have the root of each remote machine show up as a subdirectory of the local root. For example, the root of host2 would be referenced from host1 as /host2, and the passwd file on host2 would be /host2/etc/passwd when referenced from host1.

Lucasfilm Extend File System

The Lucasfilm Ltd. (LFL) remote file system, called EFS (Extended File System), uses kernel hooks at the system call level. A character structured special file is used to trigger special handling by namei and the system call code. All client support is in the kernel, all server code is embodied in user-level client server processes.

Authentication is the same as for rcmd (/etc/hosts.equiv and ~/.rhosts). There is no uid-gid mapping scheme, all machines must share common passwd and group files.

Naming is wired down to be /net/hostname because of code in the kernel for handling "cd ..". This could be fairly easily fixed by doing pathname translation a component at a time.

Aside from the normal system calls, chdir and execv are supported. In addition symbolic links almost work correctly. There is one problem which makes it impossible to cd across the net then reference a symbolic link which comes back to the client machine (a single server buffer which is re-used).

The system has been used mostly with 4.1BSD - 4.2BSD machines, though there is a UDP based version for talking to V7 machines written back when LFL had it's own 68K port of a V7 kernel. Byte order is not important, LFL has VAXen, SUN's, CCI's, and so on, all tied together. In addition, directory structure differences are handled -- this is a big problem with VAX-SUN-CCI combinations.

Information on performance is not readily available. The server has a 12.5KByte buffer which limits read-write calls to send-receive's of at most this size. All data is sent across TCP connections so performance is pretty much limited by TCP and hardware performance except for the V7 variant which isn't used any more. One guess is that a write to /dev/null across the network runs at no more than 60-80KBytes per second. EFS is commonly used to transfer very large files (8-12MB

pictures) and the LFL users are reasonably satisfied, though it could easily be sped up some if someone was interested in moving the server into the kernel.

Some work was done about a year ago on improving performance by cacheing information in the server and re-using server processes. In addition, the kernel client code caches connections and re-uses a server when possible. In general latency is very good. The big problems are the overall design (where it sits in the kernel and how it is integrated), the fact that cd is really glued onto the side (a server process is used for each cd), and the fact that the server process is at the user level (degrading performance).

EFS has been around a long time. Early versions existed before 4.1a and it's been in production use with TCP since 4.1a. This predates most people's systems, including Peter Weinberger's Version 8 stuff, though it was done after most of the "remote disk" interfaces.

Sun's Network File System

The second remote file system Sun Microsystems has designed is the Network File System, or NFS for short. The NFS is a much more ambitious project than the ND protocol. NFS was designed from the start to give simultaneous access across multiple machine architectures. NFS operates on top of existing protocols and uses a machine independent data format called XDR (eXternal Data Representation).

To understand where NFS interfaces to UNIX, it is useful to understand some changes Sun made to the UNIX file system code. Sun generalized the notion of a file system and built a canonical file system interface in UNIX. A new data structure called a vnode was introduced between the file structure and the file system specific inode structure. A set of operations was defined for vnodes, and implemented for each file system type. Each mounted file system has not only a mount point, but also a type which is used to select which type of file system operations to use on that file system. As of January 1985, there were three different file system types supported. The first was locally mounted 4.2BSD style file systems. The second type was locally mounted MS-DOS style file systems. The third type was a remote file system using the NFS interface.

Operations on the 4.2BSD file systems have a fairly one-to-one mapping with the operations in the vnode interface. Operations on the MS-DOS file systems are more restrictive or no-ops due to the comparitive simplicity of the MS-DOS file system. Operations on the Network File System map quite closely to the operations on vnodes except they are encapsulated in messages to the remote host on which the remote file system actually resides.

The Sun NFS uses a stateless network protocol. This has both advantages and disadvantages. The biggest advantage of stateless protocols is much easier error recovery from server failures. One-time operations on file names, e.g. stat() and rename(), are straightforward to implement and are effectively atomic. Conversely, operations like open, creat, flock naturally cause state to be carried somewhere. Sun chose to keep this state in the client an not to propogate it to the server.

In the Sun NFS, files are not opened on the server in the sense of section 2 of the manual. Opening a file simply results in a unique "handle" being generated that can be used to reference that file in the future without passing the name of the file again. The client records this handle in a remote inode structure. For a UNIX file system, this basically consists of the device and inode number, along with a reference to the system. Subsequent operations like read and write simply send along the device/inode handle with the request. The server can use this to identify the file that is to be read/written. Since read/write requests also include the offset, the offset need not be maintained on the remote host as this would be a form of state information. Sun also designed their RFS protocol so that if the same operation was made more than once, the second and subsequent actions would be no-ops. This means they do not have to carry any network connection state either. If a packet was received more than once as the result of retransmissions or network error, no damage would result from acting on it twice regardless of prior actions.

The problem is that several file system operations effectively force you to keep state. There are two notable examples that cause problems for the Sun Network File System. The first is file locking. Applying a lock is an operation that is only effective if applied at the server. If the server fails, the lock is lost, so file locking is not supported on the basic NFS system. They may provide locking later through a separate network locking service. The second problem for NFS is that the semantics of removing an open file cannot be properly supported. Since the server is stateless, it does not know that someone has the file open, so when you issue an unlink() call on the file, it is

really removed. Subsequent references to that device/inode combination will fail. To get around this problem, the Sun NFS client tries to move the file to a hidden name and issues a remove call on the file after final close is called on the file. If the client crashes before final close, NFS leaves unremoved funny filenames in the directories of the server. Presumably there is some daemon to clean these up occasionally.

As a result of limitations on the Sun NFS, it is not a truly transparent remote file system. This is a major problem if it is to be used with existing programs. It is likely that existing programs that used facilities such as flock, will never run under NFS without modification or relinking.

Sun Microsystems has spoken a great deal about trying to make NFS an industry standard by publishing the XDR and remote procedure call facilities, and calling on others to adopt the NFS system. Unfortunately Sun has failed to make the one move that would more than any other make NFS an acceptable standard: make their 4.2BSD implementation available for public distribution free of proprietary restrictions. Even IBM did this when they wanted DES accepted. The lack of true public accessibility to NFS combined with its other shortcomings does not bode well for Sun's attempt at standardization.

LOCUS

LOCUS started as a DARPA research project at UCLA CS Department. The LOCUS development has now been spun off from the UCLA and is primarily in the hands of an independent company (Locus Computing Corporation, or LCC), headed by Dr. Gerald Popek and staffed by many of his former graduate students.

LOCUS is what you might call a "tightly coupled" network. That is, the computers in a LOCUS network continuously keep track of one another and attempt to act to the world as a single, unified resource. One advantage of a tightly coupled network is that authentication issues are pretty much irrelevant. It is not necessary to go to heroic measures to determine remote access permissions, since the entire collection of computers is a single entity, and every site can safely "trust" every other site to do the right things as far as access control is concerned. The possibility of an intruder eavesdropping or forging traffic on the LAN has not been addressed to date in LOCUS and should be handled at a lower network level.

Although a file system in LOCUS may be thought of as physically "mounted" on the site that is actually connected to the disk, the data is actually accessible from any site on the network. LOCUS has a concept of a "global mount table", so that every site on the net knows about every file system mounted everywhere on the net. Every file system has a "global file system number", a "GFS number" for short, by which it is known internally. Thus, for instance, a file can be identified uniquely by the ordered pair (GFS#, inode#).

LOCUS file names do not, in general, tell you which site the file system is physically mounted on. This information is available through the global mount table, but it is not really important to users in general from the standpoint of ordinary file access. LOCUS differs in this respect from systems where the site name is somehow embedded in the path name. Since a file name does not include the physical location of the file, it is possible (as an occasional maintenance operation) to move a file system from one site to another, just as you might move a file system from one part of a disk to another on a single site, and allow the users to access the files at the new location via the same names as before, as if nothing had happened.

All the remote file access stuff is in the kernel. Hence, programs do not have to be recompiled or modified in any way in order to access remotely stored files.

LOCUS has experimented with various kinds of replication strategies for file systems (i.e., storing multiple copies of all or part of a file system, to guard against failure either of a single site or of the network communication links). Unfortunately, the massive problems related to reconciling conflicting versions of a file on two different sites (e.g., when reuniting the net after a partition) caused LOCUS to abandon for the time being most attempts at replication.

The one kind of replicated file system which LOCUS does support uses what they call a "primary-site" replication strategy. That is, the file system is stored on several (typically all) sites, and a process which only wants to read a file can use any copy of the file (typically the copy stored on the site where the process is running). But all modifications to the file must go only through a designated "primary site" for the file system. The "primary site" coordinates the distribution of updated copies of the file to the other sites. If the primary site for a file system happens to be down

at a given time, files in that file system can be read (via copies on other sites), but the file system is in effect marked as read-only until the primary site returns. UCLA has set up the root file system as a replicated file system in this manner since read-only accesses to the root are extremely common (e.g., binaries and the passwd file), and since each site must have its own root so that it can run as an autonomous site when necessary.

Now, there are some cases where full location transparency in a file system is impossible or undesirable. For example, it would be highly inefficient if there were only one "/tmp" for the entire net, since this would require all "/tmp" activity to go through a single site, and if that site were down, all kinds of programs would curdle. Also, some files (such as "/etc/utmp" and the things in "/usr/adm") cannot conveniently be maintained in a global fashion; rather, each site really wants to have its very own separate "/etc/utmp".

The way LOCUS handles this problem is via a special directory name (currently called "/local"), together with a new parameter in a process's user structure which defines a directory path to be substituted in place of "/local" whenever it is encountered. Also, they use symbolic links to map such directories as "/tmp" to "/local/tmp", or "/usr/spool" to "/local/spool", or to map individual files to site-specific places (e.g., "/etc/utmp" is a symbolic link to "/local/utmp").

Each site at UCLA has a file system named after that site and stored on that site. Two of their sites are named Jason and Medea, with "local" file systems named "/jason" and "/medea", respectively, as well as file systems named "/jason/tmp" and "/medea/tmp", and so on. This means that a reference to "/tmp", in a process running on the site Jason, will resolve via the symbolic link to "/local/tmp", which in turn will be transformed to "/jason/tmp". On the other hand, a reference to "/tmp" in a process running on Medea will end up referring to "/medea/tmp". A process could explicitly utter "/jason/tmp" or "/medea/tmp" if it wanted to, of course, but this is not generally necessary.

The system is being modified slightly so that the magic string at the beginning of a file name will be "<LOCAL>" instead of "/local". This substitution will eventually be valid only when the magic string occurs in a symbolic link. This is being done because it has turned out to be somewhat of a confusion and a problem for "/local" to be valid in the context of arbitrary commands.

LOCUS has facilities for process migration (moving an existing process from one site to another) where the two sites are of identical CPU types, as well as for initiating a new process on a different site from the parent. Depending on what one is doing, the information in the user structure which tells the system how to map references to "/local" can either remain the same when site boundaries are crossed, or it can be changed to refer to the environment of the new site.

The various sites in a LOCUS network communicate via message packets sent over a LAN. For efficiency's sake, they do not use any kind of layering in the LOCUS kernel-to-kernel protocol. As an experiment, one graduate student last year linked two LOCUS sites over the ARPANET using IP packets. Message packets are of two basic kinds: short packets for control messages and system calls which do not require passing of data from a file, and long packets which contain everything in a short packet, plus a block of file data. All message packets are individually acknowledged via a special packet type and retransmitted as necessary. LCC is working on batch acknowledgment of several packets at once in order to improve throughput and efficiency.

Unfortunately, LOCUS is implemented in the 4.1BSD kernel and is quite cumbersome, requiring modifications here and there throughout the kernel. If UNIX were a message-passing kernel, it would probably have been much easier to transform it into LOCUS, since the machine-to-machine message packets could simply have been shunted into the already existing message queueing and processing routines.

LOCUS does not currently support demand-paging of binaries across the net. Hence, a demand-paged binary (magic number 413) residing on a file system physically mounted on another site is treated as if it were a regular read-only-text binary (magic number 410). The LOCUS people at LCC may be working on improvements to this feature. In any case, the efficiency of system binaries is not affected since the root is replicated on all sites. A private user's applications will run somewhat less efficiently if he were running on a remote site but accessing files on the local system.

UCLA is currently working on forming a heterogeneous network (adding an IBM 4381 to their VAX farm). This will involve a scheme of "hidden" directories, to allow multiple load modules for different CPU types to be accessed via the same name. It will also involve a modification to the "primary-site" replicated file system to allow selected files in a file system to be

stored on some sites but not others. The primary site will have to store everything, but the other sites will only have to store the binaries of their own type -- VAX binaries on VAXen, IBM binaries on 4381's, and so on.

Part 3: The CWI Remote File System

Remote File System Plans at CWI

Given the variety of remote file system implementations already in existence, you might wonder why someone would choose to implement yet another remote file system. There are several reasons. First, none of the "good" remote file systems are in the public domain, or even readily available. Second, most of the systems rely on remote systems having the same login name to UID/GID assignments as the client. While this simplifies authentication, it is often impossible to arrange in a network of hosts managed by different organizations. None of the existing systems have been designed to be used as commonly as current network services such as mail, file transfer, or remote login. Most require the use of an explicit remote mount system call to allow access to the remote system. Generally this is a privileged request. The expectation is that remote file system service will eventually replace in some way the explicit file transfer services of today. As such, one might expect to see a hundred hosts to be accessed in a period of several days from a single host on a net such as the DARPA Internet which has hundreds (and soon thousands) of hosts. The explicit use of mount for every remote filesystem is somewhat mindboggling in such an environment.

As the result of the shortcomings mentioned above, we have decided to implement a remote file system for 4.2/4.3 BSD. While not limited to this operating system, it makes a good starting point. The system is implemented at the system call level. System calls on remote files result in the generation of a remote procedure call (RPC) to the remote host to accomplish the request. The client is being implemented entirely within the kernel. As a result most programs will run without changes of any kind. At the same time we will also be implementing a canonical file system interface much like the UNIX Version 8 system so we can experiment with alternative file systems for fonts and bitmaps. Initially our server will run in user mode, but will probably be implemented in the kernel as an option if it significantly improves performance.

The work on this implementation was started over a year ago when I was still at the Ballistics Research Laboratory of the U.S. Army. Dan Tso from Rockefeller University and I started experimenting with the IBIS code from Purdue. We concluded that a different solution was called for. Since then we have worked intermittently on its development in a user mode testbed and on a full kernel mode client. I joined CWI in July of this year in a temporary one year appointment with the remote file system as my major assignment. As a result development has moved from BRL to CWI.

There are several goals to our project. We want all authentication to be done symbolically, based on hostname/username pairs. The system should be relatively independent of the underlying network topology and protocols. Connections to remote hosts should happen automatically without the user having to explicitly issue a remote mount request to the system. Performance is important as well. We hope the limiting factor for data transfers will be the TCP/IP networking code being used in the initial implementation. The system is designed to be independent of the actual networking protocols used, but will require either a reliable stream or a reliable datagram facility from the network. The upcoming DOD Reliable Datagram Protocol should be a natural choice for use by the remote file system.

The remote procedure call protocol will have two modes, generic and native. In generic mode, all RFS data will be transferred in a canonical format, easily handled by hosts with different word sizes, byte ordering, and operating systems. The native mode will be used by hosts that find they have the same byte ordering and data formats, and hence can avoid the conversion to canonical network format. In native mode, things like stat structures are transferred without alteration from one machine to the other. This saves considerably on CPU intensive data conversion.

A new descriptor type has been introduced called DTYPE_RINODE. This complements the existing types, _INODE and _SOCKET. A RINODE is created whenever a remote file is opened. The RINODE contains a pointer to information necessary to contact the remote server (an rfs connection structure) and a unique identifier for the remote inode. This notation may change when the canonical filesystem interface becomes better defined.

Client Operation

The client side of the RFS is activated when a *portal directory* is named by a special rfs system call. Eventually this will be accomplished with by a generalized mount system call. This system call causes the directory to be treated specially by namei(). If namei() finds that it is searching this *portal directory*, typically /n, then it takes the next element in the path as a hostname and the rest of the path as a path from the root of the remote host. If a connection does not exist to that host already, one is initiated by namei() on the users behalf (and by the users process). If a connection is made, an RPC occurs to satisfy the file system request, such as an open or a stat. The contents of the "/n" directory are actually a figment of the kernel's imagination, and include ., .., and an entry for each remote host which has an active connection. This allows the pwd command to operate as expected. The namei routine returns to the caller if a remote reference is encountered, and the caller then passes the request to the remote system using information left in the user structure. It is possible that the remote system will determine that a path has led back to the client system (chdir .. off the root of the remote system). In this case, the client loops back and calls namei again with data left from the remotenamei() call.

There are at least two operations that the client code must perform that are best done in user mode. Since our implementation is in kernel mode, we have to provide a user mode daemon that services requests from the kernel for information. The interface is in the form of a privileged system call which the daemon calls with results to queries that are returned by the system call. In a sense it is a reverse system call. The two queries supported at this time map UIDs to usernames, and hostnames into network address structures.

A new system call will be added to the system called readndir(), whose semantics will be similar to the readdir subroutine, but will read a large chunk of the directory at once. A readndir system call is necessary for two reasons. First, the readdir subroutine still "knows" the format of directories and uses the regular read system call to get the contents. Remote manipulation of the directory will be easier if directory entries are treated as atomic entities. This will also make the directory format conversion easier, and more explicit, rather than burying it as a special case in the read system call.

An additional field will be added to the stat structure giving a unique host identification. This number will be assigned at runtime and will be monotonically increasing. Its purpose is to create a unique triplet which will unambiguously identify a file to the system or user. Without this additional information, two stat calls on files of two different systems could yield identical device and inode indications. A number of programs rely on being able to identify files in this manner.

Server Operation

The servers are started up by the daemon whenever a new host attempts a connection to the server's host, one server per host. All of the requests between a given client and server are carried on one network connection, which is assumed to deliver reliable datagrams or a reliable stream. Multiplexing is done based on client process ID causing the RPC. Once a client and server have authenticated each other as being valid host representatives, the client introduces each new user as they request access to files on the remote host. Users are then accepted or rejected based on their ability to access the remote host. The current authentication scheme is based on the Berkeley R* protocol authentication mechanisms (/etc/hosts.equiv and .rhosts). It is trivial to add additional or different schemes by modifying the server appropriately.

Most operations can be handled with a simple request/response model with one small packet in each direction. There are a few cases where this is not possible. In particular, reads and writes can often be much larger than the maximum packet size. When a read or write exceeds the maximum packet size, the data is sent as a sequence of packets which form a single logical packet. The client is expected to keep track of the current offset into a file so that the server need not preserve file offsets. Each read or write request carries the current offset in the file. Atomic append mode still has the expected effect. This will make eventual server recovery procedures much simpler. Each communications channel will in fact carry many simultaneous request/response transactions, but only one per process under normal synchronous operations.

Areas still requiring attention are the implementation of select on remote files, and asynchronous I/O on remote special files. While simple operations should succeed without effort, the ramifications of blocking on disconnected tty ports could cause problems by consuming server resources. Use of select on the server side will greatly ease this problem when reading special files.

Acknowledgement

The author would like to extend special thanks to Steve Dyer, Sam Leffler, and Rich Wales for providing their comments and descriptions, without which this paper would have been notably incomplete. For those interested in learning more detail about the issues related to remote file system implementation, I would highly recommend the paper by Cole et al. from Masscomp, and the paper by Sandberg et al. from Sun Microsystems. Each treats a number of the important issues in detail, albeit with their own bias. Finally, I would like to thank Dan Tso of Rockefeller University for his inspirations and his willingness to test and fix what I had concocted.

Trademarks

Without lawyers, the following would not have been necessary.

CCI is a trademark of Computer Consoles, Incorporated.

DEC is a trademark of Digital Equipment Corporation.

IBM is a trademark of International Business Machines Corporation.

LCC is a trademark of Locus Computing Corporation.

LFL is probably a trademark of Lucasfilm, Limited.

LOCUS is a trademark of Locus Computing Corporation.

MARI is probably the trademark of someone in Newcastle.

MASSCOMP is a trademark of Massachusetts Computer Corporation.

NETIX is a trademark of ITT Bell Telephone Manufacturing Company.

Newcastle Connection is probably a trademark of MARI.

PDP-11 is a trademark of Digital Equipment Corporation.

Sun Microsystems is a trademark of Sun Microsystems, Incorporated.

UNIX is a trademark of AT&T Bell Laboratories.

VAX and VAX/750 are trademarks of Digital Equipment Corporation.

References

Cole, C. T., et al., Masscomp Corp., **An Implementation of an Extended File System for UNIX**, Summer 1985 Usenix Conference Proceedings, The Usenix Association, June 1985.

Cross, H., Rural Software, **A Bit About Eighth Edition**, EUUG Newsletter, Volume 5, Number 1, Spring 1985.

Dyer, S. P., Harvard University Science Center, Cambridge, MA., USA. **A Remote File System for UNIX**, ca. 1980.

Hac, A., Johns Hopkins University, **Distributed File Systems — A Survey**, Operating Systems Review, Volume 19, Number 1, ACM SIGOPS, January 1985.

Leffler, S., Lucasfilm Ltd., personal communications, August 1985.

Lyon, B., et al., Sun Microsystems, Inc., **Overview of the Sun Network File System**, Winter 1985 Usenix Conference Proceedings, The Usenix Association, January 1985 (and with minor changes in the Sun NFS documentation package).

McKie, J., Centrum voor Wiskunde en Informatic, Amsterdam, NL, **A Short Report on the NLUUG Conference, 10/12/82**, January 1983 (Version 8).

McKie, J., Centrum voor Wiskunde en Informatic, Amsterdam, NL, personal communications, August 1985.

Randall, B., et al., **The Newcastle Connection**, Software Practice and Experience, December 1982.

Randall, B., et al., **The Newcastle Connection**, notes from the EUUG conference, Spring 1983.

Rowe, L. A., and Birman, K. P., **A Local Network Based on the UNIX Operating System**, IEEE Transactions on Software Engineering, Volume SE-8, Number 2, March 1982 (Cocanet).

Sandberg R., et al., Sun Microsystems, Inc., **Design and Implementation of the Sun Network Filesystem**, Summer 1985 Usenix Conference Proceedings, The Usenix Association, June 1985.

Wales, R., Locus Project, UCLA, personal communications, August 1985.

Wambecq, A., Bell Telephone Manufacturing Co., **Netix: A Distributed Operating System Based on UNIX Software and Local Networking**, Summer 1983 Usenix Conference Proceedings, June 1983.

4.3BSD Overview

Kevin J. Dunlap
Digital Equipment Corporation
Computer Systems Research Group
University of California
Berkeley CA 94720

The definitive paper on 4.3BSD is "Performance Improvements and Functional Enhancements in 4.3BSD" by M. Kirk McKusick, Mike Karels and Sam Leffler, presented at the Summer 1985 USENIX in Portland, Oregon, USA.

4.2BSD provided new functionality, but due to the lack of time was not tuned to the level the developers would have liked. 4.3BSD is the next release of Berkeley's Unix offering. This release includes the system tuning that time restraints prohibited on the previous release as well as additional functionality. Performance improvements were provided by use of cacheing, optimization of existing algorithms, selection of more efficient search algorithms, and utilizing the more efficient facilities provided by 4.2BSD. Some of the newly added functionality were expansion of the network capabilities to handle subnets and gateways, support for windows and system logging. Also added to the release was the extension of the libraries and utilities to handle the new Internet name server, new system management tools, and Pascal support for *dbx*.

1. Introduction

4.3BSD is a concentration on performance improvements to the Berkeley operating system and its complement of utilities. This release provides some new functionality, but not in the sense that 4.2BSD did. 4.2BSD supplied major new functionality such as the inclusion of the networking software. This new release from Berkeley provides a tuned system.

The first section describes the performance improvements made in the kernel, libraries and utilities. The second section describes the new functionality of the system.

1.1. Performance Improvements

1.1.1. Kernel Optimizations

The optimization of the system was done for a general timesharing environment. As new machines are costly, most 4.2BSD sites have resorted to increasing the machines memory as memory costs have declined. Taking this into account, optimizations to the kernel were done at the cost of memory usage. The following sections list the changes made the the kernel.

1.1.1.1. Name Cacheing

The major improvement to name cacheing was to use a data structure that stored names with pointers to the inode* table. This data structure is independent of the inode table. By using this table of names, the cache gives an accurate representation of the most recently accessed names. The table's independence from the inode

*Inode is an abbreviation for "Index node". Each file on the system is described by an inode; the inode maintains access permission, and an array of pointers to the disk blocks that hold the data associated with the file.

table allows the size of the cache to be flexible.

The improvement to the name cacheing resulted in a cache hit rate of more than 70%. It did not however improve the performance of programs that sequentially scan a directory. A second performance improvement was made to name cacheing to alleviate this behavior. The system now tracks the directory offset of the last component of the most recently translated path name for each process. If the next name the process requests is in the same directory, the search is started from the point that the previous name was found.

During normal working hours, a timesharing system may be expected to do 500,000 to 1,000,000 name translations. The name cache for timesharing systems at Berkeley have a hit rate of 70% to 80%; with the directory offset cache getting a hit rate of 5% to 15%. Together the two cacheings will give almost an 85% hit rate. These two changes to name cacheing have reduced the system time spent on name translation from 25% to 10% [McKusick85].

1.1.1.2. Intelligent Auto Siloing

The standard VAX† terminal input hardware runs in two modes. They can either generate interrupts for every character typed or collect characters into a silo that is checked and drained periodically by the system. For the system to give quick response for interactive input and flow control, the silo must be checked 30 to 50 times per second. Normal users of Ascii terminals type at a rate less then 30 characters per second, for this use it is more efficient to interrupt for every character typed. When input is being generated by a another machine the input rate is more than 50 character, for this use it is more efficient to use the device's silo input mode. Since most systems use their dial up ports for both users logins and uucp logins, we can not set these ports to a static mode. Thus, the system monitors the input rate and selects the mode used based on the rate character are being received on the port.

1.1.1.3. Process Table Management

Systems have grown larger and with them the process table has grown past 200 entries. With tables this large, linear searches consume large amounts of CPU time and should be eliminated from frequently used facilities. The kernel process table is now multi-threaded to allow selective searching of active, zombie and unused process slots. Free slots can be found in a constant time by taking one from the front of the free list. The number of process slots used by a user can be found by scanning only the active list. In the 4.2BSD release, the kernel maintained link lists of descendents of each process. This list is now being used when dealing with process exit status; parent processes seeking the status of their children now avoid linear searches of the process table and examine only their direct descendents. The algorithm for finding all descendents of an exiting process has been changed to follow the links between child process and siblings, instead of performing multiple linear searches.

A unique process identifier is assigned whenever a process is forked. When creating a new process, the system previously scanned the entire process table looking for a unique process identifier. Now, the process table is scanned once looking for a range of unique identifiers and does not do another scan until that range is exhausted.

1.1.1.4. Scheduling

The scheduler previously scanned the entire process table once per second to recompute the process priorities. A process that ran for its entire time slice had its priority lowered, and a process that had been sleeping or used up its time slice had its priority raised. On a systems running many processes, the scheduler represented nearly 20% of the system time. This overhead has been reduced by changing the

†VAX, MASSBUS, UNIBUS, and DEC are trademarks of Digital Equipment Corporation.

scheduler to only consider runnable processes when recomputing priorities. To insure the sleeping processes get their boost, their priority is recomputed when they are placed back on the run queue. The list of runnable processes is a fraction of all the processes on the system, thus the cost of invoking the scheduler has dropped proportionally.

1.1.1.5. Clock Handling

The hardware clock interrupts the processor 100 times per second. Since most of the clock-based events do not need to be done at a high priority, the system schedules a lower priority software interrupt to do less time-critical events. Often there are no such events, and the software interrupt handler finds nothing to do. The high priority event now checks to see if there are any low priority events, before it schedules the software interrupt. Rather than posting a software interrupt that would occur as soon as it returns, the hardware clock interrupt handler simply lowers the processor priority and calls the software clock routines directly. These two optimizations have eliminated nearly 80 of the 100 interrupts per second.

1.1.1.6. File System

The file system typically uses a large block size of 4096 or 8192. To store small files efficiently, these large blocks are broken into smaller fragments, usually in multiples of 1024 bytes. The 4.2BSD file system uses a best fit strategy to fragment these blocks. This minimizes the number of full size blocks that have to be broken up. As the file grows, fragmented blocks are copied to larger and larger fragments until it fills a full block. With the new method, the first time the file system is forced to copy a growing fragment it places it at the beginning of a full size block. This will accommodate growth without continually copying the fragment. Thus allowing the file to fill the rest of the block. If the file stops growing, the rest of the block can be used for holding other fragments.

As files are created in a directory, the directory structure grows based on the number of files in the directory. Each time a new file is created this structure must be scanned to see that the file name is unique. Previously as the files in the directory were removed the corresponding blocks within the directory structure were not removed. In this situation file creations are expensive, because the system spends time scanning a relative empty structure. In 4.3BSD this problem has been corrected; when the system scans the directory structure to create a new file, it removes the empty blocks it finds. The next time it has to do a scan of the structure, it does not have to scan these empty blocks.

1.1.1.7. Network

The buffer space allocated for streams sockets and pipes has been increased to 4096 bytes. Stream sockets and pipes now return their buffer size in blocks in the stat structure. This allows the standard I/O library to use optimal buffering. To increase compatibility with other pipe implementations, stream sockets return a dummy device and inode number in the stat structure. The TCP maximum segment size is calculated according to the destination and interface in use; non-local connections use a more conservative size for long-haul networks.

1.1.1.8. Exec

When *exec*-ing a new process, the kernel creates the new program's argument list by copying the arguments and environment from the parent process's address space into the system, then copying it to the stack of the newly created process. The two copies were done one byte at a time. The copies are now done a string at a time. This reduced the time to process an argument list by a factor of ten; the average time to do an exec call decreased by 25% [McKusick85].

1.1.1.9. Context Switching

When the kernel posted a software event to force a process to be rescheduled, the process could be rescheduled for other reasons and would delay the event trap. At a later time the process would be selected to run, the pending system call would complete, and the event would take place. This causes the scheduler to be called for a second time to schedule the process yet again. This has been fixed by canceling any software reschedule events when saving the process context. This change doubles the speed with which processes can synchronize using pipes or signals.

1.1.1.10. Setjmp/Longjmp

When the kernel routine *setjmp* saves its current system context it would save more registers than necessary. By trimming the save to the minimum set of register needed, the overhead of the system call decreased by an average of 13% [McKusick85].

1.1.1.11. Compensating for Lack of Compiler Technology

The current C compilers available do not do any significant optimization. The C language is not well suited for optimization because of its liberal use of unbounded pointers. In the past optimization was done by having *sed* scripts run over assembly language and replace calls to small routines with the code for the body of the routine. This would eliminate the cost of the subroutine call and return, it did not eliminate the pushing and poping of several arguments to the routine. The *sed* script has been replaced by a more intelligent expander, *inline*, that merges the pushes and pops into moves to registers.

1.1.2. Improvements to Libraries and Utilities

It would seem that optimizations to the kernel would have the greatest payoff since they affect all the programs that run on the system. The kernel has been tuned many times before, so there are few areas for significant improvements. However, many of the libraries and utilities have never been tuned. There were many programs that spent 90% of their run time doing single character I/O. Changing these programs to use the standard I/O library cut their run time by a factor of five! Described in the following sections are other such non-kernel improvements made to the system.

1.1.2.1. Hashed Databases

There is a standard set of database management routines called *dbm*, that can be used to speed up lookups in large data files. These routines have been rewritten to use multiple files and are being used for lookups in the password and host files. This has significantly improved the running times of programs that use these files, such as *mail* subsystem, *ls* and the *C-sh* doing tilde expansion.

1.1.2.2. Buffered I/O

Buffered I/O is an optimal way of doing I/O, it is a method of doing reads and writes with out performing multiple function calls for each I/O of a character. Single character output degrades the performance of programs and causes congestion on a network. The standard error file (**stderr**) is now using buffered I/O that is found in the standard I/O library. Several important utilities did not use the standard I/O libraries and make use of its optimal I/O routines. These programs include the editor, the assembler, loader, C compiler and many other commonly used programs. They were fixed.

1.1.2.3. Mail System

The file locking primitives for mail previously used *link* and *unlink*. These routines modify the contents of directories, hence requires synchronous disk operations and cannot take advantage of the system name cache. The mail locking primitives

have been changed to use the 4.2BSD advisory locking facility. The mail system has also benefited from extensive profiling and tuning of *sendmail.*

1.1.2.4. The C Run-time Library

The memory allocation routines have been tuned to make better use of memory for allocations with sizes that are powers of two. The string routines have been rewritten to take advantage of the VAX string instructions, and the I/O routines were fixed to do buffered I/O.

1.1.2.5. Network Servers

With the introduction of network servers in 4.2BSD a number of servers were introduced, most of these servers daemons would sit in the process table waiting for another process to use them. These daemons spent a vast amount of time sitting idle, and consuming up resources. Most of these daemons were eliminated from the process table by merging them into a single "Internet daemon." This daemon listens to all the service ports and forks the appropriate server process when a request for their service is requested. This eliminated as many as twenty processes from the process table.

1.1.2.6. Csh

The C-shell in 4.2BSD was grossly inefficient. When it was converted to run on 4.2BSD a set of routines were written to simulate the old jobs library. These routines would generate up to twenty system calls per prompt. The Csh has been modified to use the new signal facilities and this has cut the number of system calls in half. There have also been some additional tuning to cut the cost of frequently used features.

2. Functional Extensions

There were many new utilities added to 4.2BSD but many of them were not fully implemented. Many of these utilities have been cleaned up and unified both old and new features in 4.3BSD.

2.1. Kernel Extensions

Many changes have been made in expanding the limits of the kernel. Most of these changes were made to allow greater flexibility and expansion in the kernel.

2.1.1. Number of File Descriptors

The hard limit of 30 open file descriptors per process has been relaxed. The default per-process descriptor limit was raised from 20 to 64. This will allow full use of the many descriptor based services available.

2.1.2. Kernel Limits

Many internal kernel configuration limits have been increased by suitable modifications to the data structures. The physical memory has been changed from 8 megabytes to 64 megabytes, and the maximum number of mounted file system has been increased from 15 to 255. The maximum file size has been changed to 8 gigabytes, and the number of processes increased to 65536. The system has been tuned for 4-8 megabytes of physical memory.

2.1.3. Memory Management

The global clock page replacement algorithm used to have a single hand that use was used both to mark and reclaim memory. The first time it encountered a page it would clear its reference bit. If the reference bit was still clear on its next pass across the

page, it would reclaim the page. The use of the single hand does not work well with large physical memories as the time to complete a single revolution of the hand can take up to a minute or more. By the time the hand gets around to the marked pages, the information is usually no longer pertinent. During periods of sudden shortages, the page daemon will not be able to find any reclaimable pages until it had completed a full revolution. To alleviate this problem, the clock hand has been split into two separate hands. The front hand clears the reference bits, the back hand follows a constant number of pages behind reclaiming pages that still have cleared reference bits.

2.1.4. Signals

The 4.2BSD signal would push several words onto the normal run-time stack before switching to an alternate signal stack. In 4.3BSD this has been corrected, the entire signal handler's state is pushed on to the signal stack. Users can now write their own return exception handler.

2.1.5. System Logging

A system logging facility has been added that sends kernel messages to the syslog daemon for logging in /usr/adm/messages and possibly printing on the system console. This gives a finer control on the messages logged and eliminates the degradation in response during the printing of low-priority kernel messages.

2.1.6. Windows

The tty structure has been modified to hold the information about the size of an associated window or terminal. These sizes are useful to programs such as editors that want to know the size of the screen they are manipulating. Other programs which need the width and height of the screen have been modified to use this facility.

2.1.7. Configuration of UNIBUS Devices

The UNIBUS configuration routines have been extended to allow auto-configuration of dedicated UNIBUS memory held devices. This makes it easier to configure memory-mapped devices and corrects the problem of reseting the UNIBUS.

2.2. Functional Extensions to Libraries and Utilities

The changes to the utilities and libraries are to allow them to handle a more general set of problems, or to facilitate the same set of problems more quickly.

2.2.1. Name Server

The name resolution routines (*gethostbyname*, *getserverbyname*, etc.) in 4.2BSD used a set of database files resident on the local machine. If these files were changed on one system and then not distributed to the other systems on the network, this would cause inaccessibility of hosts or services on the network. These files may be replaced by a network name server that can insure a consistent view of the name space on a multi-machine network.

2.2.2. System Management

Rdist, is a new utility provided to assist system managers in keeping all their machines up to date with a consistent set of sources and binaries. New versions of *getty*, *init* and *login* merge the functions of several files into a single place, and allow more flexibility in startup of processes such as window managers.

A new utility keeps the time on a group of cooperating machines synchronized to within 30 milliseconds of each other.

2.2.3. Routing

Many bugs have been fixed in the routing daemon. It now understands how to deal with subnets and point-to-point networks

2.2.4. Compilers

The symbolic debugger *dbx* has had many new features added, and all known bugs fixed. *Dbx* has also been extended to work with the Pascal compiler. The fortran compiler *f77* has had many bugs fixed.

REFERENCES

[McKusick85] McKusick, M. K., Karels, M., Leffler, S., "Performance Improvements and Functional Enhancements in 4.3BSD." *Proc. of The Summer 1985 USENIX Conference.* Portland, Oregon, USA June 1985.

The cat -v discussion is irrelevant

David M. Tilbrook

Imperial Software Technology

ABSTRACT

"This feels like a Republican victory party ..." -- Vic Vyssotsky, 1985

The so-called UNIX-philosophy has been preached from the pulpits by the high-priests of orthodoxy at many a UNIX conference. Does this zealous fervour have any connection with the failure of UNIX to make any significant advances in recent years? Why are there large areas of computer science that seem to be ignored by the UNIX world and why is that when some areas are attempted on UNIX they prove to be as unworkable or cumbersome as they were on the more traditional environments? This paper is a highly personal view by one who has been dismayed by the failure of the UNIX community (himself included) to make any significant advances in real-time systems (whatever they may be), software engineering (something palatable at least) and a variety of other problems.

"The first clue something is wrong with APL is that whenever two APL users/programmers get together they start discussing extensions to the language." -- Tom Duff, 1975

This is not a attack on the 'cat -v' talk or ideas. For the most part I agree with Mr. Pike's main points. However, I do feel that there have been two major problems with respect to the original talk:

1) Many people reacted with misdirected hostility to BSD, a system that has made tremendous contributions to the community in some areas and certainly provides a much superior environment to other available environments for certain applications;

2) The arguments were largely expending energy in the wrong direction, concentrating on minor issues (i.e., stylistic points such as flags to commands) rather than fundamental problems with current UNIX implementations and uses.

I am not going to defend the first point. There are large parts of the BSD system that I find dismaying. However, my experience with the commercially available alternatives has been far from pleasant [1]. Rather, I would like to put the case that the arguments and discussions about UNIX are highly reminiscent of the APL hacks discussing the ravel operator. The fundamental limitations of the system are not going to be overcome through either stylistic adherence to a set of loosely defined principles or the power coding of myopic hackers in universities of vulture capital shops.

It is high time that the UNIX research community recognize and accept that it is time to apply one of the so-called UNIX philosophy principles; that one should be prepared to throw out tools and start again, and the tool that should be thrown away is UNIX itself.

To defend this position is difficult and unpopular. Such a move threatens many people. There is a huge investment in UNIX at both the individual and corporate levels. Indeed I am not proposing that everyone rip up their licenses. There is a large segment of the UNIX world that must continue to use UNIX and will do so for a long time. However, UNIX's successor is not going to be reached by constant enhancement of the current system: neither is it going to evolve in parallel with what is becoming a major obsession within the UNIX community, that of satisfying the market place.

It is essential to recognize that the evolution of UNIX thus far has been less than graceful and the

[1] When the Cambridge conference panel was asked to choose between 4.2bsd and System V, five of the six members chose 4.2bsd. The sixth chose V8, which is not commercially available.

'real' progress less than spectacular [2]. I recognize that there have been improvements in some ideas and facilities and that some have definitely been of major importance to the computer science community (e.g., Make). However, there is little evidence that there will be any change in the way UNIX evolves and a great deal of evidence that progress is going to be hampered by commercial interests.

This gradual decrease of progress and the accompanying increase in complexity and problems (i.e., bugs) in a system's 'middle age' is not unprecedented. In fact it seems inevitable. What sufficed or was deemed inessential in a system's beginning in the interest of meeting initial requirements or objectives inevitably become insufficient and essential as the system's use and its users' objectives change.

In the case of UNIX, the initial developers created a system that was the well designed integration of four or five good ideas. It was relatively conventional in approach, on a popular machine, fairly small and understandable, etc., etc., etc. It was not designed to do everything. In fact certain areas were deliberately excluded (e.g., Read-time, IPC, data-base support) and it was there that the problem began.

There were many efforts to shoe-horn the missing parts into the UNIX environment. This was done out of a need to handle certain applications (e.g., real-time in MERT) while simultaneously taking full advantage of the tools and facilities offered by the base system. Sometimes UNIX was sold into areas which were completely outside the UNIX realm and required substantial developments to satisfy the needs of those areas. For example many of us are guilty of building applications which are ill-suited to UNIX, but required if UNIX was to be available to us for our own uses (e.g., COBOL compilers, RT-11 emulators, data-base systems).

From a small two person development, UNIX exploded into a mega-project with rapidly expanding and diversifying objectives and applications, and as is inevitable, rapidly decreasing coherence and quality.

Furthermore, due to the nature of the UNIX community, some problems have been tolerated due to cost of rectification, and certain application areas have either been ignored (as not being of interest) or developed by teams who lacked sufficient UNIX-experience to ensure that their implementations were compatible with the rest of the system [3].

To catalogue all the sins and follies would take too long. The list of things UNIX does not handle or handles poorly should be obvious to anyone who has used it for any period of time or who has tried to bend it further than it yields (e.g., trying to ensure reliability when signals must be handled). My own major concerns relate to controlling multiple process applications (what facilities exist are primitive, undisciplined and complicated [4]), to documentation which is largely unchanged in 15 years (why do people still insist on orienting documentation to paper output [5]) and a seemingly inability of suppliers to adequately test their products (I mention no names). These are just some of many areas where UNIX, whilst offering a good solution for 80% of the problem, seems sadly deficient for the remaining 20%. Others have expressed views about its inadequacy on large systems (it was initially a small-machine system) and the problems with security and reliability which are likely to remain unsolved.

---

[2] I ignore the commercial and marketing success as being of little interest to the researcher except in that it has meant increased availability.

[3] Signal handling is an area that is largely unchanged since the early days of UNIX despite the fact that it is extremely difficult to create reliable systems that avoid all possible race conditions. SCCS is a splendid example of a non-UNIX tool in the way files are named, and input and flags are handled. Worst of all is the difficulty encountered in managing large numbers of related source files, facility of UNIX that has proved to be so important.

[4] There is a glimmer of hope that streams can offer some relief in the area of multiple process control and communication. However, it is unlikely, when commercially available, that a style of use will have evolved that will provide the 'standard' way of building systems from components. Part of UNIX's success is due to the fact that the developers used and tuned it for a number of years before its release to the outside world thus had time to experiment sufficiently to develop (perhaps unconsciously) a 'standard' approach to software and its combination.

[5] The answer is of course so they can sell books to a captive readership and documentation reading software packages to those who recognize paper is inadequate.

Having complained a great deal about the system, I should now propose a solution. However, I recognize that as I am a long-time UNIX user I am no longer qualified to create such a system. It is unlikely that the next generation of the research computing environment can come from the UNIX community itself. I hope that the developers will be aware of UNIX's strengths, however, a team whose primary background is UNIX will probably not recognize a solution to a problem (unless it is called grep) without trying to cast it into the UNIX mould.

If this is the case, what role can the UNIX research community play in the advance to the 'UNIX' replacement?

Our first priority is to recognize our true requirements for a computing environment. Such an evaluation should be done without regard to actual implementation considerations. This is an ambitious assignment, which must be done, however, to evaluate other solutions with respect to their acceptability. Such an evaluation should be done on the basis of how well it fulfills our particular needs, not our prejudices towards particular solutions.

Secondly we must understand how the basic UNIX features or facilities succeed or fail in fulfilling our needs. To demand the existence of a facility without an appreciation of its true importance or value can be dangerous. For example, it is unlikely that any system that does not provide some sort of hierarchical file organization would be acceptable to most of us. But why? Can we honestly reject a system that doesn't provide such a file system on that basis alone? Far too often users fail to recognize the difference between a requirement and a partial fulfillment of that requirement.

Finally we must be prepared to honestly accept UNIX's short-comings and be prepared to evaluate alternatives without prejudice. To continue to design or code around UNIX short-comings is too costly and it is unlikely that some of the rare 'real' advances (e.g., streams, C++) will be available without unacceptable costs [6].

The last task is the most difficult to accomplish. The UNIX community has worked very hard at promoting UNIX for their own needs and in many cases are now unable to separate the marketing hype from the true value. We have been telling each other how wonderful the system is for so long that we are in danger of suffering from the symptoms discussed in the following article from The Guardian, Saturday, August 31, 1985, entitled: "What the team thinks is wrong"

> "Group think" can be self-defeating, said Dr. Pat Shipley, an occupational psychologist, at a session on ergonomics, the science of the workplace.
>
> "Group think was used to describe the British Cabinet's deliberations leading to the Falklands war," said Dr. Shipley.
>
> The irrational dimensions of group think underlying impaired critical judgement included: the group protecting itself from adverse information with self appointed mind guards; stereotyping the enemy as too stupid to be a threat, or too evil to negotiate with; dealing with challenges to cherished values and assumptions by ignoring them or rationalising them away.

I hope that I am wrong.

---

[6] It has been announced that future releases of UNIX for the VAX will not be forthcoming from the original supplier and, from my experience, the alternative offered is not a pleasant one.

Recent Work in Unix Document Preparation Tools

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

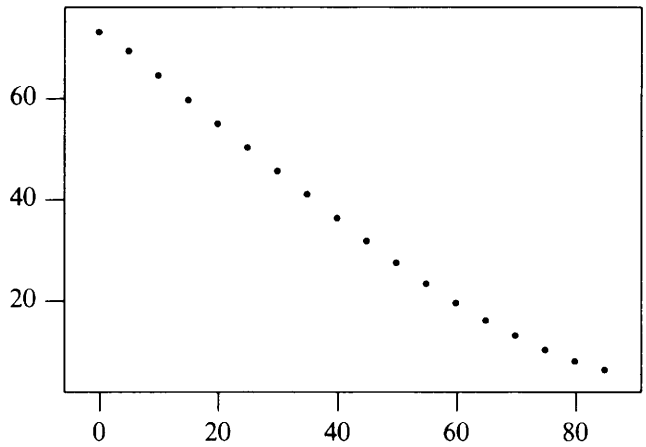
Document preparation based on *troff* continues to be an active area of research. This paper describes a new tool, *grap*, a program for typesetting graphs. *Grap* is a preprocessor for *pic*, rather than the usual *troff* preprocessor. Although originally intended only for document preparation, it has also been used for algorithm animation and exploratory data analysis, and has served as an "assembly language" for several compilers for specialized graphs.

The paper also describes enhancements to *pic*, particularly built-in functions and control-flow primitives, that permit the creation of figures of some complexity.

1. The GRAP Language for Graphs

In most document preparation systems, the only way to include a graph is by (mechanical or electronic) cutting and pasting of a separately prepared figure. The *grap* language for describing graphs [1] is meant to make it easy to include them in documents prepared with *troff* and the other document preparation tools [2] on the Unix system. *Grap* was designed and implemented by J. L. Bentley and the author.

In its simplest use, *grap* converts a set of *x,y* pairs into a scatter plot, generates ticks automatically, and puts the result in a standard frame, as in this plot of remaining life expectancy as a function of age:

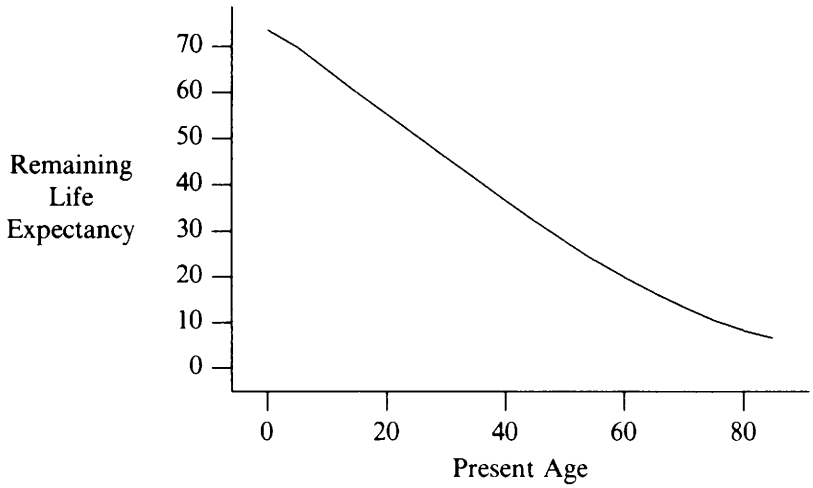


Normally a graph is part of a larger document. The parts of the document intended for *grap* are delimited by the commands `.G1` and `.G2`; everything else is copied through untouched. The input for the graph above is just the data itself:

```
.G1
0       73.6
5       69.8
10      64.9
...
85      6.5
.G2
```

The default display may be refined by specifying more parameters. Labels may be added on any side, ticks may be placed by an explicit list or an iterator, data may be copied from a separate file, and the points may be connected by lines of various styles:

```
label bottom "Present Age"
label left "Remaining" "Life" "Expectancy" left .3
ticks left from 0 to 70 by 10
frame top invis right invis
draw solid
copy "life.d"
```



The file `life.d` contains the age-expectancy data shown above. The clause `left .3` moves the text from its default position.

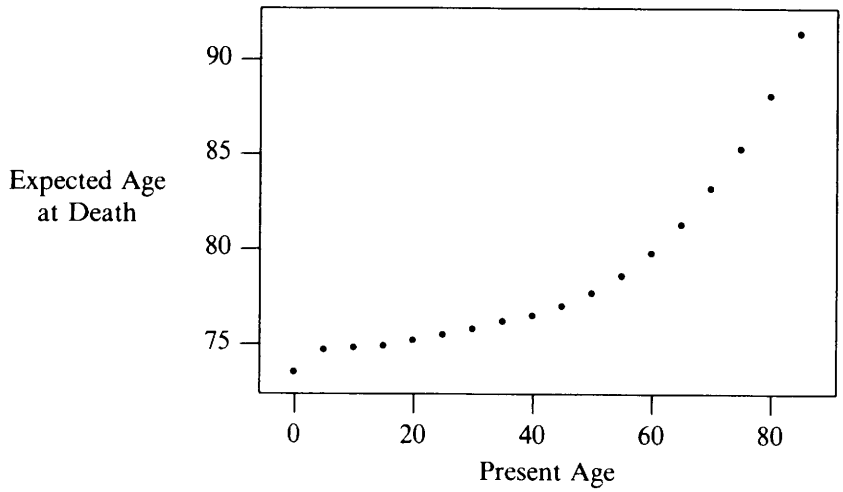
The central core of *grap* includes commands for plotting aribtrary text at any point, drawing arbitrary lines and arrows, setting range and optional logarithmic scaling of coordinate axes explicitly, and drawing grid lines.

Grap does not attempt to provide a large variety of built-in graph types. Rather, it offers primitive operations out of which many different graphs can be built. One of the most important of these primitive operations is a simple macro processor.

`define` *name* `{` *replacement text* `}`

defines a macro. Subsequent occurrences of *name* will be replaced by the *replacement text*. Instances of **$1**, **$2**, etc., in the replacement text will be replaced by the corresponding arguments in a macro call like *name*(arg_1,arg_2,...).

To illustrate, consider plotting expected age at death rather than remaining years, for which the y coordinate is the sum of age and expectancy:




```
label bottom "Present Age"
label left "Expected Age" "at Death" left .3
define show { bullet at $1, $1+$2 }
copy "life.d" through show
```

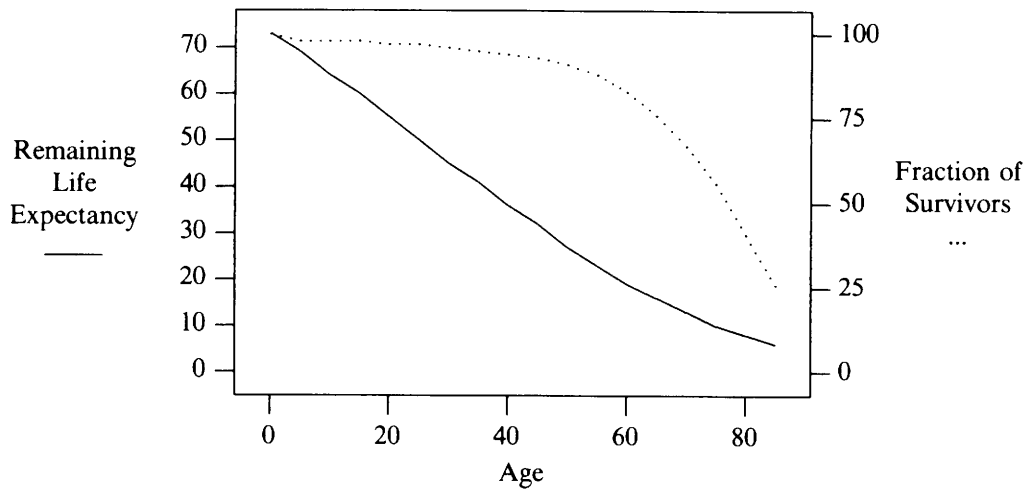
In a `copy` statement, each line of the source file is converted into a call of the specified macro, with each field becoming the corresponding argument. In fact, it is not necessary to define the macro separately:

```
copy "life.d" through { bullet at $1, $1+$2 }
```

is equivalent, and notationally more convenient.

As this example suggests, *grap* provides the ability to do arithmetic, both on input data and on variables. It also has an `if-else` statement and a `for` loop.

It is possible to show multiple curves on a single plot; each set of values is independently scaled and plotted. For example, this graph plots a second set of data that shows the fraction of an original 100 people still alive at the given age:

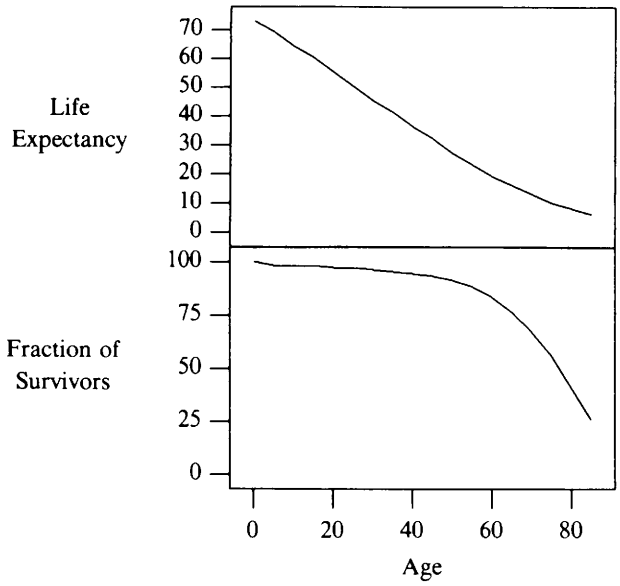


```
label bottom "Age"
ticks bottom from survivors 0 to 80 by 20
label left "Remaining" "Life" "Expectancy" "\l'.3i'" left .3
ticks left from expectancy 0 to 70 by 10
label right "Fraction of" "Survivors" "..." right .2
ticks right from survivors 0 to 100 by 25
draw expectancy solid
draw survivors dotted
copy "life3.d" through {
        next expectancy at expectancy $1, $2
        next survivors at survivors $1, $3
}
```

Data or parameters intended for a particular coordinate system are labeled with the name of that system.

One of the most useful features of *grap* is the ability to place several subgraphs in one overall graph. As a simple example, consider plotting the life expectancy and survivor data above as two separate graphs sharing a common *x* axis:

```
graph Expectancy
        frame ht 1.25 wid 2
        ticks left from 0 to 70 by 10
        tick bottom off
        label left "Life" "Expectancy" left .3
        draw solid
        copy "life3.d" through { $1, $2 }
graph Fraction with .Frame.north at Expectancy.Frame.south
        frame ht 1.25 wid 2
        ticks left from 0 to 100 by 25
        label left "Fraction of" "Survivors" left .3
        draw solid
        copy "life3.d" through { $1, $3 }
        label bottom "Age"
```



The **graph** statement defines a sub-graph with its own coordinate systems, data, etc. Subgraphs may be positioned arbitrarily with respect to previous subgraphs using *pic* positioning commands.

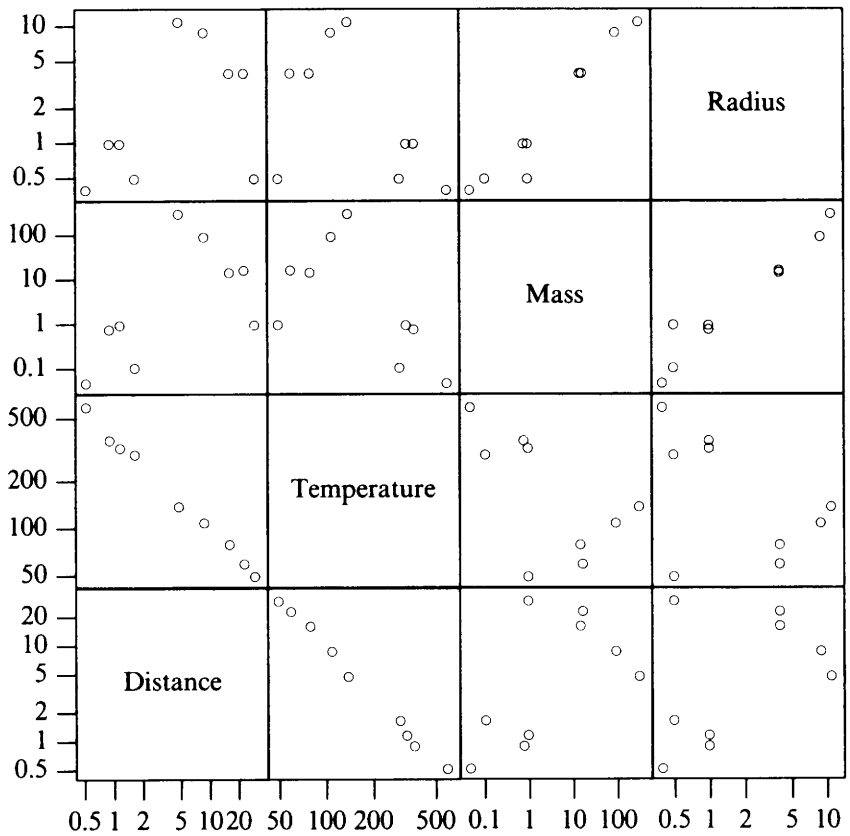
One unexpected use of *grap* has been as an assembly language for "compilers" for small,

specialized languages for preparing restricted kinds of graphs. The most interesting of these is a language called *scatmat*, for describing scatter-plot matrices; it makes heavy use of the facility for defining subgraphs.

Give a set of n observations of k attributes, a scatter-plot matrix is a $k \times k$ array of scatter plots. For example, given distance, temperature, mass and radius for the nine planets, as in

```
1       330     1       1
1.5     300     .11     .5
5       140     318     11
...
40      50      1       .5
```

a scatter-plot matrix would look like this:



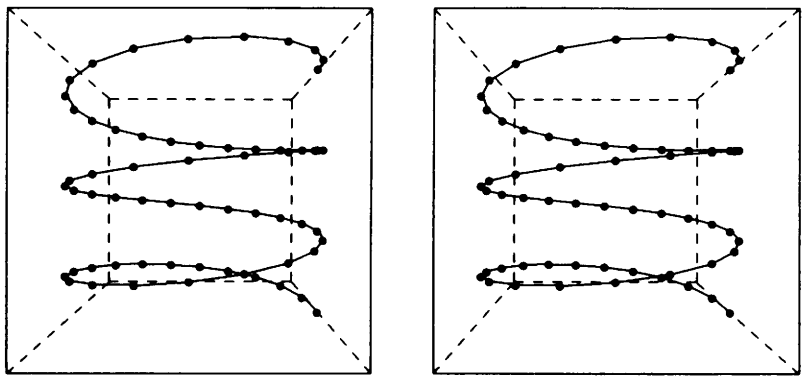
This graph is described in a simple language that is processed into *grap* by a small compiler written in *awk*. The input language looks like this:

```
file "planets.d"
frames ht 1 wid 1
spread 0
alllog
name Distance
        field $1
name Temperature
        field $2
name Mass
        field $3
name Radius
        field $4
```

2. PIC Enhancements

Pic is a language for drawing figures and diagrams, in the same spirit as *eqn* and *tbl*. The primitive objects in *pic* are lines, boxes, circles, ellipses and spline curves. Each object has attributes of size, position and associated text that may be set, either absolutely or in terms of previously defined objects. As in *grap*, there is a simple macro processor and facilities for doing arithmetic and storing the results in variables. In addition, *pic* has block structure, so that aggregate objects may be defined and treated as a unit.

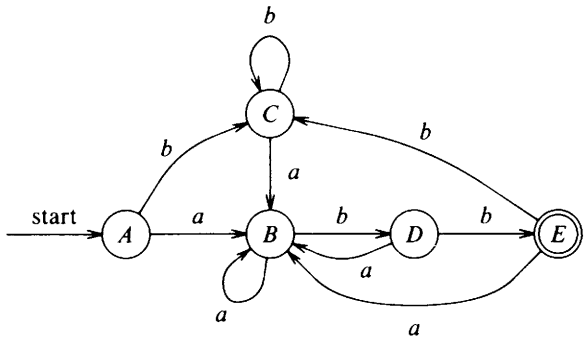
The development of the *grap* program made it easy to add several features to *pic* to make it more broadly useful than it was before. One trivial example is the addition of built-in functions for sine, cosine, log, exp, etc. Another change was to add the same `if-else`, `for` and `copy` statements as are found in `grap`. These made it possible, for example, to do stereo pictures (visible to readers who can cross their eyes):



The program is:

```
.PS
lpicx = 2; rpicx = 0; gs = 3.8; planez = -.5
eyey = .5; eyez = -1; leyex = .44; reyex = 1 - leyex
define bullet { "\s-4\(bu\s+4" }
        # plot point at $1,$2,$3
define p {
  tx=$1; ty=$2; tz=$3; sf=gs*(planez-eyez)/(tz-eyez)
  bullet at (lpicx,0) + (sf*(tx-leyex),sf*(ty-eyey))
  bullet at (rpicx,0) + (sf*(tx-reyex),sf*(ty-eyey))
}
        # line from $1,$2,$3 to $4,$5,$6 of type $7
define l {
  ax=$1; ay=$2; az=$3; bx=$4; bby=$5; bz=$6
  sfa=gs*(planez-eyez)/(az-eyez); sfb=gs*(planez-eyez)/(bz-eyez)
  line $7 from (lpicx,0)+(sfa*(ax-leyex),sfa*(ay-eyey))\
          to   (lpicx,0)+(sfb*(bx-leyex),sfb*(bby-eyey))
  line $7 from (rpicx,0)+(sfa*(ax-reyex),sfa*(ay-eyey))\
          to   (rpicx,0)+(sfb*(bx-reyex),sfb*(bby-eyey))
}
        # frame
  l(0,0,0, 0,1,0); l(0,1,0, 1,1,0); l(1,1,0, 1,0,0); l(1,0,0, 0,0,0)
  l(0,0,1, 0,1,1, dashed); l(0,1,1, 1,1,1, dashed)
  l(1,1,1, 1,0,1, dashed); l(1,0,1, 0,0,1, dashed)
  l(0,0,0, 0,0,1, dashed); l(0,1,0, 0,1,1, dashed)
  l(1,0,0, 1,0,1, dashed); l(1,1,0, 1,1,1, dashed)
tp = 2 * 3.1415926535; dx = .05; ub = 3
for x=0 to ub by dx do { p(.5+.5*cos(tp*x),x/ub,.5+.5*sin(tp*x)) }
for x=0 to ub-dx by dx do {
  l(.5+.5*cos(tp*x),     x/ub,     .5+.5*sin(tp*x), \
    .5+.5*cos(tp*(x+dx)),(x+dx)/ub,.5+.5*sin(tp*(x+dx)))
  }
.PE
```

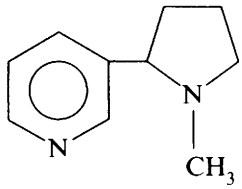
These features have also been used for some fairly complicated diagrams in the new edition of "the Dragon book" [3]:



Pic in its new form has also been used as the output for a program that implements a language for drawing chemical structure diagrams. The program itself is written in *awk*; its output is *pic*, with numerous calls to a set of macros that define common chemical structures like rings and bonds. For example, this input:

```
# Nicotine
benzene put N at 4
bond right
ring5 pointing down put N at 1
bond down from .N.s
CH3
```

produces this output:



The benzene ring is defined as a series of macro calls that looks like this:

```
define benzene {[
        makering(6, ringside, $1); ringlines(); ringcirc(.5)
]}
# makering(numverts, radius, rotation):  make symmetric ring
define makering {
        verts = $1; thisrad = $2; rot = $3; cr = 0.08
        if verts<3 || verts>6 then { illegal verts }
                                V0: circle invis rad cr at tt(-1)
                                V1: circle invis rad cr at tt(0)
                                V2: circle invis rad cr at tt(1)
                                V3: circle invis rad cr at tt(2)
                                V4: circle invis rad cr at tt(3)
        if verts >= 4 then {    V5: circle invis rad cr at tt(4) }
        if verts >= 5 then {    V6: circle invis rad cr at tt(5) }
        if verts >= 6 then {    V7: circle invis rad cr at tt(6) }
        C: 0,0
}
# ringlines(invis):  fill in all lines in this ring
define ringlines {
        for i=0 to verts-1 do { line $1 from tt(i) to tt(i+1) }
}
# tt(i) -- make vertex i of verts for this ring
define tt {
        (thisrad * sin((($1)/verts*360+rot)/57.296), \
         thisrad * cos((($1)/verts*360+rot)/57.296)) }
# ringcirc(relative radius): make circle at center of ring
define ringcirc { circle radius $1*thisrad at C }
```

One other improvement in *pic* has been the removal of all static limits on sizes or numbers of objects. Machine-generated input stresses programs quite differently than people do, so a program must be prepared to cope with a lot of input. Of course this applies to *grap* as well as *pic*, especially with input generated by *scatmat*.

3. Conclusions

Although often denigrated for appalling syntax and unpredictable semantics, *troff* remains an excellent tool for experimenting with document preparation. With the addition of *grap* and the enhancements to *pic*, our ability to cope with complicated graphical displays has been substantially improved.

It appears that many tasks can be profitably approached by designing and implementing a language specialized to that task, so that users speak in terms closely related to their problem. The

specialized languages for graphs, scatter-plot matrices and chemistry diagrams are examples from one domain, but many others can be found in other areas.

In most of these languages, it appears necessary to provide some degree of programmability; otherwise, users are restricted to those things that the implementor thought of. For *grap* especially, the ability to program the processor to define a new style has proven invaluable.

Acknowledgements

Grap was developed by Jon Bentley and the author; the *scatmat* program was also developed by Bentley. The program for typesetting chemistry diagrams is the result of collaboration among Bentley, Lynn Jelinski and the author. Bentley had nothing to do with *pic*, but he did force me to work on it, and the stereo picture is his. Probably he ought to be a co-author of this paper.

References

References

1. J. L. Bentley and B. W. Kernighan, *GRAP — A Language for Typesetting Graphs. Tutorial and User Manual,* AT&T Bell Laboratories CSTR 114 (December, 1984).
2. B. W. Kernighan, "The Unix Document Preparation Tools — A Retrospective," *Protext I Conference*, Dublin, pp. 12-22 (October, 1984).
3. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley (1985).

Principles of Type Design for the Personal Workstation

by Charles Bigelow

Composed in the Lucida® typefaces designed by Charles Bigelow & Kris Holmes for digital laser printing. Printed at the B&H studio on an Imagen 8/300 printer with resolution of 120 dots per centimeter.

THE PERSONAL COMPUTER WORKSTATION IS THE new tool of modern literacy. The appeal of the personal workstation comes from the immediacy of the relationship between person and machine, as expressed in a dynamic sequence of images on a video screen: an interactive dialogue between computer and user [1].

In his *Phaedrus*, Plato tells us that true knowledge comes from dialogue. The images in a human-computer dialogue are predominantly composed of text, and the text is composed of letters. The Latin word for letter, *littera*, gives us the [English] word "literacy". Modern "computer-literacy" is like traditional literacy — reading and writing. Therefore, legibility of text is a crucial factor in the usefulness of a personal workstation. If text written on a computer screen cannot be read, the system is useless.

Discussions [in English] of computer typography by computer engineers and scientists frequently use "font" as a general term for all forms of alphabet designs, though traditional typographers would distinguish "typefaces" from "fonts". Typefaces are images that are designed and read, whereas fonts are objects that are manufactured and distributed.

Accordingly, "typeface" means a group of characters whose forms are shaped in accordance with a particular set of design principles and which share certain design features. A typeface design is thus an abstract work of art intended to create a certain kind of visual image in the perception of the reader. A typeface may be independent of particular devices, though dependent upon, or influenced by, the kind of technology used to create the typographic image. In contrast, "font" means a concrete rendering of a typeface in a particular character set for a particular size-range for a particular imaging system. A font is thus a crafted, technical product which implements a design in a specific device or system. Fonts are linked to particular devices, even within one kind of imaging technology.

The most readable typefaces are transparent to the reader, presenting the text in the clearest way possible while remaining essentially invisible themselves [2]. The best text types are not noticed by the reader because their qualities are transferred to the reading experience. The visual pleasure of reading text composed in a readable type is perceived as part of the pleasure of the text. The converse is also true: a barely decipherable type is perceived as part of the difficulty of the text. The difficulty of deciphering poorly designed types on computer terminals may be manifested in other ways — as eye strain, headaches, or diffuse dissatisfaction with the work environment.

Until recently, there were few remedies to the problem of illegible types on video display screens. The standard "character generator" technology provided only a single font of one size of one style of coarse-resolution dot-matrix letters. Computer users and programmers could not modify such fonts because the character images were contained in hardware or firmware.

The video terminal characters themselves were too often designed by engineers untrained in letter-form design. Computer manufacturers concerned with hardware and software did not realize that typography is an essential part of the "user-interface" between person and machine, and that art is necessary for the design of legible text. Moreover, the design of computer characters was not attractive to traditional lettering artists because the computer tools were clumsy and mechanical in comparison to the responsive pens and brushes and receptive papers used in calligraphy and lettering.

Computer literacy has therefore been less pleasant and productive for the reader than traditional scribal and typographic literacy. The crude appearance of barely decipherable dot-matrix characters glowing on a rigid glass screen has prevented full acceptance of computers by a tradition-minded literate public. Readers are justifiably conservative about the shapes of letters because literacy requires an expensive educational investment from the individual and society. Any change in the appearance of letters that makes them less legible wastes precious educational resources.

But now the look of computer literacy is changing. The newer computer graphics technologies offer technical solutions to some of the aesthetic problems of digital screen types. These new techniques also pose new problems, to which the solutions will determine whether the personal workstation will be a worthy successor to the pen, the typewriter, the printing press, and the book.

The new generation of computer workstation uses raster graphics display technology. A raster image is a mosaic comprised of individual "pixels" (picture elements). In the simplest kind of raster, each pixel represents one of two possible values: black or white; on or off. An on/off pixel can be represented by a single "bit" (binary digit) of computer information. A two-dimensional "map" of such a binary image is called a "bit-map" [3].

A bit-mapped screen can display several sizes and styles of types in "windows" where images, texts, and programs can be stored, edited, and run. Windows can be like pages or documents arranged on a desk, as well as like virtual terminals accessing the computer system. Computer information is typographically displayed to the user with greater clarity than on a traditional terminal. The computer screen image is becoming more like a book, though it does not yet approach the typographic complexity of a modern newspaper [4].

However, the analog shapes of traditional typography cannot be directly transported to digital media. Each technology has its own unique way of rendering letterforms [5]. Printing types that were designed for hand casting and mechanical composition cannot be faithfully reconstructed as pixel fonts on a digital raster display screen. Digitization at low resolution degrades traditional typefaces. Text on display screens is obviously inferior to text in a well printed book. We can understand this if we consider the matter of resolution.

A raster image is a matrix composed of parallel rows and columns of marking dots which can be turned on or off at regularly spaced intervals, to form a mosaic image. Because the image is created by regular scanning of the display, the term "raster-scan" display may also be used [6]. The resolution of a raster device is determined by the number of image lines per centimeter. Early digital typesetting machines had resolutions in the range from 240 to 300 lines per centimeter, equivalent to an em square of 100 by 100 lines for a 10 or 12 point font. This was adequate for certain kinds of publishing, such as newspapers, but many traditional typefaces did not look as good in digital format as in their original analog letterpress form. Digital typesetters became popular for a broad range of typesetting applications when resolutions were raised to 400 and 500 lines per centimeter. Today, even higher resolutions of 700 - 900 lines per centimeter are used in some laser image setters which can render the subtlest details of delicate typefaces.

In comparison, CRT screens for terminals and workstations have resolutions in the range of 24 to 40 lines per centimeter — only 10 percent or less of the linear resolution of an average digital typesetter. In engineering terms, the typeface designs are "undersampled". Not only is important information lost, but noise is introduced in the form of false patterns — the familiar jagged stair-cases and crenellations of computer images [7].

The minimum resolution for high quality digital text can be estimated by reference to studies of visual perception. Experiments by psychophysicists and perceptual psychologists suggest that the visual system cannot detect spatial frequencies greater than 60 cycles per degree of visual angle [8]. For example, a bar grating of regularly spaced black and white lines will be perceived as a solid gray tint, rather than a grating pattern, if the spacing is so fine that more than 60 black and white line pairs are imaged in one degree of visual angle, as measured at the retina. This provides a measure of the upper limit of the visual system's ability to resolve the kind of detail produced by a digital raster. At a reading distance of approximately 30 centimeters, 60 cycles per degree of visual angle is equivalent to a resolution of 120 cycles per centimeter at the screen.

In the 1920's, Harry Nyquist, a mathematician studying digital signal processing at Bell Laboratories, showed that a signal can be sampled and reconstructed without loss or distortion if the sampling rate is *at least* twice the rate of the highest frequency in the original signal [9]. This minimum sampling rate is commonly known as the Nyquist limit, or the Nyquist rate. Sampling at a rate below the Nyquist limit introduces a form of noise called "aliasing," in which the high frequency components of the original signal are erroneously reproduced as spurious lower frequency components of the reconstructed signal. In digital typography, one form of these aliases is termed [in English] "jaggies"—the jagged stair-case patterns that fringe the edges of digital type. Other kinds of aliasing include distortions of stem weights and letter proportions.

To faithfully sample and reconstruct a signal of 120 cycles per centimeter, a minimum sampling rate of 240 lines per centimeter would be necessary. This Nyquist limit is only a theoretical minimum sampling rate. The difficulty of quantifying the mechanisms of human visual perception means that for high quality text images, real sampling rates often have to be higher than the theoretical minimum. The practical evidence suggests that today's screen resolutions of 24 - 40 lines per centimeter are at least one decimal order of magnitude too low to produce text of optimal visual quality. Traditional analog typefaces cannot be imitated and jaggies cannot be eliminated on today's display screens.

The design of low-resolution screen types is thus an excercise in "minimalist" lettering art. Al-

though the designer is constrained by technical limitations, there is also a fascinating aesthetic challenge to distill the essence of letterforms in a difficult medium.

How can we produce readable screen types without becoming mired in the intractable problem of imitating traditional analog types in a low-resolution digital medium that cannot reproduce the details of classical typefaces? The answer is to design digital types which follow the *principles* of readability found in traditional letterforms, while tuning the letter features and details to the low-resolution digital raster. Imitative letter design demonstrates that a new technology can produce decipherable text. Creative design shows that the technology can produce beautiful text.

During the Renaissance transition from script to print, early printing types were not successful copies of Humanist handwriting. A skilled scribe writing with an edged pen on vellum could produce a sharper, clearer, more lively and more rhythmic page of text than could an early printer confronted by rough, hand-made papers, soft lead-alloy types, uncertain recipes for printing inks, and imprecise methods of printing. Types that attempted to imitate handwriting were inevitably of inferior quality. We are told that some bibliophiles refused to allow printed books into their library. Within a few decades after the invention of printing, punch-cutters ceased to imitate and began to originate. Types were cut for optimum legibility in the printing medium. A new kind of letterform, based on the precise sculpting of refined contours rather than the real-time traces of a moving pen became the dominant look of literacy. The elegant engraving of Garamond replaced the lively script of Sinibaldi.

In discussing the effect of technology on writing, it is helpful to have names that denote the principles of formation of the different kinds of writing rather than names that denote particular historical or technical features. Scribal letterforms, such as handwriting and calligraphy, can be called *ductal*, because their essential shapes are traced out as the path of a moving pen. (The sequence and pattern of pen (or brush) strokes is called "ductus", from the Latin word *ducere*, 'to lead'.) Typographic letterforms, such as printing types, can be called *glyptal*, because their essential shapes are carved or engraved contours. (The compound term "grammatoglyptae", from the Greek words *grammata*, 'letters', and *glyphein*, 'to carve', was used to describe printing type by Aldus Manutius in the preliminaries of his Virgil edition of 1501. For those who prefer a Latin root, another term *sculptal* from Latin *sculpere*, 'to carve', would serve as well.) Both ductal and glyptal letters are analog forms — produced by smoothly varying changes — though the logic and evolution of the resultant letters are distinctly different, as can be seen in the histories of the Latin alphabet.

Digital letterforms, such as raster-based screen and printer types can be called "pictal", because their essential shapes are composed of patterns of discrete "pixels" (picture elements). (The terms come from the Latin word *pingere*, 'to paint, to tattoo, to embroider'.) In the lower resolutions, pictal letters appear to have closer affinities with mosaics, tile patterns, and pointillist paintings than with handwriting or traditional letterpress printing. The specifically pictal details of digital letters tend to obscure their basic functional qualities, and therefore it is important to look beyond the surface features to the deeper structure of the letterforms in order to design readable letters for digital rasters.

Because the digital raster is a *virtual* design medium without tactile or material characteristics, the designer of pictal letters cannot rely upon traditional tools such as pen, brush, or graver to help guide the resulting shapes. The pictal letter requires a rational analysis and orderly methodology for the resulting text image to be readable.

Letterforms have a *perceptual* structure as well as a conceptual structure; attempts to design types by logic and mathematics alone have failed. The traditional lettering artist learned intuitive principles of visual perception as one part of an apprenticeship that also emphasized correct tool use through long practice. Our task today is to rationalize as much of this traditional knowledge as possible so that pictal letters produced with computer tools will have the same degree of readability as letters produced by traditional craftsmanship.

The main principles to be followed in designing screen types are familiar to lettering designers: *Size; Weight; Contrast; Spacing; Proportion; Differentiation*. The difficult part of the task is applying these principles in the Procrustean digital grid.

Size

The screens of computer workstations are often viewed at a greater distance than books and typewritten documents. A reader can easily adjust a hand-held book to the ideal reading distance, but a computer terminal is more like an appliance; it is

heavy and requires a stable position with a certain amount of surrounding space on a desk or work surface. Additional paraphernalia such as a keyboard, mouse, or graphics tablet also tend to intervene between reader and screen. The ideal screen viewing distance depends on the legibility of the main text font and the characteristics of the display. Some ergonomic guidelines recommend a viewing distance range of 40 to 70 centimeters; other guidelines, a range of 33 to 50 centimeters. As an approximate guideline, screen fonts should be from 1.2 to 2 times as large as corresponding printed fonts.

This measure must be corrected for the fact that the apparent size of text in the Latin alphabet is dependent more on the height of lower-case letters, or capitals in German-language orthography, than on the nominal body size of the font. The x-heights of common text types range from about 40% to 60% of the type's body size. A type with an x-height of 50% of the body is of medium-large appearance, and thus a legible 10 point text face might have an x-height of about 5 points. If we assume a display screen with resolution of approximately 28 lines per centimeter (72 lines per inch) or one pixel per printer's point, then a corresponding screen font should have an x-height of 7 to 10 pixels, to adjust for the greater average reading distance.

Weight

The weight of a typeface is its relative density, or proportion of black image to white background. This density is sometimes called "color" by printers and typographers, but it is an achromatic color, based on the shade of gray of the text image rather than on hue. For typefaces of normal proportions, weight can be measured as the ratio between the thickness of a straight vertical stem (such as the stem of an 'l') and the x-height. The greater the stem thickness in proportion to the x-height, the heavier the weight, and the darker the text appears. Conversely, the smaller the stem thickness in proportion to the x-height, the lighter the face. Condensation and extension of the width of a face will also change the density, but most text faces will have normal width proportions.

The normal weight ratio of text types ranges from 5 to 6 stems per x-height. Book faces for extended reading may be slightly lighter, up to 6.5:1, and display faces heavier. Ratios lower than 5:1 generally make the face appear too dark for easy reading, and ratios greater than 6:1 may make the face appear too light.

This presents a difficult problem for the screen type designer. For example, on a screen with square pixels, a font with an x-height of 7 pixels and a stem weight of one pixel will be too light, but a stem weight of two pixels will be too heavy. The digital raster cannot permit non-integer stem weights, and thus an optimal ratio seems unachievable.

The arithmetic of the problem will be different if the nominal aspect ratio of the screen pixel is not square. Some popular personal computers have pixels which are vertical rectangles. These will make some sizes and proportions easier to design, but others more difficult, because the basic problem is raster resolution rather than aspect ratio. Yet, the square aspect-ratio is a better typographic design medium because it permits greater symmetry and is easier to comprehend than an arbitrarily proportioned rectangle. Most modern workstations use a square grid, and typographers should welcome this trend. Nevertheless, a skilled designer can produce good types in a non-square grid because the visual image, not the mosaic matrix, is the true typeface.

The actual pixels on a CRT display screen are not idealized squares and rectangles, but blurred circular or elliptical spots and lines. Therefore, the perceived stem thickness is almost always different than the nominal thickness computed from the specified raster resolution. Physical factors which influence perceived stem weight include: the size and intensity contour of the writing spot, the amount of spot overlap, the on-to-off speed of the writing beam vs. the off-to-on speed, the characteristics of the phosphor, and the brightness and contrast of the display. When the letterforms are black and the screen background is white, these factors may combine to erode away a significant portion of the perceived stem weight. Illuminated letters on a dark screen background may appear bolder for similar reasons. In many cases, the physical characteristics of the display influence the designer's choice of type proportions.

Other factors such as the ambient illumination of the room, the size of the display screen, and reading distance may also influence the acceptability of typeface weight. A weight ratio that appears acceptable in text on a large, brightly illuminated screen of 800 by 1000 pixels may appear too bold on a small, dimmer screen of 300 by 400 pixels.

Thus, there is an interaction between font size and stem thickness which makes some size/stem combinations significantly more legible than oth-

ers. The set of all possible low-resolution type proportions must be sifted to pass only those of acceptable weight ratios. The designer must attempt to understand the conditions under which the screen text will be read as well as the technical parameters of the screen.

Contrast

In traditional text typefaces and writing in the Latin alphabet, vertical letter elements are almost always thicker than horizontal elements. The stems of an 'n' are thicker than the serifs or the connecting arch; the vertical bowls of an 'o' are thicker than the horizontal hairlines. This noticeable difference between vertical and horizontal features can be called contrast. Faces with high contrast have a brilliant, glittery look, whereas faces with low contrast have a stolid, monotonous look. Non-Latin scripts, such as Hebrew, Arabic, and Devanagari (used for several languages of India) have an opposite contrast in which horizontal elements are thicker than the vertical.

Originally a result of the way the scribal writing tool was held and manipulated, contrast is preserved in glyptal and pictal typefaces because it aids recognition and discrimination of letterforms. For screen types to have some of the legibility of traditional typefaces, the traditional contrast must be preserved. Types in which the vertical and horizontal elements are the same thickness have an unfamiliar texture; this unfamiliarity impairs legibility.

When both horizontals and verticals are only one pixel in thickness, and the letters are black on an illuminated background, the problem is exacerbated by the erosion of vertical stems described above. Stems become even thinner than horizontals, contrary to the visual expectations of the reader. Such types not only appear weak and spindly, they seem unclear and ill-defined, as though the reader's vision were blurred, or something were misadjusted on the display screen. What is actually blurred and misadjusted is the design of the type. Note, however, that thicker stems require a larger x-height to maintain the proper weight, so there is a lower limit to the size at which contrast can be achieved on screen types.

Spacing

Lettering artists know that the "counters", the negative shapes of the background, are as important as the positive shapes of the letters. The relation between form and counter-form, between letters and their surrounding and internal spaces, is a crucial part of alphabet design. The vision researcher understands this in a different way: a text image is composed of spatial frequencies, regular alternations of dark shapes on a light background.

Experiments in vision research suggest that the human visual system is most sensitive to spatial frequencies in the range from 2 to 6 cycles per degree of visual angle. In the same way that a phrase of music may be composed of multiple voices in harmony, a line of text is composed of multiple spatial frequencies. The fundamental frequency of text is the regular alternation of black vertical stems with intervening white counters (the space inside a letter like 'n' or 'o') or inter-letter spaces. Estimates of the fundamental spatial frequencies of printing types at text sizes show a range from 4 to 6 cycles of degree of visual angle — within the range of peak sensitivity of the visual system. When the larger text sizes used in luxury books, where typographic economy is not a factor, are included in the estimates, the range expands to include 2 and 3 cycles per degree, the range predicted by vision research.

The sizes and spacings of type are not arbitrary; they have been carefully tuned to the mechanisms of the visual system, not by rational analysis, but by centuries, even millennia, of careful scribal experimentation. Screen types should also be tuned to this band of fundamental frequencies. Throughout the ages, scribes and type designers have painstakingly adjusted the spacing and fitting of letters to maintain a rhythmic and harmonious visual pattern in the line of text. This is equivalent to maintaining a regular fundamental frequency in the text image. Intentional interruptions in the basic frequency, such as word spaces, are thereby noticeable and significant. Accidental irregularities in the basic spatial frequency, such as dark tangles or light voids caused by poor letter spacing, also attract attention, but they impart no information. Irregular spacing is therefore noise that distracts the reader, interrupts the smooth flow of reading, and obscures the real textural information, thus impairing the legibility of the text.

A failing common to many screen types is irregular letter spacing, too tight in some combinations, too loose in others. This results from designing a type as a collection of individual letters rather than as an organized system of figure and ground. The negative space of the background must be designed simultaneously with the positive shapes of

the letters, and in many cases the designs of individual letters are shaped by the need to fine-tune the spacing of the entire font. Attempts to achieve the tight inter-letter spacing fashionable in advertising typography prevent good overall spacing, because they create disparities between letter combinations that can space closely, such as pairs like 'll', and those that must space widely, like 'vy'. Tight, irregular letter spacing is useful in advertising typography, where it attracts attention to sales "blurbs" that a reader would otherwise ignore, but readable text requires a regular rhythm. Jan Tschichold's didactic discussion of spacing in his *Meisterbuch der Schrift* is as relevant to screen types as to printing types [10].

Proportion

Because the alphabet is a system, the proportions of the letters must be tuned to each other and to the overall proportions of the alphabet design. The widths of the lower-case letters must conform to three main criteria: the x-height of the alphabet design; the optimal spatial frequency of the text; and our historically evolved letter shapes. The average width of the letters in relation to the size of the type determines the fundamental spatial frequency of the font at a particular reading distance. This frequency should be within a certain range, as discussed above. Moreover, the different widths of the letters in relation to each other help the reader to discriminate their forms.

Proportionally spaced types are more legible than monospaced types because of the more finely tuned pattern of the text. When monospaced types are a necessity, great care must be taken to compensate for the irregular rhythm and distorted proportions. The limitations of mechanical typewriter technology created a need for monospaced types, but these limitations are not technically necessary in digital typography, and therefore monospaced types can be retired from many text applications on display screens.

Differentiation

The alphabet is a semiological system of graphic signs which signify the phonetic elements of speech. In speech, these phonetic elements, sometimes called phonemes, are carefully differentiated from each other; therefore the letters of the alphabet must be similarly differentiated. A legible type comprises letterforms that are easily discriminated one from the other. The task of type design is then to ensure that the letterforms are unambiguous, discriminable, and distinguishable. Although resolution is limited for screen fonts, these goals can be achieved by concentrating on three areas: *serifs; primitive elements; asymmetry.*

Serifs. Serifs act as 'flags' on the character shapes to aid in discrimination. Note that in a sans-serif type, an 'r' followed by 'n' can easily be confused with an 'm', whereas the same combination in a serifed type is less easily confused. Similar demonstrations can be made for other combinations. such as 'c' + 'l' compared to 'd'. While sans-serif types may seem more modern (though they first appeared in 1816) and thus more appropriate to the computer, they are less legible for text because they lack these small but significant distinguishing elements. However, in the smallest screen sizes, ten pixels or fewer per body, there is so little information available for each letter that serifs become obtrusive elements which can alter the basic forms and interrupt the regular spacing of letters. The careful designer applies serifs sparingly to the letters of the smallest sizes.

Primitive Elements. Latin-based alphabetic characters are like molecules constructed from simpler atomic elements. These primitive elements are called "strokes" because they were originally a single motion of a pen or brush. The various kinds of strokes include: verticals, horizontals, curves, and diagonals. The alphabet can be sub-divided into various groups of letters made up of particular primitive elements. For example, in the lower case, the letters 'n', 'm', 'h', 'u', 'r' form one group based on the vertical straight stem and arch; 'o','c', 'e' form a group based on the curved bowl; 'b', 'd', 'p', 'q', form a related group based on the curve plus straight; and 'v', 'w', 'x', 'y' form a group based on the diagonal stroke.

These form groups help the reader to discriminate and distinguish the letterforms. To avoid the "jaggies", the digital noise that appears as annoying stair-case patterns on curves and diagonals of low-resolution fonts, many screen fonts have been reduced to combinations of straight vertical and horizontal elements. This design technique reduces the jaggies, but it also destroys the legibility of the font by eliminating two of the three basic primitive elements and collapsing the form groups together. When every letter in the alphabet resembles every other letter, discriminability is lost and the alphabet degrades toward indecipherability. It is preferable to maintain the traditional shape primitives

and to keep the letterforms unambiguous, even if the diagonals and curves show jaggies.

Asymmetry. The principle of construction from primitive elements must not be applied in a simple-minded way. Traditional letterforms, while related by constructive principles, are neither rigidly symmetrical nor strongly assimilated. In particular, the upper portions of the lower-case characters are the most carefully differentiated parts of an alphabet design. The gaze of the reader appears to fixate more on the "x-line" (the top of the lower-case) than on the base line. When the forms of the lower-case are strongly assimilated toward one basic shape with vertical and horizontal mirror symmetry, the individual characters lose their identity. This can easily be seen in the set that includes 'a', 'b', 'd', 'g', 'p', and 'q'. The 'b', 'd', 'p', and 'q' should share many features, but they should not be strict mirror or rotational images of one another. Care must also be taken to prevent the 'a' and 'g' from being too closely assimilated to the others. Similar principles should be observed throughout the rest of the font design.

Screen + Printer: What You See, What You Get, What You Read

The goal of legible screen fonts assumes that the screen is where the text will be read. However, text written on a screen may also be printed on paper. There are now systems which attempt to integrate screen and printer typography, by representing the arrangement and typefaces of a document on the screen. These are called "What You See Is What You Get" (WYSIWYG) editing and layout systems. The WYSIWYG principle is that the screen should show exactly how the printed document will look. WYSIWYG text editors and document formatters attempt to show text in various typeface styles, sizes, spacings, and page organizations. Traditional typography offers so vast and complex a range of possibilities that present WYSIWYG systems can only offer a much restricted subset. It is inevitable that typographic information will be lost or distorted at screen resolutions.

A true WYSIWYG system is impossible to achieve, but the principle has been beneficial in focusing attention on legibility and the typographic design of documents. It can, however, lead to serious problems when rigidly and naively applied to real editorial situations.

Some WYSIWYG systems start with the screen resolutions and force the printer to conform to the limitations of the screen. In the simplest case, each screen pixel is mapped one-to-one onto the page of paper output by the printer. While this provides a certain cartesian satisfaction, since it can be logically demonstrated that the printer page is "exactly" like the screen display, the two images will actually appear very different. As discussed above, the screen characters are often eroded by the characeristics of the display technology, whereas the printed characters are usually emboldened, either by ribbon-spread on a dot-matrix printer or by toner effects on a "black-writing" laser printer. A "white-writing" laser printer will erode character weight, but often to a different degree than the screen fonts. Thus, if a font is tuned to the optimal weight and contrast on the screen, it will appear too dark and too low in contrast on the printer output. Conversely, if the fonts are tuned to the printer, they will often appear too light and too high in contrast on the screen. What you see is not what you will get, at the present level of display and printer technology.

A second problem with forcing the printer to match the screen is exaggeration of jaggies. The stair-case aliases on the screen are somewhat ameliorated by the soft, fuzzy contour of the CRT writing spot. The spot does not have sharp edges, nor is it square or rectangular; instead it is blurry and round. The low-contrast edges of the pixels tend to soften the apparent jaggies. Printers, however, produce a high-contrast spot at a higher resolution which clearly renders the edges of the staircase aliases. A laser printer with several times the resolution of the screen will render a single screen pixel with several printer pixels. This emphasizes the rectangularity of the raster, and further enhances the jagginess of the digital artifacts. Printer fonts that are constrained to simulate screen resolutions look noticeably inferior to printer fonts that are optimized to the full resolution and imaging characteristics of the printer.

A different WYSIWYG approach stores fonts as high resolution outline master images. These are scan-converted by the computer to the screen bitmap images to represent the sizes of typefaces chosen by the user. This "device-independent" method is intellectually appealing because the same master outline is used to produce all actual raster "glyphs" on the screen or on the printer or typesetter. However, the automatic scan-conversion algorithms do not perform as well as a skilled type designer at display screen resolutions. Current font data struc-

tures and scan-conversion algorithms do rasterization that is good at bitmap resolutions around 480 lines per centimeter, (equivalent to 200 x 200 pixels per em-square at 12 point size), acceptable at 240 lines per centimeter (100 x 100 pixels per em), mediocre at 120 lines per centimeter (50 x 50 pixels per em), incompetent at 60 lines per centimeter (25 pixels per em), and a hopelessly botched hash at 30 lines per centimeter (12 x 12 pixels per em). The best bit-map screen fonts are presently produced by designers hand-tuning the pixel patterns of each character and coordinating the overall structure of each alphabet. Various workstation tools called "bit-map editors" have been created to aid designers in this task.

Although the automatic systems are the work of skilled mathematicians, it is unlikely that we will see major improvements in automatic scan-conversion until the mathematicians develop more powerful structural descriptions of the alphabet while paying more attention to the actual appearance of the characters. This requires the active assistance of lettering artists during the design of the data structures and algorithms. Scan-conversion is a problem that is as much perceptual as mathematical. Letterforms must *look* right to the reader. Successful alphabet design, even at low resolutions, requires knowledge of historical forms as well as understanding of the mechanisms of perception. This understanding is intuitively perceived by artists, but not yet successfully analyzed by computer scientists. As Blaise Pascal, a great mathematician himself, wrote in his *Pensees*, "The reason why mathematicians are not intuitive is that they cannot see what is in front of them."

WYSIWYG spacing

It is difficult to match the letter spacing and fitting of a 28 line per centimeter screen font with a 120 line per centimeter printer font such that a given text will have the same words on each line, the same line endings and hyphenation, and occupy the same relative space on both screen and printer page. Engineers who build WYSIWYG systems tend to conceive of the coordination of screen fonts with printer fonts as a numerical problem in matching spacing values, and to believe that this is the most significant aspect of the WYSIWYG problem. Type designers tend to perceive it as two parallel problems in optimizing legibility, because they are thinking more about the needs of the reader than of the computer.

The "real" typeface is neither the screen font nor the printer font, nor the typesetter font, nor even the artist's drawings or the dot image on a bit-map editor, but the image in the mind of the reader. Solving the numerical problems of matching letter spacing is not enough. The text must also be readable. The spacing rhythms of the text, crucial for legibility, must not be tortured on a procrustean bed — stretched and truncated to fit an arbitrary numerical measure. Instead, both low and high resolution fonts must be developed in parallel — matched in spacing but optimized in legibility. This can be done most effectively when the typefaces are original designs, crafted for the digital media.

We must not forget that fonts are for reading. Inferior fonts degrade the entire information system at the crucial human interface. There is nothing to be saved by wasting expensive hardware, software, and human time by attempting to make do with inferior, semi-legible fonts.

There is increasing suspicion that the automated office has not provided the increases in productivity promised by computer system vendors. One reason is that the fonts on such systems have not been as legible as the traditional typewriter and printing typefaces familiar to the literate office worker. Modern office workers spend a vast amount of time reading digital fonts. Literate education in modern society is an immense investment matched by the expense of office workers' salaries and wages. Fonts of inferior legibility waste these investments.

Grayscaling

While designers have been working on aesthetic solutions to screen font design, engineers have been seeking technical solutions. It is expensive and difficult to increase the spatial resolution of CRT screens, but ideal workstations should be inexpensive — no more than the cost of a typewriter - and so other solutions are being explored. One idea is to use varying levels of pixel intensity to increase the display information from one bit per pixel (the black & white bit-map display of current workstations) to several bits per pixel (the grayscaled display of a few experimental and commercial workstations). A monochromatic but variable intensity pixel-map display is often called a grayscale display [11].

Grayscaled fonts contain more information and appear to better depict traditional letterforms, at

least when viewed in isolated words and phrases. Also, the low-contrast edges of grayscaled curves and diagonals reduce the visual effect of the jaggies. The letterforms appear smoother. There is experimental evidence that for small details, the visual system confuses intensity and size. Many researchers have therefore suggested that grayscaled text would better approximate the fine details of traditional alphabets, and thus be more readable than bitmap text.

The improved legibility of grayscaled fonts for continuous, long-term reading, is, however, only an hypothesis. There is also evidence that the eye relies upon high-contrast edges to focus the text image during reading, and that readers may suffer greater visual fatigue when reading text displayed on a CRT raster [12]. The soft, low-contrast edges of grayscaled fonts could actually reduce legibility by preventing the visual focusing mechanism from finding a clear, sharp edge. Our alphabet has evolved as a system of high-contrast edges. It is not yet certain whether the conservative eyes of readers will accept grayscaled text, nor whether grayscaled text is physiologically more difficult to read, despite its less jagged appearance. It may be that a "hybrid" grayscaled font in which the vertical stems have high-contrast edges but the curves and diagonals have gray valued edges would be less fatiguing during long reading sessions. At the moment, there are more speculations than firm knowledge.

Grayscaled fonts are also more expensive to display and more difficult to design. They require more bits of memory to store the gray value at each pixel, and more elaborate and stable CRT's and controller electronics. The shapes of grayscaled letterforms are inherently more dependent on precise control of brightness and contrast on the CRT monitor.

The design of gray valued characters is problematic because as yet we have no common agreement on what kinds of digital filters will optimize legibility when creating grayscaled low-resolution fonts from high-resolution master images. Pixel-by-pixel construction of grayscaled fonts by lettering artists could aid in the perceptual problem, since the artist could judge the designs by eye on the screen, but there are no effective pixel-editing tools for grayscaled fonts as yet. We do not yet have a clear concept of how such an editor should function for an artist, since the choice of gray value for each pixel is influenced by the values of the neighboring pixels. Furthermore, it may be difficult for artists to work with more than 8 levels of gray (3 bits of information per pixel), while some research suggests that 64 or 128 gray levels (6 or 7 bits of information per pixel) is theoretically optimal for screen font representation.

Proper letter spacing also remains a serious problem with grayscaled fonts. For economy, a computer system will store each letter in only one grayscaled version for a given font size, but spacing and fitting would be improved by using alternate versions of characters, selected according to their combination. Yet, if multiple versions are stored or automatically produced from high-level masters, there will be serious computational and memory burdens on the system. Special hardware would be necessary to convert fonts stored as outlines into grayscaled text "on the fly", line by line rather than character by character.

Grayscaled fonts show great promise, and several grayscale screen displays will certainly be produced in the next few years, but the problems of grayscale are subtle and not easily solved. Legibility for long-term reading and working at grayscaled display screens may prove elusive without further research and experimentation.

The Visual Editing of Text

Of course, legible fonts are simply the foundation of typography. The type designer builds and tunes the perceptual instrument, but authors and editors compose the text, and typographers and printers display it in its full-dress performance for the reader. Once readable types have been achieved on workstation screens, the next problem of computer literacy is to produce clear layouts and understandable arrangements — what Fernand Baudin has called "the visual editing of text" [13] and what Max Caflisch has called "the logical arrangement of information" [14]. Even with legible types, the computer display screen remains ignorant of the accumulated knowledge of our typographic traditions. It is an unexplored wilderness that beckons a new generation of typographers.

Conclusion

The personal workstation offers powerful tools to the literate person, but these tools are dependent upon typography: legible fonts in effective arrangements. Digital technology is presently limited in its ability to reproduce analog letterforms: traditional typefaces cannot be successfully reproduced

at current display screen resolutions. To optimize legibility, new fonts must be designed for the digital media. These fonts will be most effective if they take into account the nature of the human visual system, the logical and historical principles that shaped our present day alphabets, the characteristics of current digital imaging devices, and the conceptual structures underlying typographic variations and arrangements. Computer technology requires a typography that preserves the fundamental features of the literate image, but expresses them with new clarity in a new medium.

Footnotes/Bibliography

[1] "Alto: A Personal Computer", C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs, in *A Decade of Research: Xerox Palo Alto Research Center, 1970 - 1980*, R. R. Bowker, 1980; and *Computer Structures: Readings and Examples*, Siewiorek, Bell, and Newell, eds., McGraw-Hill.

"The Personal Computer Lilith", Niklaus Wirth, Report Nr. 40, Institut fuer Informatik, Eidgenoessische Technische Hochschule Zuerich, April 1981.

[2] "The Crystal Goblet, or Printing Should be Invisible", Beatrice Warde, in *The Crystal Goblet*, The World Publishing Co., 1956.

[3] "The Smalltalk Graphics Kernel", Daniel H. Ingalls, in *Byte* Vol. 6, Nr. 2, August 1981.

[4] *Smalltalk - 80: The Interactive Programming Environment*, Adele Goldberg, Addison-Wesley, 1984.

[5] "Digital Typography", Charles Bigelow and Donald Day, *Scientific American*, Vol. 249, Nr. 2, August 1983.

[6] *Principles of Interactive Computer Graphics*, William Newman and Robert Sproull, McGraw-Hill, 1979.

[7] "Some Quantization Effects in Digitally-Generated Pictures", Richard G. Shoup, Society for Information Display, International Symposium, 1973.

Transmission and Display of Pictorial Information, D.E. Pearson, Pentech Press, London, 1975.

[8] "Monocular vs. Binocular Visual Acuity", Fergus W. Campbell and D. G. Green, *Nature*, Vol. 208, Nr. 5006, 1965.

"Contrast and Spatial Frequency", Fergus W. Campbell and Lamberto Maffei, *Scientific American*, November, 1974.

[9] "Certain Topics in Telegraph Transmission Theory", Harry Nyquist, Bell System Technical Journal, 1928.

[10] *Meisterbuch Der Schrift*, Jan Tschichold, Otto Maier Verlag, Ravensburg.

[11] "The Display of Characters Using Gray Level Sample Arrays", John E. Warnock, Computer Graphics, Vol. 14, Nr. 3, 1980.

[12] "Visual Fatigue and Operator Performance with DVST and Raster Displays", Gerald M. Murch, Proceedings of the Society for Information Display, Vol. 24, 1, 1983.

[13] "Constellations & Configurations d'ecritures", Fernand Baudin, in *De Plomb, d'Encre, & de Lumiere*, Imprimerie Nationale, Paris, 1982.

[15] *Typographie braucht Schrift*, Max Caflisch, ATypI, c. 1980.

Simula and C - a Comparison

by
Georg P. Philippot
NCR Education Nordic Area
P.O.Box 2685 St.Hanshaugen
N-0131 OSLO 1
Norway

1. Introduction

A newcomer to the UNIX world, Simula is an object oriented language of the same family as C. They are both block structured, machine independent, general purpose high level languages. Though in principle you may use any of the two for any given task, there are individual areas for which one is better than the other.

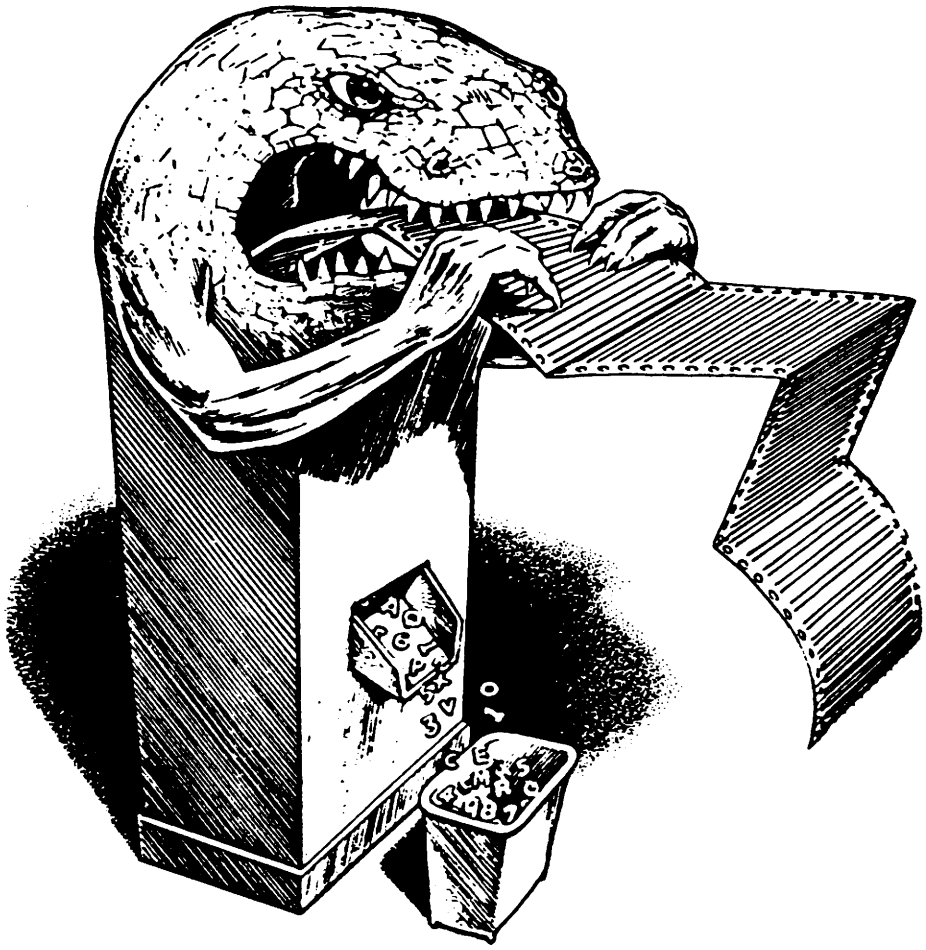
This paper describes the similarities and differences between two great languages, assuming some basic knowledge of C. It is hoped that this may give the audience some appetite for learning more about Simula and how it fits into a UNIX environment.

2. When are they equal?

As mentioned above, Simula and C belong to the same family of languages. This family also includes Pascal, Ada, and Modula, to mention a few. This, for one thing, means that a Simula programmer easily learns C and vice versa. Although they may look differently, programs in the two languages have the same structure, as can be seen from the stepwise development of this program:

```
main() {                        procedure main; begin
        initialiser();                  initialiser;
        oversett();                     oversett;
        }                               end;
initialiser() {                 procedure initialiser; begin
        }                               end;
oversett() {                    procedure oversett; begin
        }                               end;
```

As we will see, the compilation process (under the UNIX system) is



KOMPILERING

identical. The source program, on a .c or .sim file, is compiled and then linked to become an absolute program on a.out format. Even some of the preprocessor commands are equivalent, even though two different preprocessors are used: **#include** is **%compile**, **#if #else #endif** is **%if %else %fi**. Naturally, program libraries can be constructed in the same manner for Simula as for C.

Simula's types are basically those of Algol. In C we find the same types, though the names tend to be shorter. The only type which has no parallel is **unsigned**. Most constants are similar in both languages, for example, strings can either be fixed named locations or pointed to by **char** pointers.

Several control structures are equivalent: **if**, **for**, **while** exist in both languages. As we shall see later, C has at least one more that would be nice to have in Simula as well.

3. C advantages

The macro feature of C is unique. It allows the user to make tradeoffs between speed and program size with very few changes in the source code. Of course Simula might do this with a preprocessor - perhaps even **cpp** - but it doesn't. The only macro function supported by Simula is that of symbolic constants.

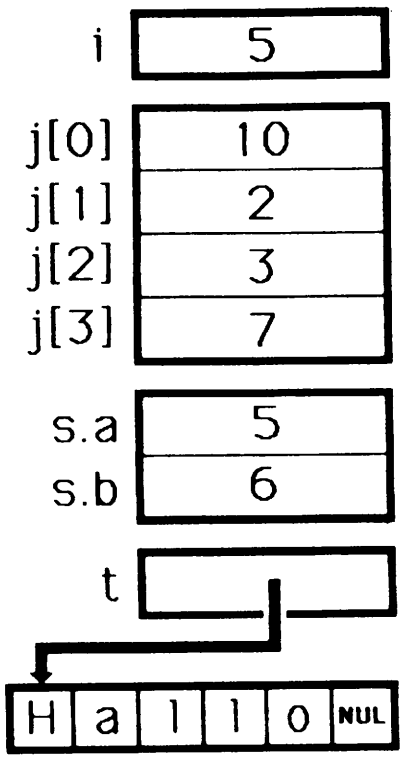
Due to the simplicity of C's array structure, it is also very flexible. Without any effort, programmers can create, for example, multidimensional arrays with variable row lengths. Simula can do the same thing using objects, but at an extremely high cost of space and time.

C allows general types, thus an object can be declared inline, not just generated dynamically. This possibility saves space and time overhead when the number and types of objects are known anyway.

All kinds of variables and inline objects, even arrays, can be initialised in their declarations. This has a documentary value, and it makes the program more compact. In most cases this form of initialisation is also faster.

C has a plethora of operators. Though most of these exist in all languages, C has the specialty of allowing most of the bit and shift manipulations that are normally only possible in assembly code. Of course you can do the same thing through special library procedures, but the program then soon becomes full of parentheses and procedure calls, and the intention of the bit operations becomes fairly unreadable. It is probably for this reason that C has been called "a high level assembly language".

```
int i=5;
int j[4]={10,2,3,7};
struct {int a,b;} s={5,6};
char *t="Hallo";
```



Here is the one control statement missing from Simula: A case statement. Strangely enough, this exists in most other languages of the family, but then Simula is one of the oldest, and the masters of that language have never managed to agree upon a suitable syntax for it!

Being brought up together with UNIX, C has a complete library in the sense that all UNIX system calls are available as routines. This is difficult to accomplish in other languages, since they do not develop in parallel with the operating system(s) they are used on.

I like the **printf** call. It is compact, flexible, readable for those who remember its conventions. It takes several different calls to generate the same output in Simula. The reason why Simula can never have a **printf** is simply that it does not support variable argument count and types.

4. Simula advantages

C has many useful features. One may be so much attracted to these, that one is led to believe the only disadvantage is that of being a little hard to read (for beginners). However, there are some strong cards up Simula's sleeve too.

Take the garbage collector, for example. There is no need to remember to free unused storage, and also no danger that someone may still be using it. All you have to do is release the last pointer to an object, and if you really did, then sooner or later the g.c. comes along and eats it up. This of course requires somewhat more disciplined use of pointers than is the case in C.

Simula objects support the notion of letting the objects themselves do appropriate actions determined by their type, also known as object oriented programming. An important tool for this is the possibility of declaring procedures local to objects. These may even be virtual, i.e. the same message from another object may trigger different procedures depending upon who is receiving the message.

In addition to **value** mode, procedure parameters may be transmitted by reference or by **name**. The specifications for this are entirely contained within the procedure declaration, so the caller need not know whether a value or an address should be passed.

FILE objects in Simula can exist also before being opened or after being closed. This is sometimes useful, for example when the same file is to be processed several times.

The long jump in C looks extremely clumsy. It is impossible to jump to somewhere you have never been before, and you have to keep label

Hovedprogram

```
jmp_buf wl;
int errorcode;
if (errorcode=setjmp(wl)) {
    ...opprydning, bruk
    errorcode til å velge riktige
    handlinger og/eller meldinger...
    return;
    };
```

Vilkårlig rutine i feilsituasjon

```
longjmp(wl, xxx);
```

variables for every potential jump destination. In Simula a simple **goto** is sufficient. Of course C has a good reason (stack balance) behind its implementation of long jumps, but this does not help the user.

5. Interesting and useful differences

Simula is a high level language, C is rather low. This only means they may have slightly different applications, not that one is inferior. We will look at some differences which may help you choose the right one in each case.

First of all, they look differently. Most Simula programmers think that C looks ugly. Fortunately, some **#define** statements will allow stubborn Simula fans to write C programs that almost look like Simula. However, time shows that they soon learn to appreciate the opportunity of writing compact programs, and forget about their beloved **begin end** symbols.

Arrays and pointers are a quite important issue in this comparison. Being very flexible in C, they are totally unsafe - nothing is checked by the C run time system, and "anything" can happen. The reward is speed. In Simula, on the other hand, there are restrictions - sometimes severe - but you can never use a wrong pointer or exceed an array bound without the friendly system's telling you at once. Similarly, the union is a very dangerous concept, only available in C, which allows memory to be saved by letting the same cell take on different meanings at different phases of the program.

The input/output system is actually quite similar. Both languages use a set of procedures for I/O, callable by the user in his own preferred sequence. They only have different names and parameters.

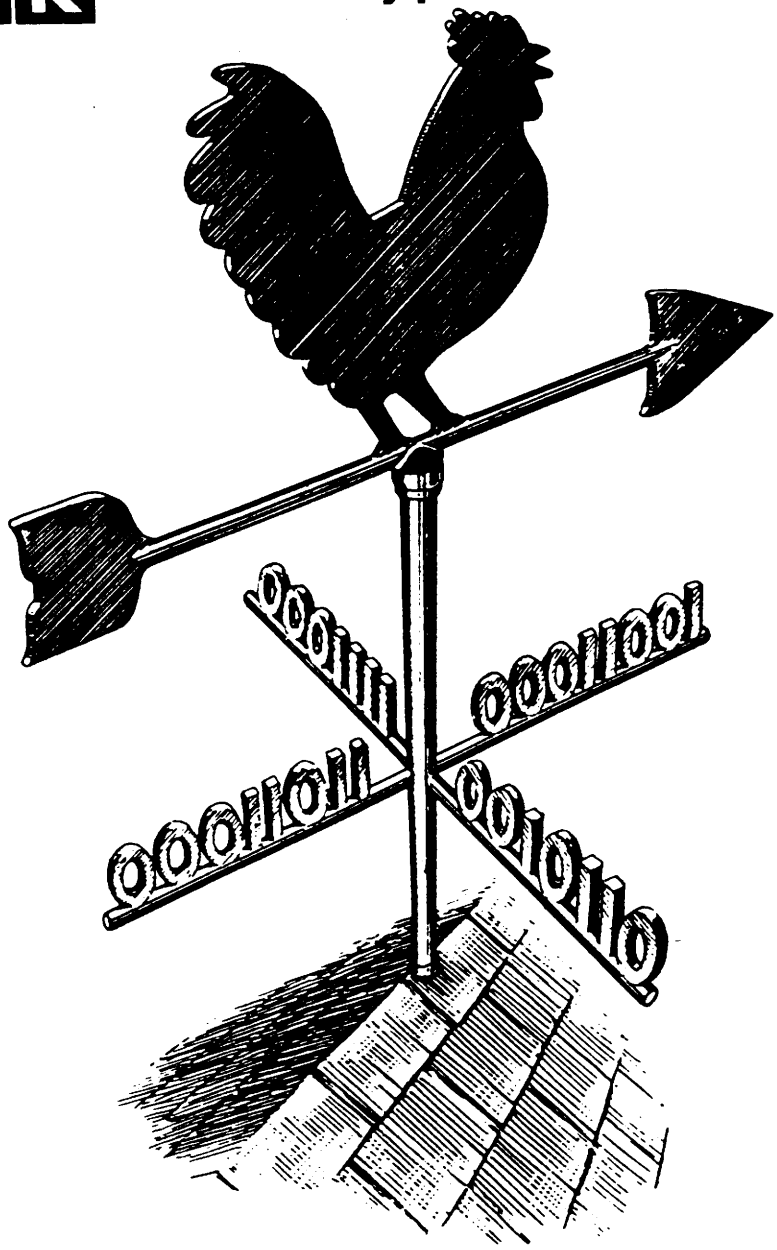
6. Conclusion

Judging from my own experience, I would recommend Simula for the safe development of both simple and complex programs. The checks built into the run time system save the programmer from wasting countless hours on debugging.

On the other hand, I would not hesitate to use C, even translate a Simula program into C, whenever speed can and should be increased by an order of magnitude. I can also recommend C for those areas of file and bit manipulation where a Simula program becomes clumsy.

The UNIX system allows the complete application to consist of a mixture of modules, each one programmed independently in that language which best supports that module. Thus one can make the considerations above for each individual section in turn, and then use pipes and a supervisory C

NCR Datatyper 4.6



PEKERE

390 C programmering NUC

program to connect these sections together.

7. References

[1] O.-J. Dahl, B. Myhrhaug, K. Nygaard:
SIMULA: Common Base Language (S-22)
Norwegian Computing Center, Oslo

[2] B.W. Kernighan, D.M. Ritchie:
The C Programming Language
Prentice-Hall

A C++ Tutorial

Bjarne Stroustrup

Bell Laboratories
Murray Hill, New Jersey 07974

"The only way to learn a new programming language is by writing programs in it" (K&R[1],page 5).

Abstract

This is a tutorial introduction to the C++ programming language. With few exceptions C++ is a superset of the C programming language. After the introduction, about a third of the text presents the more conventional features of C++: basic types, declarations, expressions, statements, and functions. The remainder concentrates on C++'s facilities for data abstraction: user-defined types, data-hiding, user-defined operators, and hierarchies of user-defined types. Finally there are a few comments on program structure, compatibility with C, efficiency and a caveat.

1 Introduction

This tutorial will guide you through a sequence of C++ programs and program fragments. At the end you should have a general idea about the facilities of C++, and enough information to write simple programs. Little is assumed about your knowledge of programming, but the progress through the concepts may be mind-boggling if you are a novice. If you are familiar with C you will notice that with few exceptions C++ is a superset of it. However, the examples have been chosen so that only few could have been written identically in C.

A precise and complete explanation of the concepts involved in even the smallest complete example would require pages of definitions. To avoid this paper turning into a manual or a discussion of general ideas, examples are presented first with only the briefest definition of the terms used. Many of these terms are reviewed later when a larger body of examples are available to aid the discussion. Reference 2 contains a more systematic and complete discussion of C++.

Output

Let us first of all write a program to write a line of output:

```
#include <stream.h>

main()
{
    cout<<"Hello, world\n";
}
```

The line `#include <stream.h>` instructs the compiler to "include" the declarations of the standard input and output facilities into the program. Without these declarations the statement `cout<<"Hello, world\n";` would make no sense. The operator `<<` ("put to") writes its second argument onto its first (in this case, the string `"Hello, world\n"` onto the standard output stream `cout`). A string is a sequence of characters surrounded by double quotes; in a string the backslash character \ followed by another character denotes a single "special" character; in this case `\n` is the newline character, so that the characters written are `Hello, world` and newline.

The rest of the program

```
main() { ... }
```

defines a function called `main`. A program must have a function named `main`, and the program starts by executing that function.

Compilation

Where did the output stream `cout` and the code implementing the output operator `<<` come from? A C++ program must be compiled to produce executable code (the compilation process is essentially the same as for C, and shares most of the programs involved): The program text is read and analyzed, and if no error is found code is generated. Then the program is examined to find names and operators that have been used but not defined (in our case `cout` and `<<`). If possible, the program is then completed by adding the missing definitions from a library (there is a standard library and users can provide their own). In our case `cout` and `<<` were declared (in `stream.h`); that is, their types were given, but no details of their implementation were provided. The standard library contains the specification of the space and initialization code for `cout` and the code for `<<`. Naturally there are many other things in that library, some of which are declared in `stream.h`, but only the subset of the library needed to complete our program is added to the compiled version. The C++ compile command is typically called `CC`. It is used like `cc` for C programs; see your manual for details. Assuming the "Hello, world" program mentioned above is stored in a file called `hello.c` you can compile and run it like this (`$` is the system's prompt):

```
$ CC hello.c
$ a.out
Hello, world
$
```

`a.out` is the default name for the executable result of a compilation.

Input

The following (rather verbose) conversion program prompts you to enter a number of inches. When you have done that it will print the corresponding number of centimeters.

```
#include <stream.h>

main()
{
    int inch;
    cout<<"inches=";
    cin>>inch;
    cout<<inch;
    cout<<" in = ";
    cout<<inch*2.54;
    cout<<" cm\n";
}
```

The first line of `main()` declares an integer variable `inch`. Its value is read in using the operator `>>` ("get from") on the standard input stream `cin`. The declarations of `cin` and `>>` are of course found in `<stream.h>`.

After executing it your terminal might look like this

```
$ a.out
inches=12
12 in = 30.48 cm
$
```

This example had one statement per output operation; this is unnecessarily verbose. The output operator `<<` can be applied to its own result, so that the last four output operations could have been written in a single statement:

```
cout<<inch<<" in = "<<inch*2.54<<" cm\n";
```

Input and output will be described in greater detail below. In fact, this whole tutorial can be seen as an explanation of how it is possible to write the programs above in a language that does not provide an input or an output operator! That is, the C++ language does not define facilities for input and output; instead, the operators >> and << were defined using only language facilities available to every programmer.

2 Types and Declarations

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that `inch` is of type `int;` that is, `inch` is an integer variable.

A declaration is a statement that introduces a name into the program. It must specify a type for that name. A type defines the proper use of a name or an expression. Operators like +, -, *, and / are defined for integers; so are, after `stream.h` has been included, the input operator >> and the output operator <<.

The type of an object determines not only which operations can be applied to it, but also the meaning of those operations. For example, the statement

```
cout<<inch<<" in = "<<inch*2.54<<" cm\n";
```

correctly treats the 4 values to be written out differently. The strings are printed as presented, whereas the integer `inch` is converted from its internal representation to a character representation fit for human eyes. So is the floating point number obtained by multiplying the integer `inch` by the floating point constant `2.54`.

C++ has several basic types and several ways of creating new ones. The simplest forms of C++ types are presented in the sections below; the more interesting ones are saved for later.

Basic Types

The basic types, corresponding most directly to hardware storage facilities are:

```
char  short  int  long  float  double
```

The first four are used for representing integers. A variable of type `char` is of the natural size to hold a character on a given machine (typically a byte), and a variable `int` is of the natural size for integer arithmetic on a given machine (typically a word). The range of integers that can be represented by a type depends on its size. In C++ "sizes" are measured in multiples of the size of a `char`, so by definition `char` has size one. The relation between the integer types can be written like this:

$$1 \equiv sizeof(char) \leq sizeof(short) \leq sizeof(int) \leq sizeof(long)$$

In general it is unwise to assume more about the sizes of integers. In particular, it is not true for all machines that an integer is large enough to hold a pointer.

`Float` and `double` are used for representing floating point numbers.

$$sizeof(float) \leq sizeof(double)$$

The adjective `const` can be applied to a basic type to yield a type that has identical properties to the original, except that the value of variables of a `const` type cannot be changed after initialization.

```
const float pi = 3.14;
const char plus = '+';
```

Note that most often a constant defined like this need not occupy storage; its value can simply be used directly where needed. A constant MUST be initialized at the point of declaration, as shown above. For variables the initialization is optional, but strongly recommended. There are very few

good reasons for introducing a local variable without initializing it.

The arithmetic operators

```
+  (both unary and binary)
-  (both unary and binary)
*
/
```

can be used for any combination of these types. So can the comparison operators

```
==   (equal)
!=   (not equal)
<    (less than)
>    (greater than)
<=   (less than or equal)
>=   (greater than or equal)
```

though if you use `==` or `!=` on the result of floating point computations you are likely to get what you deserve. Note that integer division will yield an integer result: `7/2` is `3`. The operator `%` can be used on integers to produce the remainder: `7%2` is `1`.

In assignments and in arithmetic operations C++ will perform all meaningful conversions between the basic types so that they can be mixed freely. For example:

```
double d = 1;
int i = 1;
d = d+i;
i = d+i;
```

The compiler will, however, warn about loss of precision in the last assignment.

Derived Types

These operators create new types from the basic types:

```
*        pointer to
*const   constant pointer to
&        reference to
[]       vector of
()       function returning
```

For example,

```
char* p;
char *const q;
char v[10];
int f(char*);
```

declares `p` as a pointer to character, `q` as a constant pointer to character, `v` as a vector of 10 characters, and `f` as a function taking an argument of type `char*` and returning an integer. A pointer variable can hold the address of an object of the appropriate type. For example:

```
char c;
p = &c;
```

Unary `&` is the "address of" operator.

The name of a vector doubles as the name of its first element, so given the declarations above you could write:

```
p = v;       // means p = &v[0]
x = f(v);    // means x = f(&v[0])
```

All vectors have zero as their lower bound. The characters `//` starts a comments which terminates by the end of the line they occur on.

Functions will be explained in §4. References will be explained in §10.

3 Expressions and Statements

Except for a minor extension to the syntax of the `for` statement, C++ statements are identical to those provided by C, but note that in C++ local declarations are statements and can be mixed freely with other statements. Except for the addition of the scope resolution operator `::` (§14) C++ expressions are identical to those provided by C. If you know C, please skip this section. The discussion of expressions and statements below is very brief.

Expressions

C++ has a host of operators that will be explained if and where needed. However, it can be noted that the operators

```
!    (not)
~    (complement)
&    (and)
^    (exclusive or)
|    (inclusive or)
<<   (left logical shift)
>>   (right logical shift)
```

apply to integers, and that there is no separate data type for logical operations.

C++ has an assignment operator `=`, rather than an "assignment statement" as in some languages. Assignments can therefore appear in contexts where one might not expect them. For example `x=sqrt(a=3*x)`. This is often useful; for example `a=b=c` means assign `c` to `b` and then to `a`. Another aspect of the assignment operator is that it can be combined with most binary operators into "assignment operators". For example `x[i+3]*=4` means `x[i+3]=x[i+3]*4` except the expression `x[i+3]` is evaluated only once. This gives a pleasing degree of run-time efficiency without having to resort to optimizing compilers. It is also more concise.

Pointers are used extensively in most C++ programs. The unary `*` operator dereferences a pointer. For example, given `char* p; *p` is the character pointed to by `p`. An alternative way of expressing this is `p[0]`. In fact, `p[i]` is defined to mean `*(p+i)`. Not only can a vector name be used as a pointer but a pointer can be used as if it were the name of a vector. A vector name is a constant though, whereas a pointer is a variable unless declared otherwise. The increment operator `++` and the decrement operator `--` are often used for pointers.

Expression statements

The most common form of a statement is an expression statement; it consists of an expression followed by a semicolon. For example:

```
a = b*3+c;
cout<<"go go go";
lseek(fd,0,2);
```

Null statements

The simplest statement is the null statement,

```
;
```

it does nothing. It can, however, be useful when the syntax requires a statement, but you have no need for one.

Blocks

A block is a possibly empty list of statements enclosed in curly braces. For example:

```
{ a=b+2; b++; }
```

It enables you to treat several statements as one. The scope of a name declared in a block extends from the point of declaration to the end of the block. It can be "hidden" by declarations of the

same name in inner blocks.

If statements

The following example performs both inch to centimeter and centimeter to inch conversion; you are supposed to indicate the unit of the input by appending `i` for inches or `c` for centimeters:

```
#include <stream.h>

main()
{
    const float fac = 2.54;
    float x, in, cm;
    char ch = 0;
    cout<<"enter length: ";
    cin>>x>>ch;
    if (ch == 'i') {
        in = x;
        cm = x*fac;
    }
    else if (ch == 'c') {
        in = x/fac;
        cm = x;
    }
    else
        in = cm = 0;
    cout<<in<<" in = "<<cm<<" cm\n";
}
```

As can be seen the condition in an if-statement must be parenthesized. The `else` part may be omitted. For the input `10i` this program will produce

```
10 in = 25.4 cm
```

Switch statements

A switch-statement tests a value against a set of constants. The tests in the example above could have been written like this:

```
switch (ch) {
case 'i':
    in = x;
    cm = x*fac;
    break;
case 'c':
    in = x/fac;
    cm = x;
    break;
default:
    in = cm = 0;
    break;
}
```

The `break` statements are used to exit the switch-statement. The case constants must be distinct, and if the value tested does not match any of them the `default` is chosen.

While statements

Consider copying a string given a pointer `p` to its first character and a pointer `q` the target. By convention a string is terminated by the character with the integer value `0`.

```
while (*p != 0) {
    *q = *p;
    q = q+1;
    p = p+1;
}
*q = 0;
```

The condition following `while` must be parenthesized. The condition is evaluated, and if its value is non-zero the statement directly following is executed. This carries on until the condition evaluates to zero.

This example is rather verbose. The operator `++` can be used to express increment directly, and the test can also be simplified:

```
while (*p) *q++ = *p++;
*q = 0;
```

where the construct `*p++` means: "take the character pointed to by `p` then increment `p`".

The example can be further compressed since the pointer `p` is dereferenced twice each time round the loop. The character copy can be performed at the same time as the condition is tested:

```
while (*q++ = *p++) ;
```

which takes the character pointed to by `p`, increments `p`, copies that character to the location pointed to by `q` and increments `q`. If the character is non-zero, the loop is repeated. Since all the work is done in the condition, no statement is needed. The null-statement is used to indicate this. C is both loved and hated for enabling such extremely terse expression oriented coding.

For statements

Consider copying ten elements from one vector to another:

```
for (int i=0; i<10; i++) q[i]=p[i];
```

This is equivalent to

```
int i = 0;
while (i<10) {
    q[i] = p[i];
    i++;
}
```

but more readable since all the information controlling the loop is localized. The first part of a for-statement need not be a declaration, any statement will do. For example:

```
for (i=0; i<10; i++) q[i]=p[i];
```

is again equivalent provided `i` is suitably declared earlier.

4 Functions

A function is a named part of a program that can be invoked from other parts of the program as often as needed. For example, consider writing out powers of 2:

```
float pow(float,int);

main()
{
    for (int i=0; i<10; i++) cout<<pow(2,i)<<"\n";
}
```

The first line is a function declaration specifying `pow` to be a function taking a `float` and an `int` argument returning a `float`. A function declaration is used wherever the type of a function defined elsewhere is needed.

In a call each function argument is checked against its expected type exactly as if a variable of

the declared type were being initialized. This ensures proper type checking and type conversion. For example, a call `pow(12.3,"10")` will cause the compiler to complain because `"10"` is a string and not an `int`, and for the call `pow(2,i)` the compiler will convert the integer constant `2` to a `float` as expected by the function.

`Pow` might be defined as a power function like this:

```
float pow(float x, int n)
{
    if (n < 0) error("sorry, negative exponents");

    switch (n) {
    case 0:   return 1;
    case 1:   return x;
    default:  return x*pow(x,n-1);
    }
}
```

The first part of a function definition specifies the name of the function, the type of the value it returns (if any), and the types and names of its arguments (if any). A value is returned from a function using a return-statement as shown.

Different functions typically have different names, but for functions performing similar tasks on different types of objects it is sometimes nicer to let these functions have the same name. When their argument types are different the compiler can distinguish them anyway. For example, one could have one power function for integers and another for floating point variables:

```
overload pow;
int pow(int,int);
double pow(double,double);
...
x = pow(2,10);
y = pow(2.0,10.0);
```

5 Program structure

A C++ program typically consists of many source files, each containing a sequence of declarations of types, functions, variables, and constants. To refer to the same thing in two source files it must be declared appropriately. For example:

```
extern double sqrt(double);
extern istream cin;
```

The most common way of guaranteeing consistency between source files is to place such declarations in separate files, called "header files", and then "include", that is copy, those header files in all files needing the declarations. For example, if the declaration of `sqrt` was stored in the header file for the standard mathematical functions `math.h`, and you wanted to take the square root of `4` you could write:

```
#include <math.h>
// ...
x = sqrt(4);
```

Since a typical header file is included into many source files it does not contain declarations that should not be replicated. For example, function bodies are only provided for inline functions (§13) and initializers only for constants (§2.1). Except for those cases, a header file is a repository for type information; it provides an interface between separately compiled parts of a program.

In an include directive a file name enclosed in angle brackets like `<math.h>` above refers to the file of that name in a "standard include directory"; files elsewhere are referred to by names enclosed in double quotes. For example

```
#include "math1.h"
#include "/usr/bs/math2.h"
```

would include `math1.h` from the user's current directory and `math2.h` from the directory `/usr/bs`.

6 Structures

Let us define a new type `ostream` to represent an output stream. The first version is trivially simple, but it will be refined until you get a feel for the real `ostream` used in the stream i/o system. The idea is to put characters into a buffer `buf` until it is full and then write `buf` to a file `file`:

```
struct ostream {
    FILE* file;
    int nextchar;
    char buf[128];
};
```

You can now declare an output stream like this:

```
ostream my_out = { stdout, 0 };
```

The construct

```
= { ... }
```

is an initializer. The members of `my_out` are initialized in order, so that `my_out.file` is `stdout`, `my_out.nextchar` is zero and `my_out.buf` uninitialized. (The `.` (dot) operator is used to access a member of a structure; `stdout` is the "standard output stream" of the underlying operating system. The basic output operation `write` can be used for `stdout`).

A simple character output function can be defined for an `ostream` like this:

```
void putchar(ostream* s, char ch)
{
    if (s->nextchar==128) {
        write(fileno(s->file),s->buf,128);
        s->nextchar = 0;
    }
    s->buf[s->nextchar++] = ch;
}
```

The keyword `void` is used to indicate that `putchar` does not return a value. As shown, the `->` operator is used to get to a member of a structure given a pointer. This code is sloppy (why?), but will actually handle the simplest cases; by writing

```
putchar(&my_out,'H');
putchar(&my_out,'e');
// ...
```

you could eventually manage to say `Hello, world`.

Naturally you would next define a function like

```
void putstring(ostream* s, char* p)
{
    for (int i = 0; p[i]; i++) putchar(s,p[i]);
}
```

and a `putlong`, and a `putdouble`, etc.

7 Problems

Proceeding as described above you could get a quite acceptable i/o system: C standard i/o is designed along this line. To save writing you would add functions that implicitly applied to the most common output stream. For example:

```
void mputlong(long i)
{
    putlong(&my_out,i);
}
```

These functions produce a character string representation of their arguments. Versions that write that representation onto a string instead of a file are also useful:

```
void sputlong(char* s, long i)
{
    // ...
}
```

However, there are problems. The most obvious, the proliferation of function names, can be handled simply by giving them all the same name:

```
overload put;
void put(ostream*, char*);
void put(ostream*, long);
void put(ostream*, double);
// ...
void put(char*);
void put(long);
// ...
void put(char*, char*);
// ...
```

Worse, there is no formal connection between these `put` functions and type `ostream`. Suppose you wanted to change the representation of an `ostream`. In any but the smallest program there is no easy way of finding all the places a member of `ostream` was used, and supposing `ostream` was a type used by many programs, how would you find the programs needing modification after even the most trivial change? Reversing the order of declaration of `file` and `nextchar` would potentially affect every program on your system, and would also invalidate every initializer.

8 Classes

A solution is to split the declaration of `ostream` into two parts: a private part holding information only needed by its implementer, and a public part presenting an interface to the general public:

```
class ostream {
    FILE* file;
    int nextchar;
    char buf[128];
    void putchar(char);
public:
    put(char*);
    put(long);
    put(double);
};
```

Now a user can only call those three `put` functions, and only those can use the names of the data members. In other words a class is a struct whose members are private unless their declarations appear after the label `public`. For example

```
my_out.put("Hello, world\n");
```

calls `put` using the usual syntax for members. A member function can only be called for a specified object of its type. When in a member function, the object for which the function was called is

accessed through a pointer called `this`. In a member function of class `C`, the keyword `this` is implicitly defined as

```
C* this;
```

You can now write

```
void ostream::put(char* p)
{
    while (*p) this->putchar(*p++);
}
```

The `ostream` prefix is necessary to distinguish `ostream`'s `put` from functions called `put` in other classes. The function body can be simplified, however, since this use of `this` is optional; in a member function, member names used without qualification refer to the object for which the function was called.

```
void ostream::put(char* p)
{
    while (*p) putchar(*p++);
}
```

would have been enough, and that is the more typical way of writing member functions. Consequently, most uses of `this` are implicit.

A `struct` is actually defined as a `class` with all members public, so a `struct` can have member functions too.

Since the representation of an `ostream` now is private, output functions for user-defined types must be written in terms of the basic `put` functions. For example, if you had a type `complex` you could define a `put` function for it:

```
void put(ostream* s, complex z)
{
    s->put("(");
    s->put(z.real);
    s->put(",");
    s->put(z.imag);
    s->put(")");
}
```

It could be called like this:

```
complex z
// ...
put(&my_out, z);
```

This is actually not very nice: the syntax for printing a value of a "basic" type is different from the one needed to print a value of a user-defined type. Furthermore, you need to write a separate call for each value.

9 Operator Overloading

Both problems can be overcome by using an output operator rather than an output function. To define a C++ operator `@` for a user-defined type you define a function called `operator@` which takes arguments of the appropriate type. For example:

```
class ostream {
    // ...
    ostream operator<<(char*);
};

ostream ostream::operator<<(char* p)
{
    while (*p) putchar(*p++);
    return *this;
}
```

defines the `<<` operator as a member of class `ostream`, so that `s<<p` will be interpreted as `s.operator<<(p)` when `s` is an `ostream` and `p` is a character pointer. Returning the `ostream` as the return value enables you to apply `<<` to the result of an output operation. For example `s<<p<<q` is interpreted as `(s.operator<<(p)).operator<<(q).` This is the way output operations are provided for the "basic" types. Using the set of operations provided as public members of class `ostream`, you can now define `<<` for a user-defined type like `complex` without modifying the declaration of class `ostream`:

```
ostream operator<<(ostream s, complex z)
{
    return s<<"("<<z.real<<","<<z.imag<<")";
}
```

This will write the values out in the right order since `<<`, like most C++ operators, groups left-to-right; that is, `a<<b<<c` means `(a<<b)<<c`. The compiler knows the difference between member functions and non-member functions when interpreting operators. For example, if `z` is a complex variable `s<<z` will be expanded using the standard (non-member) function call `operator<<(s,z)`.

10 References

This last version of `ostream` unfortunately contains a serious error and is furthermore very inefficient. The problem is that the `ostream` is copied twice for each use of `<<`: once as an argument and once as the return value. This leaves `nextchar` unchanged after a call (the example above does work correctly, however; why?). A facility for passing a pointer to that `ostream` rather than the ostream itself is needed.

This can be achieved using "references". A reference acts as a name for an object; `T&` means reference to `T`. A reference must be initialized and becomes an alternative name for the object it is initialized with. For example:

```
ostream& s1 = my_out;
ostream& s2 = cout;
```

The reference `s1` and `my_out` can now be used in the same way, and with the same meaning. For example, assignment

```
s1 = s2;
```

copies the object referred to by `s2`, that is `cout` into the object referred to by `s1`, that is `my_out`. Members are selected using the dot operator

```
s1.put("don't use ->");
```

and if you apply the address operator you get the address of the object referred to:

```
&s1 == &my_out
```

The first obvious use of references is to ensure that a pointer rather than the object itself is passed to an output function (this is called "call by reference" in some other languages):

```
ostream& operator(ostream& s, complex z) {
    return s<<"("<<z.real<<","<<z.imag<<")";
}
```

Interestingly enough the body of this function is unchanged, but had you assigned to `s` you would now have affected the object given as the argument itself rather than a copy. In this case, returning a reference also improves efficiency.

References are also essential for the definition of input streams, since the input operator is given the variable to read into as an operand.

```
class istream {
    // ...
    int state;
public:
    istream& operator>>(char&);
    istream& operator>>(char*);
    istream& operator>>(int&);
    istream& operator>>(long&);
    // ...
};
```

Note that `istream` needs more functions than `ostream`, since type conversion applies to basic types like `int` and `long`, but not to pointers to those types.

11 Constructors

The definition of `ostream` as a class made the data members private. In particular it rendered the initialization

```
ostream my_out = { stdout, 0 };
```

illegal. Only a member function can access the private members, so you must provide one for initialization. Such a function is called a constructor and is distinguished by having the same name as its class:

```
class ostream {
    // ...
    ostream(FILE* fp);
    ostream(int size, char* s);
};
```

Here two were provided; one takes a file descriptor like `stdout` above for real output; the other takes a character pointer and a size for string formatting.

You can now declare streams like this:

```
ostream my_out(stdout);
char xx[256];
ostream xx_stream(256,xx);
```

Providing a class with a constructor not only provides a way of initializing objects, but also ensures that all objects of that class will be initialized. It is not possible to declare a variable of a class with a constructor without a constructor being called. If a class has a constructor that does not take arguments, that constructor will be called if no arguments are given in the declaration.

12 Vectors

The vector concept built into C++ was designed (for C) to allow maximal run-time efficiency and minimal store overhead. It is also, especially when used together with pointers, an extremely versatile tool for building "higher level" facilities. You could, however, complain that a vector size must be specified as a constant, that there is no vector bounds checking, etc. An answer to such complaints is "you can program that yourself". Let us therefore test C++'s abstraction facilities by trying to provide these features for vector types of our own design and observe the difficulties

involved, the costs incurred, and the convenience of use of the resulting vector types.

```
class vector {
    int*v;
    int sz;
public:
    vector(int);
    ~vector();
    int size() { return sz; }
    void set_size(int);
    int& operator[](int);
    int& elem(int i) { return v[i]; }
};
```

The function `size` returns the number of elements of the vector, that is indices must be in the range `[0..size()-1]`; `set_size` is provided to change that size; `elem` provides access to members without checking the index, and `operator[]` provides access with bounds check.

The idea is to have the class itself be a fixed sized structure controlling access to the actual vector storage which is allocated by the vector constructor using the free store allocator operator `new`:

```
vector::vector(int s)
{
    if (s<=0) error("bad vector size");
    sz = s;
    v = new int[s];
}
```

You can now declare `vectors` very nearly as elegantly as "real vectors":

```
vector v1(100);
vector v2(nelem*2-4);
```

The access operation can be defined as

```
int& vector::operator[](int i) {
    if (0<=i && i<sz) return &v[i];
    error("vector index out of range")
}
```

The operator `&&` (andand) is a logical-and operator. Its right hand operand is only evaluated if necessary, that is, provided its left hand operand does not evaluate to zero. Returning a reference ensures that the `[]` notation can be used on either side of an assignment:

```
v1[x] = v2[y];
```

The function with the funny name `~vector` is a destructor. A destructor is called implicitly when a class object goes out of scope, so if you define it like this:

```
vector::~vector()
{
    delete v;
}
```

it will, using the `delete` operator, deallocate the space allocated by the constructor, so that when a `vector` goes out of scope all its space is reclaimed for potential re-use.

13 Inline expansion

Given the frequency of very small functions you might worry about the cost of function calls. A member function is no more expensive to call than a non-member function with the same number of arguments (remembering that a member function always has at least one argument), and C++ function calls are about as efficient as you can get for any language. However, for extremely small functions the call overhead can become an issue. If so, you might consider specifying a function to be "inline expanded". If you do, the compiler will try to generate the proper code for the function at the place of the call. The semantics of the call is unchanged. For example, if `vector::size()`

and `vector::elem()` were inline substituted:

```
vector s(100);
// ...
i = s.size()
x = elem(i-1);
```

would generate code equivalent to

```
// ...
i = 100;
x = s.v[i-1];
```

The C++ compiler is usually smart enough to generate code as good as you would have got from straightforward macro expansion. Naturally it will sometimes have to use temporary variables and other little tricks to preserve the semantics.

You can indicate that you want a function inline expanded by preceding its definition by the keyword `inline`, or, for a member function, simply by including the function definition in the class declaration, as was done for `size()` and `elem()` above.

Inline functions slow down compilation and clutter class declarations so they should be avoided when they are not necessary. For inline substitution to be a significant benefit for a function the function must be very small. When used well `inline` functions simultaneously increase the running speed and decrease the object code size.

14 Derived classes

Now let us define a vector for which a user can define the index bounds:

```
class vec: public vector {
    int low, high;
public:
    vec(int, int);
    int& elem(int);
    int& operator[](int);
}
```

Defining `vec` as

```
: public vector
```

means that first of all a `vec` is a `vector`. That is, type `vec` has all the properties of type `vector` in addition to the ones declared specifically for it. Class `vector` is said to be the "base" class for `vec`, and conversely `vec` is said to be "derived" from `vector`.

Class `vec` modifies class `vector` by providing a different constructor, requiring the user to specify the two index bounds rather than the size, and by providing its own access functions `elem()` and `operator[]()`. A `vec`'s `elem()` is easily expressed in terms of `vector`'s `elem()`:

```
int& vec::elem(int i)
{
    return vector::elem(i-low);
}
```

The scope resolution operator `::` is used to avoid getting caught in an infinite recursion by calling `vec::elem()` from itself (unary `::` can be used to refer to global names).

The constructor can be written like this:

```
vec::vec(int lb, int hb) : (hb-lb+1)
{
    if (hb-lb<0) hb = lb;
    low = lb;
    high = hb;
}
```

The construct `:(hb-lb+1)` is used to specify the argument list needed for the base class

constructor `vector()`.

This line of development of the vector type can be explored further. It is quite simple to provide multi-dimensional arrays (overload `()` as the access function), arrays where the number of dimensions is specified as an argument to a constructor, Fortran style arrays that can be accessed both as having two and three dimensions etc.

Such a class controls access to some data. Since all access is through the interface provided by the public part of the class, the representation of the data can in fact be changed arbitrarily to suit the needs of the implementer. For example, it would be trivial to change the representation of a vector to a linked list. The other side of this coin is that any suitable interface for a given implementation can be provided.

15 More about operators

An alternative direction of development is to provide vectors with operations:

```
struct Vec : public vector {
    Vec(int);
    Vec(Vec&);
    ~Vec();
    void operator=(Vec&);
    void operator*=(Vec&);
    void operator*=(int);
    // ...
};
```

Since a `Vec` has no private members (except the ones inherited from `vector`) it can be specified as a `struct`. The assignment operator is overloaded, and can be defined like this:

```
void Vec::operator=(Vec& a)
{
    int s = size();
    if (s!=a.size()) error("bad vector size for =");
    for (int i = 0; i<s; i++) elem(i)=a.elem(i);
}
```

Assignment of `Vec`s now truly copies the elements, whereas assignment of `vector`s simply copies the structure controlling access to the elements. However, the latter also happens when a `vector` is copied without explicit use of the assignment operator: (1) when a `vector` is initialized by assignment of another vector, (2) when a `vector` is passed as an argument, and (3) when a `vector` is passed as the return value from a function. To gain control in these cases for `Vec` vectors you define the constructor:

```
Vec::Vec(Vec& a) : (a.size())
{
    int sz = a.size();
    for (int i = 0; i<sz; i++) elem(i)=a.elem(i);
}
```

This constructor initializes a vector as the copy of another, and will be called in the cases mentioned before.

For operators like `=` and `+=` the expression on the left is clearly "special" and it seems natural to implement them as operations on the object denoted by that expression. In particular, it is then possible to change that object's value. For operators like `+` and `-` the left hand operand does not appear to need special attention. You could, for example, pass both arguments by value and still get a correct implementation of `+`:

```
Vec operator+(Vec a, Vec b)
{
    int s = a.size();
    if (s != b.size()) error("bad vector size for +");
    Vec sum(s);
    for (int i = 0; i<s; i++) sum.elem(i) = a.elem(i) + b.elem(i);
    return sum;
}
```

This function does not operate directly on the representation of a vector, indeed it couldn't since it is not a member. However, it is sometimes desirable to allow non-member functions to access the private part of a class object. For example, had there been no "unchecked access" function, `vector::elem()`, you would have been forced to check the index `i` against the vector bounds three times every time round the loop. This problem was avoided here, but it is typical, so there is a mechanism for a class to grant access to its private part to a non-member function. A declaration of the function preceded by the keyword `friend` is simply placed in the declaration of the class. For example, given

```
class vector {
    // ...
    friend Vec operator+(Vec, Vec);
};
```

you could have written:

```
Vec operator+(Vec a, Vec b)
{
    int s = a.size();
    if (s != b.size()) error("bad vector size for +");
    Vec& sum = new Vec(s);
    int* sp = sum.v;
    int* ap = a.v;
    int* bp = b.v;
    while (s--) *sp++ = *ap++ + *bp++;
    return sum;
}
```

One particularly useful aspect of the `friend` mechanism is that a function can be the friend of two or more classes. To see this consider defining a `vector` and `matrix` and then defining a multiplication function.

16 Generic vectors

"So far so good", you might say, "but I want one of those vectors for the type `matrix` I just defined". Unfortunately, C++ does not provide a facility for defining a class vector with the type of the elements as an argument. One way to proceed is to replicate the definition of both the class and its member functions. This is not ideal, but often acceptable.

You can use a macro processor to mechanize that task. For example, class `vector` presented above is a simplified version of a class that can be found in a standard header file. You could write:

```
#include <vector.h>

declare(vector,int);

main()
{
    vector(int) vv(10);
    vv[2] = 3;
    vv[10] = 4;    /* range error */
}

implement(vector,int);
```

The file `vector.h` defines macros so that `declare(vector,int)` expands to the declaration of a class `vector` very much like the one defined above, and `implement(vector,int)` expands to the definitions of the functions of that class. Since `implement(vector,int)` expands into function definitions it can only be used once in a program, whereas `declare(vector,int)` must be used once in every file manipulating this kind of integer vectors.

```
declare(vector,char);
// ...
implement(vector,char);
```

would give you a (separate) type of vector of characters.

17 Polymorphic vectors

Alternatively you might define your vector and other "container classes" in terms of pointers to objects:

```
class cvector {
    common** v;
    // ...
public:
    common*& elem(int);
    common*& operator[](int);
    // ...
};
```

Note that since pointers and not the objects themselves are stored in such vectors an object can be "in" several such vectors at the same time. This is a very useful feature for container classes like vectors, linked lists, classes, etc.

Furthermore, a pointer to a derived class can be assigned to a pointer to its base class, so the `cvector` above can be used to hold pointers to objects of all classes derived from `common`. For example:

```
class apple : public common { ... };
class orange : public common { ... };
// ...
cvector fruitbowl(100);
// ...
apple aa;
orange oo;
// ...
fruitbowl[0] = &aa;
fruitbowl[1] = &oo;
```

However, the exact type of an object entered into such a container class is no longer known by the compiler. For example, in the example above you know that an element of the vector is a `common`, but is it an `apple` or an `orange`? Typically that exact type must be recovered later in order to use the object correctly. To do this you must either store some form of type information in the object itself or ensure that only objects of a given type are put in the container. The latter is trivially

achieved using a derived class. For example, you could make a vector of `apple` pointers:

```
class apple_vector : public cvector {
public:
    apple*& elem(int i) { return (apple*&) cvector::elem(i); }
    // ...
};
```

using the "type casting" notation *(type)expression* to convert the `common*&` (a reference to a pointer to a `common`) returned by `cvector::elem` to an `apple*&`. The alternative, storing type identification in each object, brings us to the style of programming referred to as "object based".

18 Virtual functions

Consider writing a program for displaying shapes on a screen. The common attributes of shapes will be represented by class `shape`, specific attributes by specific derived classes:

```
class shape {
    point center;
    color col;
    // ...
public:
    void move(point to) { center=to; draw(); }
    point where() { return center; }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

Functions that can be defined without knowledge of the specific shape (for example `move`, and `where`), can be declared as usual. The rest is declared `virtual`, that is to be defined in a derived class. For example:

```
class circle: public shape {
    int radius;
public:
    void draw();
    void rotate(int i) {}
    // ...
};
```

Now if `shape_vec` is a vector of shapes as defined above you can write:

```
for (int i = 0; i<no_of_shapes; i++) shape_vec[i].rotate(45);
```

to rotate (and re-draw) all shapes 45 degrees.

This style is extremely useful in all interactive programs where "objects" of various types are treated in a uniform manner by the basic software. In a sense the typical operation is for the user to point to some object and say *Who are you? What are you?* or *Do your stuff!* without providing type information. The program can and must figure that out for itself.

19 Compatibility

C++ is not completely compatible with C, but it comes very close. In C++

```
int f();
```

declares a function that does not accept arguments; in C it declares a function that takes any number of arguments of any types (in C++, that can be stated as `int f(...)`). In C names of structures have their own name space separate from the one used for variable names; in C++ there is only one name space. To compile a C program as a C++ program you typically (only) need re-write your own header files (there are already C++ versions of the standard ones). In addition, C++ has 11 more keywords than C; these cannot be used as names of variables, etc. You can link C and C++ object files together without modification.

20 Efficiency

Run-time efficiency of the generated code and compactness of the representation of user defined data structures was considered of primary importance in the design of C++. A call of a member function is as fast as an equivalent (C++ or C) non-member function call with the same number of arguments. A call of a virtual function typically involves only three memory references extra. The representation of a class object takes up only the space needed for the data members specified by the user (allocated conforming to machine dependent alignment requirements). When virtual functions are declared for a class, objects of that class will containing one extra hidden pointer.

21 Caveat

Experienced C programmers have ended up with perfectly awful C++ programs because they immediately started using all the new features at once. It is worth remembering that most programs are best written without operator overloading, and using only a few examples of inline functions, private data, friends, references, derived classes, and virtual functions. Proceed with caution.

22 Acknowledgements

C++, as presented here could never have matured without the constant help and constructive criticism of my colleagues and users; notably Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Ravi Sethi, and Jon Shopiro. C++ clearly owes most to C; the influence of Simula67 is pervasive in the class facilities, and you may also notice Algol68 like facilities.

23 References

[1] B.Kernighan and D.M.Ritchie: The C programming Language, Prentice-Hall 1978.
[2] B.Stroustrup: The C++ Programming Language, Addison Wesley 1985.

Error Recovery for Yacc Parsers

Julia Dain
Dept. of Computer Science
University of Warwick
Coventry CV4 7AL
UK

We aim to improve error recovery in parsers generated by the LALR parser-generator *Yacc*. We describe an error recovery scheme which a new version of *Yacc* automatically builds into its parsers. The scheme uses state information to attempt to repair input which is syntactically incorrect. Repair by alteration of a single token is attempted first, followed by replacement of a phrase of the input. A parser for the C language is generated from existing specifications and tested on a collection of student programs. The quality of error recovery and diagnostic messages is found to be higher than that of the existing portable C compiler. The new version of *Yacc* may be used by any current user of *Yacc*, with minor modifications to their existing specifications, to produce systems with enhanced syntax error recovery.

1. Introduction

The portable C compiler *pcc* [Johnson78b] is widely used in UNIX environments but its diagnostic messages are poor. The parser for *pcc* is built by the LALR parser-generator *Yacc* [Johnson78a] which automatically generates error recovery routines. Many other popular UNIX utilities contain syntax analysers built by *Yacc*, such as the pattern matchers *lex*, *awk* and *grep* and the FORTRAN 77 and C++ compilers, and these utilities would also be easier to use if they had improved diagnostics. The aim of the work presented here is to improve the error recovery scheme which *Yacc* builds into its parsers and thus to improve the error handling in *pcc*.

This paper describes the old method for error recovery in parsers built by *Yacc*, and a new general-purpose method which is independent of source language and which may be used with existing *Yacc* input specifications with minor changes. We present tests on the resulting C compiler which show an improvement in error handling. We assume familiarity with LR parsing as described in [Aho77] for example.

UNIX is a trademark of AT&T

2. *The portable C compiler and Yacc*

In some computing environments, for example a university where many students are learning to use a new language, the quality of error diagnostics produced by a compiler is at least as important as the efficiency of generated code. Students using a UNIX environment who learn C after Pascal often ask why the portable C compiler is so poor compared with the Berkeley Pascal system [Joy80]. A reason for their dissatisfaction is that *pcc* is unable to diagnose many simple syntax errors and produces misleading error messages, whereas the authors of the Berkeley Pascal compiler paid particular attention to developing a good error recovery scheme, presented in [Graham79].

Yacc produces LALR(1) parsers from a set of grammar rules (productions) and actions. The parsers contain default reductions, that is any state of the parser which has a unique reduction in its actions is given that reduction as entry for all symbols which cannot be shifted. To make use of the existing automatic error recovery scheme, described in [Aho74], the productions of the grammar should contain error productions of the form $A \rightarrow \alpha$ **error** β, where A denotes a non-terminal, α,β denote strings of grammar symbols, and **error** denotes the token reserved by *Yacc* for error handling. When the parser is presented with an input token which is not a legal symbol for the current state, it enters error recovery mode and inserts the **error** token on the input. The parser pops states from the stack until the top state is one which can shift **error**. Parsing then continues as dictated by the parse tables, except that any token for which there is no parsing action is deleted from the input. When three input tokens have been shifted, the parser assumes recovery is complete and leaves error recovery mode. In effect the parser assumes that an error has occurred while looking for a derivation of a non-terminal A and that a series of tokens approximating to a derivation of A has been consumed from the input.

Yacc allows the user some control over error recovery actions by permitting error productions to have semantic actions associated with them. These can be used to specify actions to be taken in particular cases. *Yacc* also allows the user to force the parser out of error recovery mode before three tokens have been shifted, and to clear the lookahead token.

The grammar for *pcc* contains eight error productions, one for the external definition construct (the highest-level block of which C programs are composed, that is function and data definitions), five for various forms of declarations and two for the statement construct. Only three of these productions have semantic actions, and these only change local variables. The productions for the statement construct are

statement → **error** ';' | **error** '}'

These productions mean that if the parser detects an error in a statement it will skip all input to the next semi-colon or right curly bracket. All the other error productions have the form

declaration → **error**

These cause the parser to skip input to anything which can follow the declaration. No use is made of the facilities to force the parser out of error recovery mode or clear the lookahead token.

In general, the method for error recovery in *Yacc* has some disadvantages. The user has to write error productions which will control error recovery to an extent which the user may not realise. These productions may introduct ambiguities into the grammar. During recovery, input and stack states are deleted silently. No information about the nature of an error is available. The advantages of the method are that it is simple to implement and efficient to run. In the particular case of *pcc*, the main disadvantage is the poor quality of diagnostic messages, which is a result of the lack of information about errors.

3. *The new method*

The new method for error recovery in parsers generated by *Yacc* uses two techniques, local correction with a forward move, and phrase-level recovery as presented in [Leinius70]. When the parser meets an illegal input token, it first tries to make a local correction to the input string by changes of a single token. If no local correction is successful, where success is judged by the number of moves which can then be made by the parser, a phrase-level recovery is made by replacing a part of the input with a non-terminal. Both already parsed input and input still remaining may be replaced.

3.1 *Local correction*

The set of tokens which are legal shift symbols for the current configuration is determined by the current state. The parser attempts to repair the input by actions in the following order: inserting a token from this set on the input before the next token, deleting the next token, or replacing it with one from the set of legal tokens. In order to determine whether a repair is "good" the parser runs a forward move on the repaired input. This is achieved by copying some of the parse stack onto an error stack, buffering the input and turning off the semantic actions. The parser then restarts from the error state (the state in which it detected error), with the altered input. If the parser can continue to make moves without detecting a further error before five input tokens are shifted, or before accepting, the alteration is taken to be a good repair. The parser is returned to the error state and the parse stack, the input is backed up to the chosen alteration, and semantic actions are turned on again. If an alteration does not allow the parser to run a forward move which consumes five tokens from the input, a forward move is run with the next altered configuration from the set above.

3.2 *Phrase-level recovery*

If no local correction succeeds, the parser is restored to the error state and the input is backed up to the illegal token. The parser chooses a goal non-terminal from the set of kernel items for the current state. Its item has the form

$$A \rightarrow v_1 \cdots v_m \,.\, v_{m+1} \cdots v_n$$

where the v_i are grammar symbols. The phrase to be replaced by the goal non-terminal A is $v_1 \cdots v_n$. $v_1 \cdots v_m$ have been parsed, so the parser pops m states from the stack and pushes the goto state for the new top of stack and A. Further reductions may now take place. To complete the recovery, input is discarded until the next input token is legal for the current state. In effect, a reduction by the production $A \rightarrow v_1 \cdots v_m v_{m+1} \cdots v_n$ has taken place.

A heuristic rule is used to choose the goal phrase from the kernel items of the error state, namely the last item to have been added to the kernel during construction of the item sets, except for the special case of state 0, where the first item is chosen.

The scheme is guaranteed to terminate, because it always consumes input tokens during a successful repair.

3.3 Changes to Yacc input specifications

Error productions are no longer required for error recovery and so may be deleted from grammar rules. The user must supply a routine *yyerrlval* as part of the *Yacc* environment. The purpose of this routine is to supply a default semantic value which is required for tokens inserted during local correction and for non-terminals used as goals for phrase-level recovery. This semantic value will typically be a leaf of the syntax tree, suitably tagged.

3.4 Error messages

The parser synthesises an error message from the recovery action taken in each case. It use the terminal and non-terminal names from the input grammar to *Yacc*. Examples of messages are

SEMICOLON inserted before RIGHTCURLY

for a successful local repair and

e ASSIGN IF NAME replaced by e

for replacement of a phrase.

3.5 Space requirements

Parsers generated by the new *Yacc* require extra space for information for phrase-level recovery and diagnostic messages. No extra space is required for local recovery, as the information required, the valid shift symbols for each state, is present in the existing tables. For phrase-level recovery, two extra words are required for each state, the goal non-terminal and the number of symbols to pop from the stack. Tables of strings are needed for synthesizing diagnostic messages; one string is required for a meaningful name for each grammar symbol, excluding literal tokens.

The parser generated by the new *Yacc* will have fewer states than the equivalent parser generated by the old *Yacc*, because there are no error productions in its grammar. The space-saving device of default reductions for all states with a single reduction is still used.

4. The C compiler

The existing C compiler *pcc* contains a syntax analyzer which is generated by *Yacc*. We took the source of this compiler, removed the error productions from the *Yacc* specifications and included a new function *yyerrlval* which returns a semantic value for inserted tokens and non-terminals. This value is a new leaf of the syntax tree. The only other changes made were to the names of some of the terminals, such as changing SM to SEMICOLON and RC to RIGHTCURLY, to improve the error messages.

The relative sizes of the old and new C compilers are shown in Figure 1.

	pcc	new version
Size in bytes of binary (ccom)	86776	98312
Parser only:		
Number of grammar rules	187	179
Number of states	312	303
Size in chars of source (y.tab.c)	42980	54617

Figure 1. Space required by the C compilers

It is obvious that the compiler performs identically to *pcc* on C programs which are syntactically correct. Error recovery for incorrect programs consists of repairs to the input and error messages. For the new compiler, repairs may be simple changes of one token or replacement of a phrase, and error messages describe the repairs. For the old compiler, error messages do not describe the action taken by the parser to recover, but are either uninformative ("Syntax error") or indicate what the parser finds incorrect.

In order to test the compiler's performance on incorrect programs, we made a collection of all C programs submitted by undergraduate students in the Department of Computer Science at the University of Warwick to *pcc* for compilation over three twenty-four hour periods, October 9, 10 and 16, 1984. Duplicate programs were removed and the programs were run through *pcc* and the new compiler. The code generated for syntactically correct programs was identical. Error recovery was evaluated according to the criteria used by Sippu [Sippu83], rating a correction as excellent if it was the same as a competent programmer might make, good if it introduced no spurious errors and missed no actual errors, fair if it introduced one spurious error or if it missed one error, and poor otherwise, or if the error message generated was meaningless. Missed errors, that is syntax errors that were present in the source code but not reported by the compiler, were counted. Also counted was the number of extra messages, that is messages about errors introduced into the source by incorrect recovery action taken by the compiler. A comparison of the performances of the two compilers, evaluated according to these criteria, is shown in Figure 2. Figure 3 shows a sample C program and its diagnostics.

	pcc		new version	
Quality of recovery action:				
Excellent	1%	(1)	54%	(64)
Good	3%	(3)	11%	(13)
Fair	54%	(64)	13%	(15)
Poor	27%	(32)	19%	(22)
Missed errors	15%	(18)	3%	(4)
Total number of errors	118		118	
Extra messages	127		82	

Figure 2. Comparison of the performance of the C compilers

```
 1   /* Kernighan and Ritchie p. 102 - mutilated */
 2
 3   strcmp(s, t)
 4   char *s, *t
 5   {
 6      for ( ; *s == *t; s++, t++)
 7         if *s == ' '
 8            return(0)!
 9      return(*s - *t);
10   ? }
```

Diagnostics from pcc:
  Line 5: Syntax error
Diagnostics from new version:
  Line 6: SEMICOLON inserted before LCURLY
  Line 7: LPAREN inserted before MUL
  Line 8: RPAREN inserted before RETURN
  Line 9: UNOP replaced by SEMICOLON
  Line 11: QUEST deleted

Figure 3. A sample C program

5. *Discussion*

The C compiler generated by the new *Yacc* performed better on the collection of incorrect programs than *pcc*. The majority of the errors were simple ones which occurred sparsely, and were therefore amenable to repair by the local recovery tactic. This pattern of occurrence of simple errors concurs with Ripley and Druseikis' analysis of syntax errors in Pascal programs [Ripley78], which showed that the majority of these are single-token errors and occur infrequently. Clusters of errors and complicated errors were not handled so well by the phrase-level recovery, and these were responsible for the large number of extra messages generated.

The diagnostic messages produced depend on the names for the terminals and non-terminals, which should be carefully chosen by the grammar-writer. Ideally, messages should be at source level rather than lexical token level, as the user will understand a message of the form

```
Line 16:          x = y
Semi-colon inserted ...... |
```

better than one of the form

Line 16: SEMICOLON inserted after ID

More communication between the lexical analyzer and the parser may be needed for this sort of message. Line numbers at present are occasionally out by one because of buffering of the lexical tokens.

A disadvantage is that the scheme shows bias towards assuming correctness of the left context. Local recovery assumes a single error in the current input token, and secondary recovery makes an arbitrary choice of item from the error state which takes no account of the right context.

Several other error recovery schemes for LR(1) parsers have been described. [Sippu81] and [Dain84] contain recent reviews of the literature. The scheme presented here bears resemblance to that devised by Graham, Haley and Joy for a Pascal compiler [Graham79], in that a two-stage recovery is attempted. There are several differences to note. Firstly, our scheme is a general-purpose recovery scheme incorporated in a new version of *Yacc*, and is used by any parser generated by the new *Yacc*. Graham requires special purpose error recovery routines and cost vectors to be supplied for use by their parser generator *Eyacc* which contains no error recovery scheme. Secondly, *Eyacc* produces parsers with certain states calling for reductions having their lookahead tokens enumerated, i.e. some default reductions are not made. Our *Yacc* has the usual default reductions. Thirdly, Graham requires the user to supply error productions in the grammar, to control secondary recovery. These are not required for our scheme.

The error recovery scheme for the compiler-writing system HLP [Raiha83] incorporates a local recovery tactic into the phrase-level recovery scheme [Sippu83]. No forward move is made on the input and there is less check on the "correctness" of a local correction; the user must supply costs for deletion and insertion of each terminal in local correction. Different criteria are used for identifying and replacing the error phrase in phrase-level recovery.

A two-stage recovery scheme for LL(1) and LALR(1) parsers which uses the concept of *scope recovery* is implemented by Burke and Fisher [Burke82]. The scheme cannot be used however in LR parsers with default reductions. The user must supply additional language information such as constructs which open and close scope in the language, lists of tokens which cannot be inserted between a given pair of tokens, and lists of tokens which cannot be substituted for a given token. Pai and Kieburtz [Pai80] use *fiducial* (trustworthy) symbols, typically reserved words, in a scheme for LL(1) parsers which they suggest as suitable for extending to LR parsers.

Requiring the user of a parser-generator to supply information to aid error recovery in addition to the grammar may result in recovery which is more tailored to the language, but imposes an extra burden on the user, who may not have a full understanding of the mechanism of the parser and its error handling. The scheme which we have implemented in *Yacc* makes few demands of this nature on its users, yet improves the quality of error recovery in its parsers.

6. *References*

[Aho74] Aho, A. V. and Johnson, S. C. LR Parsing. *Computing Surveys 6,* 2 (June 1974), 99-124.

[Aho77] Aho, A. V. and Ullman, J. D. *Principles of Compiler Design.* Addison Wesley, 1977.

[Burke82] Burke, M. and Fisher, G. A. A practical method for syntactic error diagnosis and recovery. *ACM SIGPLAN Notices 17, 6* (June 1982), 67-78.

[Dain84] Dain, J. A. Error recovery schemes in LR parsers. Theory of Computation Report No. 71, Dept. of Computer Science, University of Warwick, Coventry, December 1984.

[Graham79] Graham, S. L., Haley, C. B. and Joy, W. N. Practical LR error recovery. *ACM SIGPLAN Notices 14,* 8 (August 1979), 168-175.

[Johnson78a] Johnson, S. C. Yacc - Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, New Jersey, 1978.

[Johnson78b] Johnson, S. C. A portable compiler: theory and practice. *Proc. 5th ACM Symp. on Principles of Programming Languages* (January 1978), 97-104.

[Joy80] Joy, W. N., Graham, S. L. and Haley, C. B. Berkeley Pascal User's Manual. Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1980.

[Leinius70] Leinius, R. P. Error detection and recovery for syntax directed compiler systems. Ph. D. Th., Computer Science Dept., University of Wisconsin, Madison, 1970.

[Pai80] Pai, A. B. and Kieburtz, R. B. Global context recovery: a new strategy for parser recovery from syntax errors. *ACM Trans. on Programming Languages and Systems 2*, 1 (January 1980), 18-41.

[Raiha83] Raiha, K-J., Saarinen, M., Sarjakoski, M., Sippu, S., Soisalon-Soininen, E. and Tienari, M. Revised Report on the Compiler Writing System HLP78. Report A-1983-1, Dept. of Computer Science, University of Helsinki, Finland, January 1983.

[Ripley78] Ripley, G. D. and Druseikis, F. A statistical analysis of syntax errors. *Computer Languages 3*, 4 (1978), 227-240.

[Sippu81] Sippu, S. Syntax Error Handling in Compilers. Report A-1981-1, Dept. of Computer Science, University of Helsinki, Finland, March 1981.

[Sippu83] Sippu, S. and Soisalon-Soininen, E. A syntax-error-handling technique and its experimental analysis. *ACM Trans. on Programming Languages and Systems 5*, 4 (October 1983), 656-679.

A PROLOG description of a symmetric solution for automatic error-recovery in LL(1) and LALR(1) parser generators.

Theo de Ridder

HIO department
IHBO "de Maere"
7500 BB Enschede
Netherlands

(ridder@im60.UUCP)

ABSTRACT

It is shown that the LL(1) and LALR(1) parsing schemes are too close to each other to justify the quality gap between their respective error-handling capabilities. Even within the constraints of a one-symbol-lookahead strategy a reasonable error-recovery with clear and precise syntax-derived messages is possible without adding any explicit error information.

A considerable improvement is obtained by synchronizing on so-called fiducial symbols. PROLOG was suitable to find a formal specification of the set of fiducials for any grammar. The fiducials optimize error-handling without disturbing the efficiency and robustness of the underlying parsing scheme. They are implemented by context parameters in LL(1) procedures and by automatic insertion of error tokens in LALR(1) grammars.

Available syntax descriptions in YACC of ADA and AWK are used to demonstrate the effectiveness of the method.

September 4, 1985

A PROLOG description of a symmetric solution for automatic error-recovery in LL(1) and LALR(1) parser generators.

Theo de Ridder

HIO department
IHBO "de Maere"
7500 BB Enschede
Netherlands

(ridder@im60.UUCP)

1. Introduction.

Programming languages contain a fundamental duality. As communication vehicles they give many usergroups a culture and an identity. But above the tower of Babel the glimmering of a universal and unified language from a mathematical world never vanishes. It is interesting that while a certain language is advocated for a very specific domain, its real power is due to some general purpose aspect. A classic example is Simula-67, enabling object-oriented programming 'avant la lettre'.

The techniques of parser generators are not limited to the field of compiler writing. We introduce the concept 'pattern-directed' programming to indicate the simularity within a wide range of languages. A pattern is an abstraction for a number of internal states. Examples are algebraic expresssions (APL), functions (FP), strings (SNOBOL), templates (PROLOG, SASL), messages (SMALLTALK), regular expressions (AWK, LEX), and context-free grammars (YACC). The absence of explicit declarations is considered as another aspect of the pattern-directed programming style. The compactness of a program representation is mainly determined by the capabilities of patterns, because the syntactic redundancy of explicit control structures is practically the same for the different languages.

When patterns are used to hide internal states, it is quite unpleasant to be pushed back to lower levels in order to handle input errors. Automatic error-recovery should be regarded as an essential feature of pattern-directed facilities. In this article we will restrict ourselves to the domain of LL(1) and LALR(1) grammars, where a solid theoretical base and efficient implementations are available. Our contribution to automatic error-recovery consists of an effective combination of well-known mechanisms [1,2,3,4,5,6,7] into a single framework.

Using PROLOG as description language was an experiment, resulting in a short and elegant representation of the algorithms. It might be visible it was the first PROLOG experience of the author, but the presence of global variables is part of minimizing the distance with traditional compiler implementations.

With an emphasis on the last three points we have tried to fullfill the following basic requirements of automatic syntactic error-recovery:

- robustness.
  No input sequence should be unrecoverable.

- minimal distance.
  The recovered text should be close to the minimal distance change.

- spurious errors.
  Recovery should not introduce spurious errors into correct input fragments.

- time and space complexity.
  The recovery algorithm should have the same time and space complexity as the given parsing method.

- error-free parsing.
  Recovery should not degrade the performance of parsing error-free input.

- error messages.
  Error messages should be clear, precise and grammar-derived. A single error should not cascade into redundant messages.

- transparancy.
  The recovery mechanism should be understandable and predictable.

- interfaces.
  There should be a flexible interface between error-handling and the lexical, syntactic and semantic analysis.

Three languages (PICO, AWK, ADA are used to demonstrate the effectiveness of our approach. With the small hypothetical PICO the basic mechanisms are explained. Awk is a typical YACC based UNIX product with terrible error messages in its original shape. ADA was interesting to validate the method for a very large grammar.

2. A meta-parser in PROLOG.

In order to tune and test different error-recovery schemes a complete parser generator has been created in PROLOG.

The LL(1) and LALR(1) parsers share the same meta-parser frontend. This frontend transforms an EBNF syntax specification into a canonical database representation of the rules and the vocabular:

```
rulename(Name).
token(Symbol).
rule(Nr, Name, Body).
```

A simple meta-scanner recognizes layout characters, end-of-file, identifiers, numbers and constants, and keeps track of token positions. It constitutes the base for an automatic scanner that reads a token into the database as:

```
nexts(Symbol).
```

The LL(1) system is completely written in PROLOG. To transform a syntax description into a LALR(1) action- and goto-table YACC and AWK are involved too:

step 1) ebnf -> prolog -> canonized rules.

step 2) canonized rule -> prolog -> syntax + error tokens.

step 3) syntax + error tokens -> YACC -> y.output

step 4) y.output -> AWK -> action + goto table.

The action- and goto-table representation in PROLOG is quite simple:

```
action(State,Token,Action).
goto(State1,Rulename,State2).
```

The meta-parser is not shown in the appendix. It does not contain anything special. We use it as an excercise for our students.

3. The LL(1) parsing scheme.

A standard implementation of the LL(1) method is a recursive descent parser. The fact that it is deterministic, without the need for backtracking, makes it efficient but also problematic for error-recovery. However there is an heuristic way to make errors more or less deterministic too by passing 'Context' as parameter of each syntactic procedure. The idea is not to skip any token in any recursion level when it is present in 'Context'. An obvious choice for 'Context' is including at each level the set of valid tokens following the current symbol. For example {FI, ELSE} is added when <stat> is parsed within:

```
ifstat : IF expr THEN stat L19 FI ;
```

If the first token of a rule is missing a LL(1) parser will not parse that rule at all. This shortcoming can be resolved by introducing fiducials. A fiducial symbol is defined by a unique occurrence in just one of the rules. Some experiments revealed that a reasonable set of fiducial tokens identifying a rule is formed by the fiducials occurring in the rule itself combined with the ones found by propagating recursively the first nonempty nonterminal symbol. A fiducial nonterminal symbol is replaced by its body with the consequence that the quality of the recovery is independent of special features like meta-brackets in the original meta-syntax. The resulting set of fiducials for <ifstat> is:

```
{ IF, THEN, ELSE, FI }
```

This set however may bring the parser in a loop. When somewhere within <stat> the token 'ELSE' or 'FI' is encountered, <ifstat> will be called recursively without ever meeting that token. The explanation is that by using a fiducial to start a rule left-recursion is introduced again. A solution is to consider a token as non-fiducial if it becomes valid in a rulebody somewhere after a symbol of which it was a fiducial. Now the final set for <ifstat> becomes:

```
{ IF, THEN }
```

Another improvement is further to equal the effect of a rule terminating into a single nonfiducial token with that of an empty rule. This causes the fiducial set for

```
L10 :   ';' stats ;
```

to be

```
{ 'DO', '=', 'THEN', 'IF', 'PRINT'}
```

in stead of the empty set.

During parsing all the work for error-handling is done only by:

```
match_nexts(Symbol,Starters,Context).
```

'Starters' contains the valids and fiducials of 'Symbol'. Match_nexts will skip an input token unless it belongs to 'Starters' or 'Context', in which case it will return with true or fail respectively. The two possible error-messages within match_nexts reveal that in fact the recovery actions are restricted to inserting and skipping a token.

The recursive descent procedures are represented by:

```
ll_parse(Rulename, Context).
```

As actual parameter 'Context' is not only filled with the next valids but also with the fiducials of 'Rulename' itself. This improved the behaviour for repetitive syntactical constructs considerable.

4. The LALR(1) parsing scheme.

YACC is based on the LALR(1) method, and produces from a rather primitive meta-syntax a table-driven parser in C. Much of its work is involved in optimizing the space-time characteristics of a sparse matrix implementation. This has the advantage that even for very large grammars parsing is still efficient.

One of our basic decisions has been to change nothing in the meta-syntax or tables. Only the grammar independent parse routine is rewritten. Like before this routine consists out of a state-transition loop with only three valid transitions:

- accept.
  Input, as specified by the syntax, is completely parsed.

- shift.
  Push the current input-token and scan a new one.

- reduce.
  Pop a rule from the parse-stack, but only if the current input-token will be become valid in forward parsing direction.

- error recovery.
  Try after each other the following alternatives to construct a valid transition for the current input-token:

  - insert a missing token.

  Go backwards on the parsestack until a state enables the current input-token to:

  - synchronize with the error-token.

  - reduce an incomplete rule.

- error panic mode.
  Skip the current input-token if not end-of-file.

One has to notice that preventing reductions before errors without changing

tables implies a certain degradation of error-free parsing.

The already available error-token has been used to introduce fiducial tokens. The error-token is automatically inserted in the grammar before fiducials. The definition of fiducials is changed compared with the LL(1) case, due the properties of LALR(1) tables and the loss of certain grammar information. A rulename now is defined fiducial if each of its alternatives start with a fiducial token. New rules are added in which the error-token replaces a nonempty left rulepart before a fiducial, followed by a nonempty remaining rulepart. Another condition is that such a fiducial may not cause a valid shift in the replaced rulepart.

To restrict the number of added states the error-token is not inserted before each rule starting with a fiducial, but only a fiducial rulename generates such an extra alternative. The given recept will append the following new rules to <ifstat>:

```
ifstat : error THEN stat L19 FI ;
ifstat : error ifstat ;
```

5. Results.

Appendix 1 gives the PICO syntax in YACC with the addition of generated error rules. It is not written left-recursive to use it in the same shape for LL(1).

Appendix 2 shows the LL(1) and LALR(1) error-messages together for a very artificial but illustrative PICO program.

In Appendix 3 the old and new error-messages are compared for an AWK program [8], using a new version of YACC with symbolic state information and the new parseroutine, transformed from PROLOG into C. Because no changes were made to the AWK sources only the linenumber was available for position indication.

Appendix 4 is the result for a small ADA program [9], using a complete YACC/LEX parser from the public domain[10].

In Appendix 5 some (VAX) statistics are given for a complete syntax-directed screen-editor for ADA. This editor was generated with our LYSE [11] system, which was basically inspired by the described error-recovery implementation.

The relevant PROLOG sources are given in appendix 6. It should be noted that the basic primitives are not optimized for execution. Parsing is made efficient by putting first all the static sets (like Starters and Contexts) into the database.

6. Conclusions.

The effectiveness and simplicity of the given strategy for automatic syntactic error-recovery prove that other methods often impose an unnecessary complexity. Of course one should not forget the inherent limitations of the one-symbol-lookahead approach. Certain ambiguities could have been resolved taking more input tokens into account. However a very small average improvement would not justify the violation of at least three of our basic requirements.

The symmetry between LL(1) and LALR(1) can be paraphrased with: in the first case one tries to forsee the future and in the second to update the history. The context parameters are the counterpart of look-ahead sets of pushed states. The definitions of fiducials are different but the idea and result are almost

the same.

There remains also a fundamental difference between LL(1) and LALR(1), because LL(1) will never come back to a point already parsed before, while LALR(1) may recover from any state not yet popped from the stack.

The use of PROLOG contributed substantially in formalizing and validating certain intuitive insights about practical error-handling.

References.

[1] Michunas, M.D., and Modry, J.A.,
Automatic error recovery for LR parsers,
Comm. ACM, 21,6(June 1978)459-465.

[2] Graham, S.L., Haley, C.B., and Joy, W.N.,
Practical LR error recovery,
Proc. SIGPLAN Symp. Compiler Constr., SIGPLAN Notices 14,8(Aug 1979)168,175.

[3] Pemberton, S.,
Comments on an error-recovery scheme by Hartmann,
Software-Practice and Experience, 10(1980)231-240.

[4] Pai, A.B., and Kieburtz, R.B.,
Global context recovery: a new strategy for syntactic error recovery by table-driven parsers,
ACM Trans. Progr. Lang. Syst. 2,1(Jan 1980)18-41.

[5] Florijn, G., and Rolf, G.,
Pgen - a general purpose parser generator,
IW 157/81, Mathematical Centre, Amsterdam, 1981.

[6] Burke, M., and Fisher, G.A.,
A practical method for syntactic error diagnosis and recovery. Proc. SIGPLAN Symp. Compiler Constr., SIGPLAN Notices 17,6(june 1982)67-78.

[7] Lewi, J., De Vlaminck, K., Huens, J., and Steegmans, E.,
A programming methodology in compiler construction, part 2: implementation,
North Holland, 1982.

[8] Habermann, A.N., and Perry, D.E.,
ADA for experienced programmers,
Addison-Wesley, 1983.

[9] Fischer, H.,
A LALR(1) grammar for ANSI ADA,
HFischer@eclb.arpa, 1984.

[10] Bentley, J.,
Programming pearls,
Comm. ACM, 28,6(june 1985)570-576.

[11] Ridder de, Th.F., Florijn, G.,
Automatic generation of syntax directed screen editors,
In Proc. EUUG spring meeting, Nijmegen, 1984.

Appendix 1.

```
/*
 * rulename L10,L19,L23,L26 are generated
 * from another meta-syntax
*/

%%
pico:     BEGIN decls stats END ;
decls:
    |     decl ',' decls ;
decl:     INT IDENT init
    |     BOOL IDENT init ;
init:
    |     ':' expr ;
L10:
    |     ',' stats ;
stats:    stat L10 ;
stat:     compstat
    |     assstat
    |     ifstat
    |     prinstat ;
compstat: DO stats OD ;
assstat:  IDENT '=' expr ;
L19:
    |     ELSE stat ;
ifstat:   IF expr THEN stat L19 FI ;
prinstat: PRINT expr ;
L23:
    |     '+' expr ;
expr:     term L23 ;
L26:
    |     '*' term ;
term:     factor L26 ;
factor:   IDENT
    |     CONSTANT
    |     NUMBER
    |     '(' expr ')';

/* added to introduce LALR(1) fiducials */
pico:     error pico ;
decl:     error decl ;
compstat: error compstat ;
assstat:  error '=' expr ;
ifstat:   error ifstat ;
ifstat:   error THEN stat L19 FI ;
prinstat: error prinstat ;
%%
```

Appendix 2.

```
 1 BEGIN
 2      INT 101;
 3      INT a 2 * 5;
 4      INT b c : (4 + 5;
 5      BOOL d;
 6      DO
 7           THEN a * 6
 8           PRINT     d = ('3'+ );
 9           IF b
10           THEN if c
11                THEN PRINT 23
12                FI
13           ELSE PRINT
14           FI
15      OD
16      PRINT a b;
17 END
```

Position	LL(1)	LALR(1)
[2,13]	skipped: NUMBER(101)	skipped: NUMBER(101)
[2,16]	missing: IDENT	incomplete: decl
[3,15]	skipped: NUMBER(2)	missing: ':'
[3,17]	skipped: '*'	incomplete: factor
[3,19]	skipped: NUMBER(5)	
[4,15]	missing: ';'	missing: ':'
[4,17]	skipped: ':'	skipped: ':'
[4,19]	missing: '='	skipped: '('
[4,20]		skipped: NUMBER(4)
[4,25]	missing: ')'	
[5,9]	skipped: BOOL	
[5,15]	missing: '='	
	missing: expr	
[7,17]	missing: IF	synchronized: THEN
	missing: expr	
[7,27]	skipped: '*'	skipped: '*'
[7,29]	missing: '='	missing: '='
[8,17]	missing: ';'	synchronized: PRINT
	missing: FI	
[8,27]	missing: ';'	synchronized: '='
	missing: IDENT	
[8,35]	missing: expr	incomplete: expr
[8,36]		incomplete: stat
[10,25]	missing: '='	missing: '='
[11,25]	missing: ';'	synchronized: THEN
	missing: FI	
	missing: IF	
	missing: expr	
[12,25]		skipped: FI
[13,17]	skipped: ELSE	incomplete: stat
[13,25]	missing: ';'	
[14,17]	skipped: FI	incomplete: L19
[15,9]	missing: expr	incomplete: stats
[16,9]	missing: ';'	synchronized: PRINT
[16,17]	skipped: IDENT(b)	incomplete: decls
[16,18]		incomplete: stat
[17,1]	missing: stats	incomplete: stats

Appendix 3.

```
 1  {
 2      ++predct[$2]
 3      predct[$1] = predct[$1]
 4      succlist[$1 "," ++succcnt[$1] = $2
 5  }
 6 END {
 7      qlo = 1
 8      for (i in predct {
 9          n++ if (predct[i] == 0) q[++qhi] = i
10      }
11      while (qlo <= qhi) {
12          t = q[qlo++]; print t
13          for (i=1; i<=succcnt[t];i++) {
14              s = succlist[t "," i]
15              if (--predct[s] == 0) q[++qhi] = s
16          }
17      }
18      if (qhi != n) print ,"tsort error: cycle in input"
19 }

awk:    syntax error near line 4
awk:    illegal statement near line 4
awk:    syntax error near line 8
awk:    illegal statement near line 8
awk:    illegal statement near line 11
awk:    illegal statement near line 12
awk:    syntax error near line 18
awk:    bailing out near line 18

 4:     SPRINTF, SPLIT, ..., or ']' expected in <var>;
        <var> incomplete reduced before ASGNOP.
 5:     SPRINTF, SPLIT, ..., or ']' expected in <var>;
        <simple_stat> incomplete reduced before NL.
 8:     ')' expected in <for>;
        <for> synchronized before '{'.
 9:     NL or ';' expected in <statement>;
        <statement> synchronized before IF.
18:     NL or ';' expected in <statement>;
        <pe_list> incomplete reduced before ','.
```

Appendix 4.

```
Starting Ada grammatical analysis

[1]
[2]  generic type labeltype is private;
[3]       with function "<" (l1,l2: in labeltype)
[4]            return boolean is <>
[5]  package LabeledBinaryTree is
[6]       type binarytreenode;
[7]       type binarytree is access binarytreenode;
[8]       type binarytreenode is record
[9]            label: labeltype;
[10]           left,right: binarytree;
[11]      end record;
[12]      procedure INSERT (label: in labeltype;
[13]                  root: in out binarytree;
[14]                  node: out binarytree;
[15] end LabeledBinaryTree;
[16]
[17] ---> Grammatical analysis complete!!
          0 lex error(s), 3 yacc error(s) <---

   3:     ';' expected in <object_d>;
     <prm_spec> incomplete reduced before ')'.
   5:     ';' expected in <gen_prm_d>;
     <gen_prm_d> synchronized before PACKAGE_.
  15:     identifier expected in <.._.prm_spec..>;
     <.PRIVATE..basic_decl_item...> incomplete reduced before END_.
```

Appendix 5.

```
95/127 terminals, 238/300 nonterminals
459/600 grammar rules, 860/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
238/350 working sets uqed
memory: states,etc. 4125/12000, parser 3113/12000
601/800 distinct lookahead sets
946 extra closures
1235 shift entries, 65 exceptions
571 goto entries
1414 entries saved by goto default
Optimizer space used: input 3376/12000, output 1147/12000
1147 table entries, 0 zero
maximum spread: 333, maximum offset: 857

size  ada.parser

text data bss  dec  hex
13312     76800    18332    108444    1a79c

size  ada.lyse

text data bss  dec  hex
61440     121856   28904    212200    33ce8
```

Appendix 6.

```
% ---------- general predicates ----------------

% find all valid starters of a rule
first(Token,Rulename).
     first(Token,Rulename,[]).
first(Token,Name,Hist) :-
     rule(_,Name,L),
     not(member(Name,Hist)),
     valid(Token,L,[Name|Hist]).

% find all valid starters of a list
valid(Token,List).
     valid(Token,List,[]).
valid('$empty',[],_) :- !.
valid(T,[T|_],_) :-
     token(T), !.
valid(T,[X|L],Hist) :-
     first(T,X,Hist), T = '$empty'.
valid(T,[X|L],Hist) :-
     valid('$empty',X,Hist), !, valid(T,L,[X|Hist]).

fiducial('$empty') :-
     !, fail.
fiducial(X) :-
     once((rule(Nr1,_,L1), member(X,L1))),
     rule(Nr2,_,L2), Nr1 = Nr2, member(X,L2),
     !, fail.
fiducial(_).

% find all non-list members recursively
member(X,[X|_]) :-
     X = [_|_].
member(X,[[Y|L]|_]) :-
     member(X, [Y|L]).
member(X,[_|L]) :-
     member(X,L).

% split a list as a for-loop
decompose(L,[],L).
decompose([X|L],[X|L1],L2) :-
     decompose(L,L1,L2).

% ---------- specific LL(1) predicates ---------

ll_fiducial(Token,Nr) :-
     rule(Nr,_,L),
     ll_fiducial(Token,L,[]).
ll_fiducial(Token,L,Hist) :-
     decompose(L,[Y|L1], [X|_]),
     once((token(X); fiducial(X);
       valid('$empty',[Y|L1]);
       single_nonfid([Y|L1]))),
     ll_fiducial(Token,[X],[Y,L1|Hist]).
ll_fiducial(Token,[X|_],Hist) :-
     not(member(X,Hist)),
     (token(X)-> (fiducial(X), T=X);
       (rule(_,X,L), ll_fiducial(Token,L,[X|H]))).
```

```
single_nonfid(List) :-
     /* terminates List to a single non-fiducial token? */.
ll_sets :-
     /* calculates all needed static sets
        (fiducials, valids) before parsing */.

% find looping fiducials
ll_nonfiduc(T) :-
     rule(Nr,_,L), fiducials(Nr,FØ),
     decompose(L,L1,L2), member(T,FØ),
     member(R,L1), fiducials(R,F1), member(T,F1),
     member(X,L2), valid([X],T).

% ---------- LL(1) parser ----------------------

ll_parse(Rulename) :-
     scan, valids(Rulename,V),
     parsebody([Rulename],[V],['$eof']).

% only called after a valid or fiducial token
ll_parse(Token,_) :-
     token(Token), !, scan.
ll_parse(Rulename,Context) :-
     rule(Nr,Rulename,L), valids(Nr,[V1|V]),
     nexts(Token), member(Token,V1), !,
     parsebody(L, [V1|V], Context).
ll_parse(Rulename,Context) :-
     rule(Nr,Rulename,L), fiducials(Nr,F),
     nexts(Token), member(Token,F), !,
     valids(Nr,V), parsebody(L,V,Context).
ll_parse(Rulename,_) :-
     empty(Rulename).

parsebody([],[],_) :- !.
parsebody([X], [V], Context) :-
     (fiducials(X,F) -> true; F = []),
     match_nexts(X,[V|F],Context), !,
     ll_parse(X,[F|Context]).
parsebody([X|L], [V1,V2|V], Context) :-
     (fiducials(X,F) -> true; F = []),
     match_nexts(X,[V1|F],[V2|Context]),
     ll_parse(X,[V2,F|Context]), fail.
parsebody([_|L], [_|V], Context) :-
     parsebody(L,V,Context).

% error recovery procedure
match_nexts(X,_,_) :-
     nexts(X), !.
match_nexts(X,Starters,_) :-
     rulename(X),
     nexts(T), member(T,Starters), !.
match_nexts(X,_,Context) :-
     empty(X),
     nexts(T), member(T, Context), !.
match_nexts(X,_,Context) :-
     nexts(T), member(T,Context),
     perror("missing", X), !, fail.
match_nexts(X,Starters,Context) :-
     nexts(T), perror("skipped", T),
     scan, match_nexts(X,Starters,Context).
```

```
% ---------- specific LALR(1) predicates -------

lalr_sets :-
     /* calculates all needed static sets
        (fiducials, valids) before parsing */.

% generate error-token rulebody tails
lalr_newrulebody([X|L1],L2) :-
     valids(X,V), subtract(V,['$empty'],[]), !,
     lalr_newrulebody(L1,L2).
lalr_newrulebody([X|L1],L2) :-
     valids(X,V), lalr_newrulebody(L1,L2,V).
lalr_newrulebody([X|L],[X|L],Hist) :-
     fiducials(F), member(X,F),
     valids(X,V), subtract(Hist,V,Hist),
     once((member(Y,L), valids(Y,Z),
          not(member('$empty',Z)))).
lalr_newrulebody([X|L1],L2,Hist) :-
     valids(X,V), lalr_newrulebody(L1,L2,[V|H]).

% ---------- LALR(1) parse loop ----------------

lalr_parse :-
     scan, push(0), lalr_loop.
lalr_loop :-
     nexts(Token), stack([State|_]),
     do_action(State,Token,Z), Z == stop, !.
lalr_loop :-
     lalr_loop.

do_action(State,Token,stop) :-
     action(State,Token,accept), !.
do_action(State1,Token,shift) :-
     action(State1,Token,shift(State2)), !,
     push(State2), scan.
do_action(_,Token,reduce) :-
     stack(L1), do_reduce(Token,L1,L2),
     forward(Token,L2), !,
     renew(stack(L2)).
do_action(State1,Token1,error) :-
     action(State1,Token2,shift(State2)),
     Token2 = 'error',
     action(State2,Token1,shift(_)), !,
     push(State2), perror("missing", Token2).
do_action(_,Token,error) :-
     stack(L1), backward(Token,L1,L2), !,
     renew(stack(L2)).
do_action(State,'$eof',stop) :- !,
     perror("not able to skip", '$eof').
do_action(_,Token,error) :-
     perror("skipped", Token), scan.

do_reduce(Token, [State1|L1], [State3,State2|L2]) :-
     action(State1, Token, reduce(M)), !,
     rule(M,R,B), poplist(B, [State1|L1], [State2|L2]),
     goto(State2,R,State3).
do_reduce('$.', _, _) :-
     !, fail.
do_reduce(_, L1, L2) :-
     do_reduce('$.', L1, L2).
```

```
% forward stack movement
forward(Token,[State|_]) :-
     action(State, Token, shift(_)), !.
forward(Token,[State|_]) :-
     action(State, Token, accept), !.
forward(Token,L1) :-
     do_reduce(Token,L1,L2),
     forward(Token,L2).

% backward stack movement
backward(_, [], _) :-
     !, fail.
backward(Token, [State1|L], [State2,State1|L]) :-
     action(State1, 'error', shift(State2)),
     action(State2, Token, shift(_)), !,
     perror("synchronized", Token).
backward(Token, [State1|L], [State2,State1|L]) :-
     goto(State1, Rulename, State2),
     action(State2, Token, shift(_)), !,
     perror("incomplete", Rulename).
backward(Token, [_|L1], L2) :-
     backward(Token, L1, L2).

stack([]).
push(N) :-
     stack(L), renew(stack([N|L])).

poplist([], L, L) :- !.
poplist([_|L1], [_|L2], L3) :-
     poplist(L1,L2,L3).
```

Screen Based History Substitution for the Shell

Mike Burrows
Cambridge University Computer Laboratory

Several UNIX † command interpreters now incorporate a history mechanism to assist interactive users in repeating or correcting commands. Many of these are similar in style to Bill Joy's C Shell, which provides a simple line oriented interface and a numbered history list. More advanced shells, such as the Korn Shell, allow screen editor facilities, but still reference history items by an event number or an explicit pattern matching syntax. This paper describes an interface that allows history substitutions to be performed as a command is typed and with minimal user effort. Entire lines or single words may be substituted with equal facility. The technique has been fully integrated with more normal editing features and Tenex-style filename completion.

The interface is extremely simple to use even with large history lists, without requiring the user to repeat "event numbers" or to review the history list periodically. Versions of the interface have been added to various existing shells and have been in use for several months, proving popular with both novices and experienced users. The latest version has been implemented as one of a number of enhancements to the System 5.2 Shell.

1. Introduction

Over the past few years there has been much interest in improving the user interfaces of interactive command interpreters (shells) under UNIX. Despite the widespread use of screen editors, most users are still tied to the conventional line-oriented interface of the Bourne Shell(ref.1) released with Version 7 UNIX. One of the most significant improvements was the introduction of history substitution in Bill Joy's C Shell.(ref.2) More recently, command line editing has been added to shells such as the Edit Shell(ref.3) and the Korn Shell.(ref.4)

Although the C Shell includes many features that make it particularly suitable for interactive use, it has remained unpopular except at sites which require support for Berkeley job control. Even then, many C Shell users prefer the Bourne Shell when writing scripts. Some of the more obvious deficiencies in the C Shell are:

- The C Shell cannot run standard Bourne Shell scripts.
- The command syntax is considered by many‡ to be inferior to that of the Bourne Shell.

---

† UNIX is a Trademark of AT&T Bell Laboratories
‡ Including the present author

- C Shell scripts run more slowly than Bourne Shell scripts.

- The history mechanism is line based; its syntax is complicated and obscure.

- C Shell control constructs are not properly integrated with the history mechanism and with job control.

These problems are not shared by the Korn Shell, which offers many advantages over the Bourne Shell, including the most useful features of the C Shell. Unfortunately, its popularity has been limited by problems of availability and cost. Although a great improvement over most other shells, the Korn Shell does leave some room for improvement:

- It retains a fairly simple history interface which treats the *command* as its primary unit. Individual arguments can be manipulated in various ways, but they cannot be recalled and inserted with the same facility as complete lines.

- The editor is constrained by the decision to emulate the styles of both Vi(ref.5) and Emacs(ref.6) without using the termcap/terminfo databases. This does not allow the shell to automatically reconfigure itself to use special keys available on different terminals.

The remainder of this paper describes Msh, a shell which incorporates a command line editor, a novel interface for history substitution and many other enhancements. When this project began the author used the Bourne Shell exclusively, so early versions of Msh were based on the Bourne Shell. Later versions have been integrated with the System 5 release 2 Shell and incorporate many C Shell features, including C Shell style job control.

The original aim of the project was to provide a measure of uniformity between the interfaces presented by the screen editor and the shell. Experience gained from early versions led to the investigation of history mechanisms, and ways of integrating them with a command line editor. The style of history substitution has evolved as the author gained more experience. The overall trend was towards simplicity, which eventually led to the use of a single key for all common history operations. The following sections describe the features of Msh, compare them with analogous other shells and explain some of the decisions made in the design.

2. *The Msh Editor*

The Msh editor provides the user with a one line window in which a command may be typed and edited before being executed by the shell. The single line display greatly simplifies the structure of the editor, without causing major inconvenience. Long lines are displayed by scrolling the text horizontally whenever the cursor would move off the edge of the screen.

The main design aims for the editor were simplicity and upward compatability with the standard shell command line. To avoid confusion, printable characters are always entered as text at the current cursor position. Control characters and special purpose keys are used for cursor positioning and editing functions. The user's erase and kill characters are preserved so that normal shell usage is unchanged.

The most important differences between the Msh editor and the editors of the Edit Shell and the Korn Shell are:

- Msh provides only one style of editing, which bears little resemblance to Vi† or Emacs. Unlike Vi, Msh has no concept of "text input" and "cursor motion" modes.

- Msh can make full use of the keys available on a particular terminal, such as the cursor motion keys. All the operations accessible through function keys are also available as control keys.

- Most common operations are available as single keystrokes. For example, "backword" and "delete previous word" are both single keystrokes.

- Multiple keystroke commands are built up from logical combinations of other keys. For example, one key signifies "more"; **more left** moves the cursor to the beginning of the line and **more right** moves it to the end of the line.

- Msh uses the termcap and terminfo databases to obtain information about the current terminal. This allows better use of terminal capabilities at the expense of slightly increased startup times for interactive shells.

Although Msh was designed with simplicity in mind, some quite complex functions have been provided, such as the ability to undo edits and to enter text repeatedly. All such operations have been provided in a manner which is convenient for the experienced user, but which does not affect the novice. A complete list of keyboard functions is given in the appendix.

3. The History Mechanism

3.1 The History List

The shell maintains a list of the last few lines that have been submitted as input. The number of lines retained in the list can be specified by the user, but the total space occupied by the history list is limited by the shell. In an effort to save space, the shell does not save identical copies of a single line. If a line is reused, it is appended to the history list, and the previous instance of the line is deleted. This behaviour does not allow the history list to be used as a full record of commands typed by the user, but it assists in implementing history substitution.

As in Ksh, the editor window can be moved up and down the history list, showing one line from the list at a time. In this way, lines from the history list can be edited and executed as though they had been retyped. Although this technique is occasionally useful, the most common method of accessing the history list is through the history substitution mechanism described below.

3.2 History Substitution

Several experimental user interfaces have been tested with Msh during its development. The trend has always been towards a simpler interface, without event identifiers and without complex key sequences. Eventually, a single key system was developed in which the shell attempts to infer the desired

---

† This is considered an advantage :-)

substitution from the current context. This technique greatly simplifies history substitution and causes it to be regarded as the norm rather than the exception. Users can afford to make much more use of the history list when the cost of a failed substitution is only one keystroke. Although context sensitive substitution requires more computational effort than interpreting explicit requests from the user, it has been found that the additional costs are low. Low cost context sensitivity has been achieved by adopting a **completion** model, rather than a substitutional model. The interface encourages users to type one or two characters of a command or a word, then request completion by the shell. This approach has several advantages:

- The history list can be accessed at any time during command entry.
- There is no need to remember the exact contents of the last few lines, or any event identifier.
- History completion integrates well with other shell features, such as filename completion. (See the description below)

Once the basic idea of history completion had been introduced, the only remaining problem was the exact nature of the user interface. Experimental versions proved that it was unnecessary to provide separate keys for line and word completion, provided that alternative completions for ambiguous requests could be easily selected. The following rules are used in the current version of Msh to satisfy history completion requests.

i If the current line is a prefix of a previously used line, the current line is replaced by the old line. Otherwise,

ii If the current word† (delimited by white space) is a prefix of a previously used word, the current word is replaced by the old word. Otherwise,

iii If the current word component (delimited by white space or slashes) is a prefix of a previously used word component, the current word component is replaced by the old word component. Otherwise,

iv Filename/command name completion is performed on the current word. (See following section, "Filename Completion"). Otherwise,

v If no (more) matches can be found, the line is returned to its original state and the terminal bell is rung.

vi The history list is always searched starting with the most recently used lines.

vii If, when a match has been found, the history completion key is pressed again, the search continues as though no previous matches had occurred.

viii Repetitions and certain inappropriate substitutions are suppressed, such as substituting a flag where a command name is required.

Msh allows the user to select from a number of alternative possibilities by pressing the completion key repeatedly. This approach is successful because not all possible completions are equally likely. The ordering imposed by the history list ensures that few keystrokes are needed to obtain the desired result; one keystroke is usually sufficient. This technique for resolving ambiguities is in contrast with the

---

† The current word is taken to be the word to the left of the cursor

filename completion scheme of Ken Greer's Tenex C Shell, which requires the user to remove any ambiguity before a completion is made. However, since Msh allows a completion to be aborted at any time during the selection process,† the user can choose to resolve ambiguities by supplying additional text if required.

The following examples demonstrate the finer points of the history completion mechanism. Suppose that the history list contains the lines:

echo Hello, World!
cat /etc/passwd
ed /tmp/foo

The most recently input line was *ed /tmp/foo* . In the following sequences, the first line is the original, and subsequent lines are obtained by pressing the history substitution key. The cursor is shown as _. In the interests of clarity, no filename completion has been performed.

1.	e	*original line*
	ed /tmp/foo	*line completion*
	echo Hello, World!	*line completion*
	ed	*word completion*
	echo	*word completion*
	e	*original line restored*
2.	ed /	*original line*
	ed /tmp/foo	*line completion*
	ed /etc/passwd	*word completion*
	ed /	*original line restored*
3.	cp p foo	*original line*
	cp passwd foo	*word completion*
	cp p foo	*original line restored*

In (1), the word *etc* is not substituted because it is inappropriate at the start of a command.

In (2), the line *ed /tmp/foo* appears only once. Msh suppresses the substitution of the word */tmp/foo* to avoid repetition.

Example (3) shows that completions need not be at the end of the line.

---

† Completions are aborted with the user's quit character.

4. *Filename Completion*

Msh allows filename completion similar to that supported by Ken Greer's "Tenex" C Shell. When a history completion has failed, or when the filename completion key is pressed, Msh attempts to perform filename completion on the current word. If the word is the start of a command, Msh searches through its builtin commands and directories on the search path to perform the completion. Otherwise, it tries to complete the last filename component of the word by examining the appropriate directory.

In order to allow the two mechanisms to be merged, alternative filename completions are treated in the same way as ambiguous history completions. Although the selection mechanism is less suitable for filename completions, it has proved beneficial to maintain a single style throughout the programme. There is no attempt to expand shell meta-characters on the command line in the style of the Korn Shell.

To illustrate the technique with an example, consider the following line. Once again, the cursor position is indicated with _.

cat /etc/pas_	*original line*
cat /etc/passwd_	*filename completion*

The same result could be obtained using the history completion key, once all matching history lines had been tried.

5. *Other Features and Improvements*

5.1 *Berkeley Style Job Control*

The C Shell is forced upon many users of Berkeley UNIX, because standard versions of the Bourne Shell do not support Berkeley job control. Msh supports Berkeley job control in a style almost identical to the C Shell. The most important differences are:

- There is no indication of current directory for foreground jobs.
- Notification of job termination is always immediate.
- Exit status is reported only for the last process in a pipeline.
- Foreground loops can be stopped.

5.2 Home Directory Expansion (user)

The C Shell's ~username syntax for home directories has been included in Msh. This syntax is understood by the filename completion system, which also allows usernames to be expanded after a ~ (tilde).

5.3 Function Definitions Override Shell Builtins

Function definitions have been modified to allow the redefinition of builtin shell commands, such as *cd* . Normally, function definitions with the same names as shell builtins are ignored. A new keyword *builtin* has been introduced to allow the redefined command to be called. One of the most common uses of this modification is in the redefinition of the *cd* builtin. This is frequently changed to provide facilities similar to the C Shell "directory stack". One of the most simple applications is to provide a simple "undo" facility for *cd :*

```
# Version of cd that saves last directory in $lastwd
cd(){
 lastwd=`pwd`
 builtin cd $1
}
# Undo the last cd command.
uncd(){
 cd $lastwd
}
```

Note that the *cd* in *uncd ()* is a reference to the function defined above, not the builtin *cd* . Thus *uncd* also sets the value of *$lastwd* , and "uncd; uncd" does not change the working directory.

5.4 Pwd Builtin Fixed Under Berkeley UNIX

Many shells that include the *pwd* (print working directory) command as a builtin do not behave as expected when the shell cd's through a symbolic link. Msh maintains the true (unique) pathname of the working directory at all times.

5.5 Initialisation Commands for Interactive Shells

Interactive invocations of Msh execute commands from an environment variable (SHINIT) on startup. This is analogous to the C Shell initialisation file (.cshrc). Use of SHINIT has been limited to interactive shells in order to avoid incompatabilities with standard shell scripts and to prevent possible security breaches. SHINIT is typically used to export function definitions to sub-shells, or to read initialisation files:

```
SHINIT='pg() more'
```

or

```
SHINIT='. $HOME/.shrc'
```

6. *Conclusion*

Msh is an extended version of the standard UNIX shell, providing character based line editing and a powerful history substitution mechanism. The simplicity and speed of the history interface have proved to be especially effective. The main features of Msh are:

- The interface is simple to learn. Almost all history substitutions are requested with a single key.
- The history list can be accessed more frequently and more effectively than in existing shells.
- It is upward compatible with the Bourne Shell; naive operation requires no additional knowledge.

7. *Acknowledgements*

The original version of Msh was designed and written at the Computer Science Department, University College London. Many people at UCL provided help and encouragement throughout the project. The editor was styled on an original package written by Bruce Skingle,‡ who offered many good ideas and performed much of the early testing. Jonathan Crompton was always a source of useful advice and encouragement. I owe special thanks to Nigel Martin,† who originally suggested the history completion mechanism. He has made many constructive comments throughout the delelopment of Msh, and helped with the final formatting of this paper.

---

‡ Now at Circulas Limited, London
† Now at The Instruction Set, London.

A. Appendix: Msh Editor Keyboard Functions

At the start of an interactive session, Msh automatically binds functions to particular keys. First, Msh tries to preserve most of the functions performed by the terminal driver by binding appropriate functions to the terminal driver special characters (i.e. erase, kill, interrupt etc). Then terminal keys with well defined uses are bound to obvious functions (e.g. delete line). Other common functions are assigned to the terminal's general purpose function keys. To provide compatability across terminal types and systems, control characters that have not been assigned are bound to fixed functions. This also ensures that certain heavily used functions are always available.

The following tables provide a complete list of Msh editor functions. The first table describes functions bound to single keystrokes and indicates which keys are bound to each function. Terminal driver special characters and control characters are shown against their associated functions. If a function can be bound to a key described by termcap/terminfo, the function is marked with an asterisk (*). The second table describes the action of the **more** key when used as a prefix to other keys.

Although Msh provides all the functions found in most screen editors, novices need learn only the first few commands of table 1.

Table 1: Single Key Functions

Name	Special Char	Termcap Defined	Control Char	Action
execute			∧J or ∧M	Execute displayed command [1]
rubout	erase		DEL	Delete last character
history		*	ESC or ∧]	History/filename completion [1]
delline	kill	*	∧X	Kill current line.
delword	werase	*	∧W	Delete last word [4]
eof	eof		∧D	End of file, if line is blank [2]
interrupt	intr		∧C	Kill current line [2]
left		*	∧H or ∧Y	Cursor left one space [1]
right		*	∧L or ∧P	Cursor right one space [1]
more		*	∧F	See table 2.
tab			∧I	Space to next tabstop [3]
quit	quit		∧\	Abort current substitution
redraw	rprnt	*	∧R	Redraw current line
literal	lnext		∧V or ∧∧	Input control char [1]
overtype		*	∧O	Overtype/insert toggle
filename		*		Filename completion.
insline		*	∧A	(n times) Insert nth previous line
put		*	∧_	Reinsert text last deleted
undo		*	∧G	Undo last modification
backword		*	∧B	Cursor back one word [4]
nextword		*	∧N	Cursor forward one word [4]
fastleft		*		Cursor to previous tabstop [3]
fastright		*		Cursor to next tabstop [3]
up		*	∧K	Scroll up history list
down		*		Scroll down history list
delete		*	∧T	Delete specified text [5]
delchar		*		Delete current character
delnxtwrd		*		Delete next word [4]
delsol		*	∧U	Delete start of line
deleol		*	∧E	Delete to end of line

Table 2: Key Sequences Introduced by more

Function	Modified function
overtype	Reset insert mode
undo	Undo, allowing further regression
more	Recall last line typed.
left	Cursor to start of line.
right	Cursor to end of line.
backword	Cursor back to last space.
nextword	Cursor forward to next space.
fastleft	Cursor to start of line.
fastright	Cursor to end of line.
up	Get "next" line in history list. [6]
down	Go to bottom of history list.
delete	Equivalent to **delete more**
delword	Delete to last space.
delnxtwrd	Delete to next space.

Notes:

1 Multiple control characters are bound to the same function to ensure that these functions will be available.

2 If the current line contains text, it is placed in the history list.

3 Tabstops are set every 8 characters.

4 Simple words are delimited by non-alphanumerics. See also table 2.

5 **Delete** can precede any cursor motion function. Text between the current cursor position and the new position is deleted.

6 This can be used to reenter sequences of commands, such as loops.

B. References

1 S.R. Bourne, "An Introduction to the UNIX Shell", *Bell System Technical Journal* , **57** (6), part 2, pp. 1947-1972.

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

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

4 D.G. Korn, "Introduction to Ksh", *USENIX Association Toronto Proceedings* , 1983.

5 W. Joy, "An Introduction to Display Editing with Vi", University of California, Berkeley, 1980.

6 J. Gosling, "UNIX Emacs", CMU, January 1983.

European Languages in UNIX[1]

Conor Sexton

Motorola International Software Development Centre
Cork, Ireland.

ABSTRACT

This document describes the approach adopted by Motorola to the internationalization of a UNIX System V derived operating system - Convergent Technologies' CTIX 3.0. The initial goal was the provision within CTIX of character sets enabling easy use and interchangeability of U.S English, French-Canadian and six European languages. The problem is broken down into its constituent parts and the solution, as well as the scheme of character set representations employed, is outlined. The many difficulties encountered during the development and testing are described. Finally, an insight is given into the direction of future Motorola development in the area of international UNIX.

---

1 UNIX is a Trademark of AT&T Bell Laborotories.

---

20 August 1985

1. Introduction.

The basis for the solution of the problem of CTIX 'internationalization' is found in the bi-directional flow of data between the user and the operating system.

The user has at his disposal one or more devices for input and output of data to and from the computer system. Typically, these devices are a terminal for input/output and a printer for output. Data sent from or to these devices pass through CTIX device drivers. The device drivers in turn send data to and receive data from the CTIX operating system, which manipulates and stores the data.

The procedure can be represented graphically as follows:

```
---------------          ---------------          ---------------
|             |  INPUT  |             | STORAGE  |    CTIX     |
|             |--------->|   Device    |--------->|             |
|   Devices   |         |             |          |  Operating  |
|             |<---------|   Drivers   |<---------|             |
|             |  OUTPUT |             |          |   System    |
---------------          ---------------          ---------------
                                                   MANIPULATION
```

from which it can be seen that four subsidiary functions are involved in the flow of data between the user and the operating system:

- o Input
- o Output
- o Storage
- o Manipulation.

It is with this functional breakdown in mind that we consider the problem of providing extended character sets within CTIX.

---

20 August 1985

2. Aspects of the Problem.

2.1 Requirements of 'international' CTIX.

The following were the initial goals of the 'international' CTIX implementation:

- Handling by CTIX of characters required in eight Latin-letter languages, specifically U.S. English, U.K. English, French, German, Spanish, Dutch, Swedish and French-Canadian.
- Retention of a 'vanilla' ASCII (English) capability. With the system in this mode, it would behave the same as earlier CTIX versions.
- Support of multiple concurrent languages on different devices on a single computer system.
- Correct terminal input and display for the particular (7-bit) terminal used with Motorola systems, for all the languages.
- Support of print output for all the languages for all of six different printer types used with the host system.
- Enable, insofar as possible, continued use of existing application software.
- Accommodation of both 8-bit and 16-bit CPU internal character representation to allow both immediate implementation and scope for future incorporation of more languages.

2.2 Internal Character Set Considerations.

When applied to character sets, the term 'internal' is here used to denote the manner in which a given character (either ASCII or non-ASCII) is represented in the processor to CTIX and its applications.

To internally accommodate the required languages necessitates use of at least an eight-bit internal character representation scheme. To enable a far greater range of characters to be used, with a consequent increase in the number of available languages, a 16-bit (2-byte) coding scheme is desirable. Within the European languages alone, there are nearly 800 different accented and other characters to be represented; when other languages (e.g Hebrew, Farsi, Kanji) are taken into account, at least a further 8000 character codes may be required. Japanese kanji alone incorporates 6349 "most frequent" characters! It was envisaged that an eight-bit coding scheme might provide an acceptable short-term solution, but that use of 16 bits would eventually be necessary. Both eight- and sixteen-bit schemes, as we shall see later, cause numerous problems in the use of CTIX and its applications.

It was considered desirable not to develop an entirely new, Motorola-specific, system of internal character coding but to determine if a suitable external code standard existed for this purpose.

2.3 Terminal Hardware and Software.

The standard terminal in use with the target Motorola processors is a 7-bit, intelligent, model. This greatly multiplies the problems inherent in the implementation of an extended character set, since the restriction to 7-bit transmission and reception by the terminal makes necessary intermediate translation of character codes between the terminal and processor.

The terminal download program determines the characters generated by depression of given keys or key sequences. It also defines which characters are displayed on the screen upon reception by the terminal of given character codes. The list of characters generated by keyboard depressions forms a keyboard definition table; the list of displayed characters forms a font definition table. The two tables are together assembled and loaded to make the download program. The Motorola terminal character set, which we will later describe more fully, has one 'configurable' range of characters -- 32 positions reserved for the characters of various international languages.

It is required that character codes generated by the terminal be translated to a form 'understandable' by CTIX and applications, and that characters sent to the terminal from the processor be reverse-translated to produce a correct terminal-displayable form. Although the current 7-bit terminal is the one to be immediately catered for, the translation scheme must not foreclose on future use of other 7- or 8-bit terminals.

It is also required that all of several terminals connected to the host processor may, if necessary, operate concurrently in different languages.

A very significant -- if mundane -- part of the terminal hardware considerations is the provision of keyboards and keytops acceptable in the various target countries. If, for example, keytops purporting to conform to the Swedish keyboard standard do not, in fact, conform, then it will be very difficult to market the entire 'international' solution, however elegant and effective, in Sweden.

2.4 Printer Hardware.

The other typical I/O device generally in use on the Motorola host processors is the printer, in various forms. It is required that six different printer types be supported. These include various daisy-wheel and band printers. Happily, for printers, we are only concerned with translation of characters output by CTIX or applications, so the size of the character translation problem is half that for terminals.

Character codes sent to a given printer type after translation from their internal CPU forms must match the codes expected by the printer if sensible output in the required language is to be printed. The correct daisy-wheel or band for the required language must be used. The codes expected by the printers determine the form of tables which must translate characters output from the processor to the printer.

2.5 Character Translation Considerations.

As already noted, facilities for translation of character codes in transit between the host processor and its printers and terminals must be

---

incorporated into the printer and terminal device drivers respectively. These must take the form of terminal- and printer-specific translation tables, residing in CTIX kernel space and accessed by the device drivers. Character codes transmitted from the terminal must be converted, in the terminal device driver by means of access to the terminal translation table, to the equivalent internal CPU representations. Character codes generated by an application in the CPU and bound for the terminal must be similarly reverse-translated to the correct screen-displayable form. Similar character translation facilities must be provided by the printer device drivers and translation tables, except that, in the case of printers, the translation is only in one direction.

The terminal and printer translation tables should take the form of lookup tables of character codes, supplied on input with a character code by the device driver and delivering, on output, a character code conforming to the requirements at the destination. In the case of terminals, both 'inbound' (character code en route from terminal to processor) translation and 'outbound' (internal CPU code en route from processor to terminal) translation must be provided. For printers, translation tables need only provide 'outbound' translation facilities.

It is required that the translation tables (and hence the language) in use should be software-switchable, i.e. that it should be possible with a CTIX shell directive to change the translation table currently being accessed by the terminal and/or printer device driver. Given that it is possible for the terminal to have many download programs corresponding to different languages, and that these can be changed easily, it is also desirable that the terminal translation table be capable of replacement. Typically, download files and translation tables supporting the same language will be in use on a given terminal at any one time.

2.6 CTIX Manipulation of Internal Characters.

The initial 'international' solution involves use of an eight-bit internal character representation scheme, with a 16-bit solution developed, but its implementation deferred. Use of a 16-bit character in any UNIX or UNIX-derived operating system environment would cause great problems, notably the doubling in length of all text files and disk files. CTIX support of 8-bit characters should be less problematic, but it is nonetheless likely that some CTIX utilities will fail. Results of testing of CTIX utilities with 8-bit characters are given later in this document.

Since it is required that the system be commercially saleable, and since there are many existing applications which run on Motorola hardware under (7-bit) CTIX, we must determine the impact of an 8-bit character scheme implementation on these applications and minimize this impact, if possible.

3. An 'International' Solution.

3.1 Internal Character Set.

One of the initial requirements of the CTIX internationalization project was that any internal character set scheme adopted should conform to some widely recognized character code standard. At the time of initiation of this project, several character set standards existed, but little information was available in respect of which, if any, of these standards

was gaining broad acceptance for the purpose of extending UNIX character sets. Motorola has adopted as its basic character set the Xerox Character Code Standard XSIS 058404. The Xerox character set is a generalization of familiar ISO and ANSI standards for coded character sets for text communication. Each character code in the set is represented by two bytes (16 bits), the first byte denoting the subset number in which a given character is to be found, the second representing the actual character itself. There is space in the Standard for each of 65536 characters to be represented with a unique, absolute numeric code. This covers the foreseeable requirements for known international languages. Presently-assigned Xerox Standard character subsets include:

- Character Set 0. Character set zero is the default 8-bit character codespace. The first 128 codes conform to ISO standard 646; the second 128 codes are the same as the supplementary graphic set for text communication from ISO 6937. Character Set 0. is shown in Fig 3.1.1.
- Character Set 041. JIS 1. Punctuation and Symbols not in Character Set 0.
- Character Set 042. JIS 2. Punctuation and Symbols not in Character Set 0.
- Character Set 043. Extended Latin characters.
- Character Set 044. JIS Hiragana.
- Character Set 045. JIS Katakana.
- Character Set 046. Greek.
- Character Set 047. Cyrillic.
- Character Set 0164. Miscellaneous Japanese symbols.
- Character Set 0356. General & Technical symbols 2.
- Character Set 0357. General & Technical symbols 1.
- Character Set 0360. Ligatures, Graphical entities and Field Format symbols.
- Character Set 0361. Accented Characters.

XEROX Character Set 0

ASCII/ISO/CCITT Roman Alphabet & Punctuation

	000	020	040	060	100	120	140	160	200	220	240	260	300	320	340	360
00			space	0	@	P	` Grave (s)	p				° Degree		— Bar	Ω Ohms	ĸ Grnland
01			!	1	A	Q	a	q			¡ Span.	± Plus-Min	` Grave	¹ Super.	Æ Diphth.	æ diphth.
02			" Neut.	2	B	R	b	r			¢ Cent	² super.	´ Acute	® Reg'd	Đ Croat.	đ Croat.
03			#	3	C	S	c	s			£ Pound	³ super.	ˆ Circum.	© Copyr't	ª Span.	ð Iceland
04			¤ Curr.	4	D	T	d	t			$ Dollar	× Times	˜ Tilde	™ Trd mrk	Ħ Maltese	ħ Maltese
05			%	5	E	U	e	u			¥ Yen	µ Micro-	¯ Macron	♪ Note		ı Dotless
06			&	6	F	V	f	v				¶ Pilcrow	˘ Breve		LJ Dutch	ij Dutch
07			' Apos.	7	G	W	g	w			§ Section	· Center	˙ Dot		L· Catalan	l· Catalan
10			(	8	H	X	h	x				÷ Divide	¨ Diereses		Ł Polish	ł Polish
11			)	9	I	Y	i	y			‘ left	’ right			Ø Norw.	ø Norw.
12			*	:	J	Z	j	z			“ left	” right	° Ring		Œ Diphth.	œ diphth.
13			+	;	K	[	k	{			« l. Quote	» r. Quote	¸ Cedilla		º Span.	ß German
14			,	<	L	\	l	|			←	¼ En frac.	_ Undrline	⅛ En frac.	Þ Iceland	þ Iceland
15			— Minus	=	M	]	m	}			↑	½ En frac.	˝ Db acute	⅜ En frac.	Ŧ Lapp	ŧ Lapp
16			.	>	N	^ Circum. (s	n	~ Tilde (s)			→	¾ En frac.	˛ Ognek	⅝ En frac.	Ŋ Lapp	ŋ Lapp
17			/	?	O	_ Low bar	o	Delete			↓	¿ Span.	ˇ Hacek	⅞ En frac.	ŉ S. Africa	☒

Reserved, unassigned

Character Set Select Code

Reserved, not used

Fig. 3.1.1 Xerox Standard XSIS 058404 -- Character Set 0.

The long-term requirement for the 'international' CTIX solution is to incorporate the full 16-bit character code range provided by Xerox XSIS 058404. It has already been noted that the effects of a 16-bit internal representation scheme on the utilities of UNIX or a UNIX-derived operating system would be traumatic. An eight-bit solution is therefore the first to have been implemented for CTIX. It has been necessary to use an eight-bit (256-element) character set, incorporating the most-used Xerox Standard codes, as the basis for the 'international' CTIX implementation. To this end, Motorola has adopted its own private character set, referred to as Motorola Private Set 040. This character set is depicted in Fig. 3.1.2 below. A character code incoming to the processor from a terminal passes through the terminal translation table accessed by the tty driver and is thus translated to its equivalent Xerox Standard representation. Code in the tty driver further translates the Xerox representation to the appropriate character in the Motorola Private Set 040. This last representation is the internal code used by CTIX utilities and applications. The reverse translation process is used for internal CPU codes sent to terminals and printers. The adoption of the Motorola Private set is intended as a temporary measure until there is some prospect of UNIX or a UNIX derivative successfully handling 16-bit codes.

Motorola Private Character Set 40_8 Right Half

	200	220	240	260	300	320	340	360
00			space	ã	ô	¢	—	¬
01			Ä	ä	õ	¡	┼	÷
02			Å	å	ö	¿	■	×
03			Ã	œ	œ	┌	□	✓
04			Æ	ç	ø	/	////	µ
05			Ç	è	ß	┐	\\\\	ŷ
06			É	é	ù	\	¶	ý
07			Ñ	ê	ú	└	†	ÿ
10			Ö	ë	û	\	™	°
11			Õ	ì	ü	┘	©	•
12			Œ	í	ů	/	®	—
13			Ø	î	°	├	º	|
14			Ü	ï	¨	┤	ª	¸
15			à	ñ	£	┴	ƒ	´
16			á	ò	§	┬	₱	π
17			â	ó	¤	|	¥	╳

Reserved, Not Used

Character Set Select Code

Fig 3.1.2 Motorola Private Character Set 040.

3.2 Terminal Hardware and Software.

The Motorola proprietary 7-bit terminal currently in use with the targe processors incorporates, on a ROM, its own character sets. These character sets conform to ANSI x3.64 specification, which details the interpretation of character codes exchanged between the host processor and the terminal. Three ANSI character sets are supported: G0, G1 and G2. The host processor switches back and forth between these sets by sending different control sequences thus:

Name	Sequence	Operation
SI (Shift In)	0/15	Select G0 character set (ASCII)
SO (Shift Out)	0/14	Select G1 character set (Line Drawing)
SS2	ESC N	Select G2 Character set for next character only.

To extend this character set to allow use of 'international' characters, a portion of the G2 character set is bank switched to allow one of eight groups of international characters to be substituted for codes 4/0 through 5/7. This provides 32 configurable character code spaces in terminal character set G2. Selection of these eight character groups is controlled by setting of a dedicated register in the terminal download program.

ASCII codes are transmitted and received by the terminal in their plain, 7-bit, forms. Other characters, of higher numeric values than the ASCII limit of 128, are sent and received by means of three-byte escape sequences. Since the international characters are configured into set G2, it can be seen from the table above that the first two (of three) seven-bit sequences used to represent a special international character will be ESC N, when transmitted or received by this terminal.

The terminal download program determines exactly the code sequences to be sent and how incoming code sequences are interpreted by the terminal. The source form of the program is of two tables, the keyboard and font definition tables. Upon depression of a given key or sequence of keys, the download program, configured for one of eight possible languages, causes the terminal either to transmit a given ASCII code or escape sequence, or to interpret in a displayable form an incoming ASCII code or escape sequence.

It is possible for a single terminal to be used with any of eight download programs configured for any of eight different languages. Several terminals may be used on one host processor, all using different languages. Keytops for keyboards conforming to the different national keyboard standards, including use of 'dead-keys' for characters not explicitly represented on the keyboards have been made available. We have defined the internal CPU character set representation method and the transmission and reception of character codes by the terminal. It remains to define the method of character code translation between the I/O devices and the processor.

3.3 Character Translation.

The character translation tables developed for 'international' CTIX take the form of line-image files which define terminal and printer input and

output character code translations. The present translation tables are specific to the terminal and printers already referred to, but it is possible with little effort to modify translation tables to accommodate other terminal and printer types. Terminal and printer translation tables differ in their form; there follows a description of both.

3.3.1 Terminal Translation.

All character codes generated by or destined for the terminal undergo translation. ASCII characters are disregarded, but escape sequences en route from the terminal to the processor ('inbound') are converted to the internal CPU code, while character codes destined for the terminal from the processor ('outbound') are converted to the proper escape sequences. Fig. 3.3.1.1 below depicts a segment of a terminal translation table. The table is divided into inbound and outbound translation directives.

The inbound translations shown operate as follows: when ESC N A is received from the terminal, character number 0243, character set zero, is extracted from the Xerox character set, to be then further translated to character code 0315 (Pound (Sterling) sign) in the Motorola Private Set 040. This code, which is the internal CPU representation, is delivered to CTIX or an application program. When ESC N C is received from the terminal, character number 0310, character set zero (diaeresis) is extracted from the Xerox set. The \a sequence indicates the presence of an accented character and that the character to be accented is 'U'. A 'U' diaeresis must be delivered to the processor, so code 0254 is selected from the Motorola Private Set 040 and delivered.

Outbound translation is the reverse process. The 'primary' directive selects the G0 terminal character set as default for output to the terminal; the G1 set is labelled 001 by the 'cselect' directive. If Xerox character number 0243, character set zero, is received bound for the terminal, then ESC N A is generated and sent to the terminal. Similarly, if \000\373 is received, ESC N B is generated and dispatched to the terminal, where it is interpreted as a German esset. If \000\310 (diaeresis) is received and if the following character is 'U', ESC N C is sent to the terminal, to be interpreted as a 'U' diaeresis. Question marks '?' in the outbound part of the tramslation table represent Xerox codes for which there exist no counterpart in the terminal font definition code. For example, Xerox character \000\241 denotes the Spanish inverted exclamation mark. The German font definition does not include this character so, in the segment of the German translation table shown in Fig 3.3.1.1, a question mark is transmitted to the terminal in its place.

```
inbound
translate \EN\x range A C
\000 \243       # pound sign
\000 \373       # esset (German)
\a \310 U       # U diaeresis
# When the input sequence ESC N is received from the terminal,
#   if the next character is A then output \000 \243,
#   if the next character is B then output \000 \373,
#   if the next character is C then output \000 \310 \000 U.

outbound
primary \017       # ASCII shift-in code selects G0 character set.
cselect \001 \016  # ASCII shift-out code selects G1 character set.
translate \000\x range \241 \377
?
?
\ENA               # pound sign
\044               # dollar sign
                        .......
                        .......
\ENB               # esset      position 373
?
?
?
?                  # position 377

translate \000\a accent \310
U \ENB             # U diaeresis
```

Fig 3.3.1.1 Terminal Translation table, inbound and outbound.

In the present implementation, there is one terminal translation table and one terminal download file in the system for each language supported. The download file required is selected at terminal boot time. Also at terminal boot time, a default terminal translation table is selected and activated. A set of shell-level commands has been provided to facilitate the user's changing active translation tables. The most general of these commands is as follows:

```
cstty [-]cstrans [-]cst16 [-]csfmt7 [-]cs040 t=<term>.<lang>
```

The directive

```
cstty -cstrans -cst16 -csfmt7 -cs040 t=
```

switches off all translation features, and is useful if the terminal is intended to operate in 'vanilla', predominantly ASCII mode. When the 'cstrans' option is selected, the terminal driver translates inbound and outbound characters to and from the internal character representation. The 'cst16' option is currently always deselected since it causes characters in their two-byte Xerox code form to be supplied to CTIX as a valid internal character representation. The 'csfmt7' option selects representation of internal character codes as 7-bit values, preceded and followed by SO and SI. Selection of 'cs040' enables translation of the type described in this document, according to the rules specified by a translation table of the form <term>.<lang>. Thus, for translation to be enabled, using Motorola Private Set 040 and the German translation table, the following directive is used:

```
cstty cstrans -cst16 -csfmt7 cs040 t=tm31.deut
```

3.3.2 Printer Translation.

Character translation tables for printers are similar to those for terminal, but have no rules for 'inbound' characters. An example segment for a daisy-wheel printer is given in Fig 3.3.2.1.

```
# Revision History:
#
#       GOC.      ISDC, Cork, Ireland.          Feb. 13 1985.
#       Initial version.
#
#
primary           \E(B      #USA-ASCII
cselect \001      \E(A      #UK
cselect \002      \E(K      #German
cselect \003      \E(2      #Swedish
cselect \004      \E(R      #French
cselect \005      \E(4      #Spanish
cselect \006      \E(1      #Italian
cselect \007      \E(3      #Norwegian

outbound          # internal character codes translated into device-
                  # dependent codes

translate \000\x range \241 \376
\005 [            # \241    Inverted Exclamation Point
?                 # \242    Cent Sign
\001 #            # \243    Pound (Sterling) Sign
\044              # \244    Dollar sign
?                 # \245    Yen Sign
?                 # \246    Reserved
\002 @            # \247    Section Mark
?                 # \250    Reserved
?                 # \251    Left single quote
?                 # \252    Left double quote
?                 # \253    Left double guillemet
?                 # \254    West arrow
?                 # \255    North arrow
?                 # \256    East arrow
?                 # \257    South arrow
\004 [            # \260    Degree sign
?                 # \261    Plus/minus sign
?                 # \262    Superscript 2
?                 # \263    Superscript 3
?                 # \264    Multiplication Sign
?                 # \265    Micro sign
?                 # \266    Paragraph Mark
?                 # \267    Centered dot
?                 # \270    Division Sign
?                 # \271    Right single quote
?                 # \272    Right double quote
?                 # \273    Right double guillemet
?                 # \274    Fraction 1/4
?                 # \275    Fraction 1/2
?                 # \276    Fraction 3/4
\005 ]            # \277    Inverted Question Mark
?                 # \300    Reserved
```

Fig. 3.3.2.1 Printer Translation Table.

A shell-level utility is provided with CTIX for the easy configuration of printer types and translation table options.

3.4 CTIX Manipulation of Internal Characters.

Before system testing of the 'international' CTIX began, it was envisaged that the operation of utilities would fall into three classes:

1. those working correctly with 8-bit internal characters
2. those indifferent to 8-bit characters, but giving unexpected results
3. those refusing, under some or all circumstances, to handle 8-bit characters.

Commands which have been tested give the following results:

Class 1		Class 2	Class 3	
ar	nice	awk	banner	prof
bc	nl	diff	basename	pwck
bdiff	nm	diff3	bfs	sed
cal	nohup	echo	cc	sh
cat	pack	file	chgrp	sort
chmod	passwd	lint	chown	spell
cmp	paste	regcmp	col	test
cp	pcat		cu	tsort
comm	pr		cxref	vi
cpio	ptx		date	wait
crypt	pwd		dc	who
csplit	rm		dd	write
cut	sdiff		dircmp	
devnm	setuname		ed	
df	size		edit	
du	sleep		env	
factor	split		expr	
find	tail		getopt	
grep	tee		hyphen	
join	time		install	
kill	touch		login	
ld	tr		logname	
line	tty		lorder	
link	uname		mail	
ln	uniq		message	
ls	unpack		mm	
make	uucp		mmt	
mount	umask		mvdir	
mv	wall			
newgrp	wc			
news	xargs			

By far the most critical of the above results is the fact that the shell and all the standard CTIX editors use the eighth bit for their own purposes, thus rendering themselves inoperable with the internationalized

version of CTIX. Work is currently in hand towards the modification of editor and shell source code to remedy this problem.

Extensive testing was also carried out on the operation of several well-known commercial application packages with the extended character scheme. While, on some of these applications, the effect of eight-bit characters is minor, none of the six packages tested -- a word processor, relational database, spreadsheet and implementations of COBOL, BASIC and SIBOL -- was bug-free.

4. Future Direction.

We believe that the approach adopted in this implementation is a good one and that the character set convention adopted is as good as any other. We wish, however, to concentrate our efforts in the future on a System V 'standard' version of UNIX, rather than a derivative of the operating system. We await announcements from AT&T of a 'standard' UNIX accommodating 8- or 16-bit internal character code representations, and would like to adopt a de facto character set standard, if such emerges.

---

20 August 1985

Communications Solutions for Mainframe UNIX

J.F. Hughes
Manager, Summit Operations
Amdahl Corporation

Bringing UNIX † to the world of high-speed mainframe computers presents major challenges to communications. UNIX was developed around full-duplex, asynchronous terminals, which are not commonly used in large data centers. In creating UTS, ‡ which is a port of UNIX to the 370 architecture class of processors, special interfaces and protocols were established to allow the use of full-duplex, asynchronous terminals as well as the more standard bisynchronous terminals. With this full-duplex support software, users can interface with UNIX applications in exactly the same fashion as would be used on mini- or micro-processor based implementations of UNIX. Thus, an editor such as VI is available.

This paper discusses the technical aspects of full-duplex support software as well as current interfaces for X.25, Ethernet and other networks. Through these communications solutions and the power of UNIX on a mainframe computer, the end-user has an effective tool for creating and accessing corporate applications.

1. UNIX COMMUNICATIONS OVERVIEW

This section provides an overview of the communications paths used between UNIX and its terminal devices.

1.1 UNIX and Communications

The I/O portion of a standard UNIX system is divided into two distinct parts: the block I/O system and the character I/O system.

A device in the block I/O system consists of randomly addressed, auxiliary memory blocks of a particular size. In a typical microprocessor or miniprocessor implementation of UNIX, the block size is nominally 512 or 1024 bytes; however, in UTS, the block size is a larger 4096 bytes. The file systems available to the user use this block I/O system to access the mounted disk devices.

The character I/O system contains all devices that do not fall into the block I/O system. As pointed out in the Bell System Technical Journal, † the term "character I/O" is a misnomer and really should be "unstructured I/O", or I/O that does not use the blocking system.

---

† UNIX is a trademark of AT&T Bell Laboratories.
‡ UTS is a trademark of the Amdahl Corporation.
† The Bell System Technical Journal, July-August 1978, Vol. 57, No. 6, Part 2, p.1939.

As a consequence of this definition, character I/O requests from the user are passed directly to the device driver for the particular device, whether it is a communications line, line printer, or even a disk if it doesn't follow the block I/O buffering procedures. The actual hardware device driver determines how the I/O is handled rather than higher level software in the kernel.

A discussion of UTS communications centers on a description of the various kernel device drivers that support the communications devices found in a UTS system.

For example, user I/O to a particular terminal is carried out by presenting a stream of characters to the device driver for output, and expecting a stream of characters from the device driver for input. The actual software interacting with the device driver, of course, may be a shell, a user application or even an error message routine deep in the kernel.

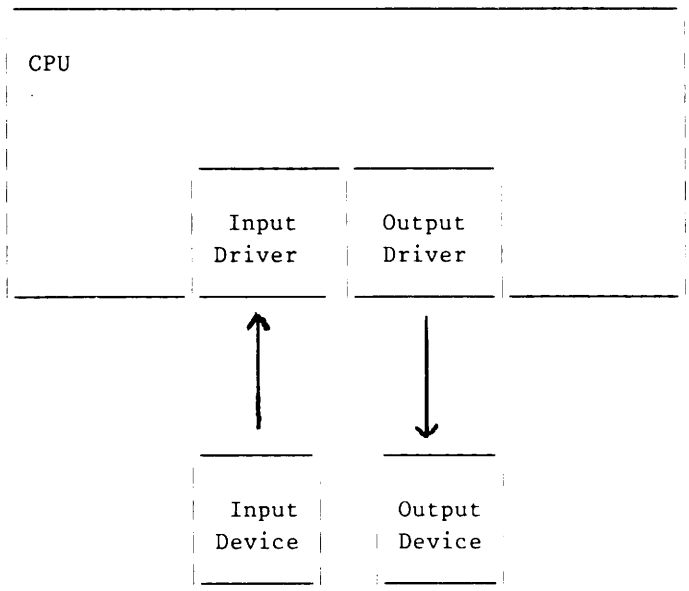


fig.1: Traditional UNIX I/O Interface

1.2 Fundamental I/O Device Support

To discuss I/O device support in a UNIX system, it is helpful to review basic terminology. An understanding of full-duplex, half-duplex, echoplex, and canonical processing will lead into a discussion of the UNIX and mainframe environments.

1.2.1 Full-Duplex Communications

In a classic UNIX system, a user terminal is actually treated as two distinct devices - one device is a display screen that responds to characters from an output device driver in the UNIX kernel, and the second device is a keyboard that provides characters to an input device driver.

Physically, the two devices are housed in the same cabinet, but from the viewpoint of the UNIX system, they are logically separate. Input from the keyboard device may or may not relate to the actual output being placed on the screen device by the output device driver.

As a consequence of this architecture, the echoing of characters to the display screen as they are typed on the keyboard is actually a function of the software in UNIX - the software receiving characters from the input device driver may or may not decide to echo by sending the characters to the output device driver.

In this definition of full-duplex devices, there are separate input and output paths between the device and the central processor. The software running on the central processor decides whether data coming in over the input line is echoed back over the output line.

For example, a Teletype was an early console device. It contained two physical interfaces between the keyboard, print element and the central processor. Using a current loop protocol, pulses on one line from the keyboard went to the input driver in the host, and pulses on the other line from the output driver went to the print head to print characters. There was no physical interaction between the two hardware paths.

Today, a somewhat standard RS-232 interface describes the analogous transmit and receive (both are one-way) lines that, together with some common signaling lines, connect a modern display/keyboard with the central processor in a full-duplex mode.

With these modern devices, a single standard host-resident device driver handles all similar physical devices. Within such a driver, there is an input portion and an output portion to handle the data following from and to the family of devices.

1.2.2 Half-Duplex Communications

In a physical sense, half-duplex communications refers to the use of a bidirectional communications line that can both send and receive data. This is contrasted with the full-duplex environment described previously, which uses dedicated input and output lines.

In this half-duplex mode, after data is passed in one direction, the line must be turned around electronically to allow data to pass in the other direction. This is a physical operation. As long as the drivers in the central processor know how to do this, application software in the central processor will still see separate input and output device drivers and will still have to decide whether or not to echo characters to the display as they are received from the keyboard.

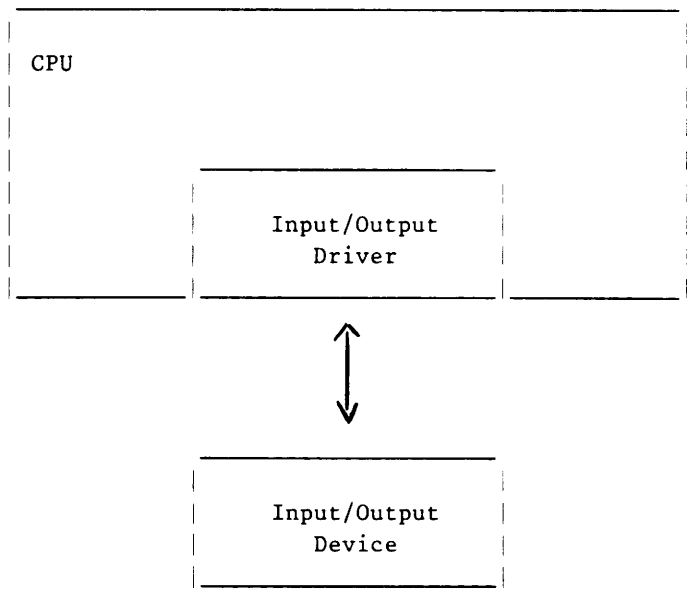


fig.2: Half-Duplex I/O Interface

A consequence of this physical operation is that while output is being sent to the terminal, input cannot be received from the terminal. In most systems, some buffering takes place at the terminal to save up terminal input until the communications line is ready to send data to the central processor.

This definition is a physical definition of the connection between the device and the central processor.

Sometimes, half-duplex devices are permanently unidirectional and will either transmit or receive data (but not both) from a central processor. This is known as a simplex connection. A console printer is an example of this device.

1.2.3 Echoplex Communications

On some ASCII display terminals, a switch setting labeled half-duplex is available. Usually, this is not the physical half-duplex mode described in the previous section, but a mode in which the characters generated by keystrokes on the keyboard are placed on the display as well as being sent to the central processor.

This is known as echoplex and is really a variant of the physical full-duplex mode described previously. There are still separate input and output paths between the device and the host processor, but through the use of this switch, printable characters are echoed to the display as they are sent to the host. This both reduces processing time in the host and loading on the communications line.

In this mode, the central processor does not have to echo characters from the input device driver to the output device driver. The user's terminal has taken care of this operation.

1.2.4 Canonical Processing Modes

When using full-duplex terminals, UNIX programs interact with the terminals in one of two modes: canonical or noncanonical.

In the canonical mode, the terminal driver in the kernel assembles input characters into lines delimited by newline, end of field or end of line characters before the input lines are presented to the application. Erase and line kill functions are enabled and character echoing is done by the terminal driver.

In this mode, the user can perform simple line editing and the application does not have to be concerned with echoing.

In the noncanonical mode, the terminal driver passes characters from the device directly to the program, which is then responsible for echoing characters to the display. Simple editing functions, such as line kill, must be performed by the program.

The majority of UNIX programs use canonical processing because it allows the programs to remain independent of the source of the input data. Programs of this type may then use the powerful I/O redirection facilities of UNIX and may also effectively use half-duplex terminals.

Noncanonical processing is used by fewer, but more significant programs. Most of them are full-screen editors; however, some are games, a few are graphics-related and a few are used in intercomputer communications.

1.3 UNIX and Mainframe Communications Environments

UNIX, as an operating system, has its origins in the mini and microprocessor hardware environments where a great variety of user terminals and central processors exists. This necessitated a simple interface to various communications devices, but it had a pronounced effect on the style of communications.

This is most evident when the connection between full-duplex ASCII terminals and the central processor is examined.

1.3.1 Historical UNIX Device Connections

Early UNIX processors used a single processor architecture where the various I/O devices were addressed by the processor over a common bus.

Output to the device was accomplished by using an output device driver that sent a character at a time to the device using the I/O instructions of the central processor. Although there might be some buffering and error recovery procedures in the device driver, there was no further logical processing after the character was sent by the central processor.

Input from the device was character-oriented and interrupt-driven: as a character appeared from the device, an interrupt was signaled in the host and the central processor was vectored to a service routine to read the character and place it in the appropriate input queue. When the character had been read, the central processor was released to resume its interrupted task.

Several observations may be made about this classic interface:

- Devices are connected directly to the central processor with minimal character processing between the device and the central processor.
- The central processor is involved in each transfer of a character to or from the I/O device.
- There is no fundamental hardware relationship between receiving and sending characters, allowing true full-duplex operations.

1.3.2 Mainframe Device Connections

In the mainframe environment, devices are normally connected directly to an intermediate processor that can do I/O processing in parallel with the processing of the central processor. This procedure recognizes the inherently greater speed of the central processor, which is better utilized by operating on non-I/O tasks, leaving the I/O activities to more specialized processors.

A channel is a specialized processor in the mainframe environment designed for I/O processing. This hardware device is a part of the processing system and accesses the central memory both for instructions and for data transfer.

Devices are connected either directly to the channel, or to a controller or communications processor that is then attached to a channel.

Communications devices such as the full-duplex ASCII terminals common to the UNIX world cannot attach directly to a channel. Consequently, a communications processor such as the Amdahl 4705 Communications Processor is used to provide an interface between the device and the channel.

A similar picture can be drawn for other devices such as DASD or tape drives, which use controllers rather than a communications processor to connect to a channel.

A 4705 used within a typical mainframe environment usually contains one of the following programs:

- Emulation Program (EP) is the software that was created to emulate an early class of hardwired communications processors. It supports asynchronous and binary synchronous communications lines and is relatively stable due to its long life. Both public domain and IBM program product versions exist.
- Network Control Program (NCP) is the System Network Architecture (SNA) component that resides in a 4705. It is undergoing continual change and is a licensed IBM program product.
- Partitioned Emulation Program (PEP) allows both EP and NCP to reside in the same physical 4705.

A normal EP provides for the definition of half-duplex and full-duplex lines, but these definitions are considerably different from the common UNIX definitions.

With these standard EP definitions, half-duplex and full-duplex lines differ in the way the lines are enabled and how the lines are turned around after a write operation.

The full-duplex lines always have the Request to Send lead on. The half-duplex lines turn off the Request to Send lead during terminal read operations. Both types of lines send data in only one direction at a time and, during a read operation, the host doesn't see the data until an End of Line character is received.

In both of these cases, there is only one data path between the device and the host processor, which makes it impossible to implement a true physical full-duplex connection that would permit non-canonical mode.

The EP definition of full-duplex does not satisfy UNIX requirements. Consequently, EP must be modified to support UNIX-style full-duplex operations. Through extensions to the procedures for creating an EP, additional code supporting full-duplex operations is included in the 4705 control program. This code is contained in the UTS/F program product.

When an EP is installed in the 4705, subchannels are used to represent the data path between the host and a device attached to the 4705. With a maximum of 256 subchannels per physical channel, a normal EP supports 255 devices (or communication lines) and a native subchannel, which is used to control the 4705.

Several important observations about mainframe I/O can be made at this point:

- The central processor is never interrupted directly by the external I/O device. The channel is used to transfer data to and from central memory and then signal the central processor.
- Although the central processor and channel can be involved in the transfer of data on a character by character basis, it is normally advantageous to transfer data in blocks. This blocking reduces the number of times the host processor must respond to channel activity.
- The channel, since it is a processor that transmits data between central memory and the external device/controller, can transmit data in one direction or the other, but not in both directions at the same time.

As a consequence of these architectural characteristics of the mainframe environment, the interface between the host and full-duplex devices in the mainframe world is more complicated than the interface in the micro or minicomputer environment.

The Amdahl UTS/F program product is software that solves this problem by integrating standard UNIX full-duplex devices into the mainframe environment.

1.3.3 Hardware Support for UNIX Communications

As the power of central processors in typical UNIX miniprocessor systems increases, vendors are discovering that it is advantageous to provide supplemental processors specifically designed for I/O. Typically, these microprocessors are contained on a device support board that is physically within the miniprocessor.

These support processors can perform routine I/O tasks, freeing the central processor for more important activity. In many ways, these boards resemble the channel or front-end processor architecture of the mainframe environment.

In early versions of UNIX, support for these boards was the vendor's responsibility. With the release of System V UNIX, however, a special interface protocol was established for these processors.

Known as the virtual protocol machine (vpm), this specification allows a vendor to implement the vpm interface on a particular microprocessor board within the overall processor system. Higher-level protocols such as Remote Job Entry (RJE) that are defined on the virtual protocol machine can then be downloaded to the microprocessor board with little effort on the vendor's part.

The 4705 communications processor does not implement the virtual protocol machine interface. From the host central processor's point of view, the 4705 only provides an interface to binary synchronous half-duplex lines, asynchronous half-duplex lines and asynchronous full-duplex lines. Any higher level protocols on other UNIX systems that might be defined through a vpm script are implemented in the host.

2. *UTS TERMINAL SUPPORT*

This section provides information on the two typical terminal interfaces provided to users on UTS.

2.1 *General*

Historically, UNIX users have used full-duplex ASCII terminals that are connected to the central UNIX processor. The applications (or shell) controlling the terminal expected to see a serial stream of characters coming from the user's keyboard, and send a serial stream of characters to the user's display.

With the early slow-speed teletype devices, short keyboard sequences (such as the abbreviated commands available in ed(1)) and even shorter responses to the display (such as little or no indication of command state or prompt) were the norm.

Today's large data processing center user, however, is accustomed to using 3270-style block mode terminals. In this environment, full-screen operations are standard. Command sequences explicitly describe the action desired and voluminous responses are common.

2.2 *Full-Duplex ASCII Terminals*

Most terminals of this type contain a keyboard and a display, which logically operate as separate devices from the central processor's point of view.

Data is passed in American Standard Code for Information Exchange (ASCII), which is a seven-bit-plus-parity † code established by the American National Standards Institute (ANSI). This provides for 128 different characters and differs from the code used in most IBM equipment.

Individual keystrokes cause data to be passed to the host. This means that when a key is pressed, its character representation is immediately passed to the central processor as input to the program requesting the terminal read. No buffering or other logic is used that might cause the delay of the character until other keys are pressed. Even if the transmission connection is best used by blocks of data (such as when there is a large cost to set up or break down the communications link that is better

---

† Some UNIX programs, such as uucp(1C), use the full eight bits to allow transfer of binary files without using encoding schemes.

spread across a block of data), individual characters must be transmitted on their own.

Another way of looking at this is to recall that the lines are asynchronous: there is no way to predict the time between the receipt of characters since it is a function of how fast the typist uses the keyboard.

The characters themselves are made up of regularly spaced signals on the line. The terms baud and bits-per-second (bps) measure the time between the bits that make up the characters and not the time between the characters.

For example, data transmitted to a display at 4800 bps from one system may appear to be received faster than data transmitted at 9600 bps from another system if the time between characters from the 4800 bps system is considerably less than the time between characters from the 9600 bps system. However, the bits making up each individual character are transmitted twice as fast in the 9600 bps system as the 4800 bps system.

The immediate receipt of characters allows for full program control over the display. In addition, special combinations of keys may be used that are beyond the normal shift-key combinations found on typewriters.

The early displays or printer elements did little more than place characters on a page or screen at the cursor or printhead position when they were received from the host. Many of the standard UNIX programs expect terminals to operate in a mode in which the only control characters on the output stream are carriage returns or line feeds, which control the position of the print head.

Newer devices allow for many different escape sequences to control the output device. These controls range from simple sequences to place the cursor or print head in a new position to elaborate sequences that change character sets or invoke special programs within the terminal. With the lowering cost of electronics, the trend will be to have more functionality within the user's workstation.

Although there have been some attempts to develop a standard set of terminal escape sequences, most terminals have unique character sequences to invoke special functions.

To provide a common interface to a wide variety of terminals with varying escape sequences, many versions of UNIX, including UTS, use a terminfo(4) database to describe the various terminals. Using this database, programs such as the full-screen editor vi(1) or the CRT screen handling and optimization package curses(3X) can use the full capabilities of most standard ASCII terminals.

As an example, the vi(1) editor has a command to move the cursor one word to the right on the same line. With terminfo in the ASCII environment, vi simply looks up the escape sequence to move the cursor one character to the right and uses it to move the appropriate number of spaces.

2.3 Half-Duplex EBCDIC 3270 Terminals

The 3270 Information Display System is the general name for a group of devices typically found in the mainframe environment. The family consists of a number of terminals, printers, controllers and auxiliary devices that follow a standard architecture.

All of these devices require the use of a controller, which serves as an interface between the device and either the channel (local mode) or a communications line (remote mode).

There are two generations of the 3270 family that differ in their device characteristics and the protocols used on the coaxial cables connecting the devices to the controllers.

The earlier family used the 3271 (remote) and 3272 (local) controllers with 3277 displays. This family of devices is often referred to as a type B coax system in recognition of the type of hardware signals used on the coaxial cable that connects the terminals to the controller.

The later family uses the 3274 controller (with different versions for local and remote use) and the 3278/3279/3280 displays. This family is often referred to as a type A coax system.

Various printers can be attached to either or both of these families.

Locally attached terminals are connected directly to a controller, which is connected to a channel. Up to 32 devices are supported on one controller, and a number of controllers may exist on the same channel.

Remote terminals are connected to a controller which is then connected to a binary synchronous (BSC or bi-sync) protocol communications line. An Amdahl 4705 Communications Processor then provides an interface between this line and a channel.

A 327x display and its controller operate in a master-slave relationship: the controller is the master and determines when data should be passed between the device and the host interface. This is distinguished from the micro or miniprocessor environment, where the display is directly attached to a processor: if a key is pressed on the keyboard, an interrupt is generated in the processor to read the data and store it in some buffer.

In the earlier family of 3277 displays, characters were saved in a local display buffer after being echoed to the screen. After polling by the controller, the display would send a fixed size buffer of characters to the controller, which would then forward portions of the buffer to the host.

In the later family of 3278 displays, characters are transmitted directly to the controller, which may then echo them to the screen. The transmission of data to the host in this case is still under control of the controller, but occurs from buffers in the controller rather than from the terminal.

As discussed earlier, transmission of data between the controller and the host is in a physical half-duplex mode rather than the historical UNIX full-duplex mode.

Data transfer uses the Extended Binary Coded Decimal Interchange Code (EBCDIC), which provides for 256 different characters and is different from the encoding sequence used for characters with ASCII terminals.

With 327x devices, there is a higher level protocol known as the 3270 Data Stream Protocol, which is a formatted data stream used to transmit data between an application program and a terminal or printer. While this is not as encompassing as the terminfo database used with ASCII terminals, it does provide a method for independently accessing the various models and devices of the 3270 family.

The 3270 family of displays is field-oriented. Hardware support within most of the devices recognizes the data fields specified in the 3270 data stream protocol. This is different from the character orientation of ASCII devices.

In UTS, an extremely powerful and comprehensive full-screen 3270 editor, ned(1), was developed that has a strong similarity to both mainframe editors and ed(1).

Many of the key system utilities in UTS have full-screen 3270 interfaces built around the C language quickscreen preprocessor qs(1). Through this facility, programs can quickly and efficiently access the full-screen features of the 3270 family of displays.

3. *HOST COMMUNICATIONS SOFTWARE*

This section provides information on the communications software based in the UTS kernel.

General host communications software is provided in the following areas:

- Host resident code to support full-duplex devices. In addition to the host code, full-duplex ASCII support software is installed in the Amdahl 4705 Communications Processor by using the Amdahl UTS/F program product.
- uucp(1C) and cu(1C), which provide the capability to interact with a remote UNIX system (which may or may not be UTS) to transfer files or execute commands.
- Remote Job Entry (RJE) support over binary synchronous protocol communication lines.

The following diagram illustrates components of a mainframe solution for UNIX:

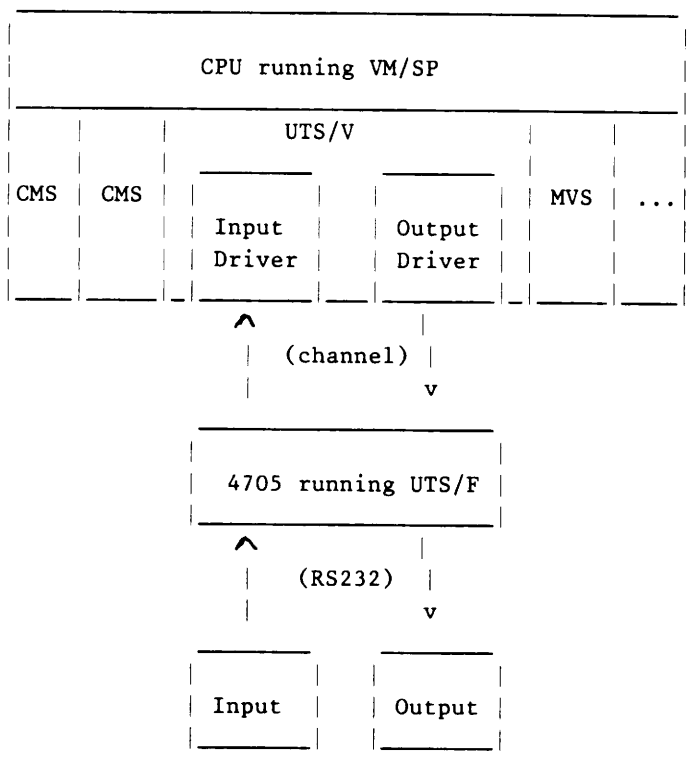


fig.3: Full-Duplex Devices and UTS/V

3.1 Full-Duplex Support Code

Full-duplex support in a UTS system is provided through resident drivers that interface with corresponding packetizing code in the 4705 Communications Processor. While the host drivers are present in the standard UTS system, the optional Amdahl UTS/F program product must be used in the Amdahl 4705 Communications Processor to communicate with these drivers.

3.1.1 Host Driver Interfaces

The full-duplex devices are identified in the host as devices in /dev/.... As with all other devices in the UTS system, they are placed there through the sysgen(1M) process, which uses config(1M) and a special device configuration list to automatically configure UTS.

Within the /etc/devicelist file used by config(1M), all of the terminals related to full-duplex operations are identified as fdxterm devices. Any autocall units are identified as acu devices and communications processors are identified as fep devices. Through this table, the channel/subchannel address of the various devices is established and some device parameters are set.

Further configuration parameters, such as the line speeds, are set in the EP code controlling the communications processor.

3.1.2 Communications Processor Software

The UTS/F product is shipped as an update to a standard Emulation Program (EP), which is software that controls the Amdahl 4705 Communications Processor.

The UTS/F software implements a packet subchannel interface. This subchannel is used by the host for input from all full duplex lines defined on the Amdahl 4705 Communications Processor. The packet subchannel may be any valid emulator subchannel.

For simplicity and ease of operation, an Emulation Program is used as the base for this additional code. Although a public domain version of EP is perfectly acceptable and is in fact shipped with the UTS/F product, an installation may use a program product version of EP to install the UTS/F product.

Packets containing input data from the full duplex lines are sent to the host under three conditions:

- The packet becomes full.
- A specific period of time has elapsed since the last packet was sent to the host and there is some data in the current packet to send.
- Error conditions exist that require host notification.

Two host subchannels are used for each full duplex line: one for output data and another for special error status conditions. These two subchannels are in addition to the single packet subchannel used for all input from full duplex devices.

Auto-call units (AT&T 801-compatible) are supported so that software in the host can place an outgoing call to another UNIX system.

To install this support, extensions to the normal EP generation macros are provided to allow the inclusion of Amdahl-developed full duplex support code. This extended EP macro library is shipped with the UTS/F product. Through the use of new Amdahl-developed parameters to standard EP macros, special full duplex support code is taken from Amdahl object libraries and included in the Amdahl 4705 Communications Processor load module.

3.2 UUCP and CU Support

Through the use of the full-duplex software described in the previous section, standard uucp(1C) and cu(1C) support is provided to allow a process on UTS to interact with a remote UNIX system. Note that the optional UTS/F program product is required for this support.

This software is not described here since it is common UNIX software.

3.3 Remote Job Entry (RJE) Support

RJE support is an integral portion of a standard UNIX system. Through it, a UNIX system can communicate with an IBM Job Entry Subsystem (JES) by mimicking an IBM 360 multileaving workstation.

In several popular minicomputer implementations of UNIX, a special microprocessor board is installed in the minicomputer to support communication protocols. This microprocessor is programmed from a script generated in the UNIX system; through this, much of the tedious communications processing is offloaded from the central UNIX processor.

In the version of UNIX distributed by AT&T, the "virtual protocol machine" interface is used as the model for this offloading of scripts to an auxiliary communications processor.

Within UTS, there is no associated microprocessor card. The virtual protocol scripts to handle rje lines have been incorporated in the kernel. The central processor executes this code by interacting with an Amdahl 4705 Communications Processor, which actually manages the binary synchronous line that is connected to a remote JES.

In programming the 4705, a binary synchronous line is generated in EP and made available as a subchannel address to the host processor executing UTS. The optional UTS/F program product is not required for this support.

4. UTS NETWORKING INTERFACES

The prior sections of this paper have discussed in detail the UTS support for standard UNIX communications where a front-end processor is used to interface an RS-232 style device to the mainframe. In many situations, networks are created using switches and telephone links based on this technology - some examples are NETNEWS in the United States and proprietary switches to interconnect a large number of terminals with a large number of UNIX processors.

Of increasing interest are the X.25, Ethernet and Hyperchannel networks, which are discussed in this section.

4.1 X.25 Network Interfacing

The current UTS/V product interfaces to an X.25 network through the use of the Comm-Pro X25 Network Access Support Package. †

This product is installed in exactly the same fashion as UTS/F is installed: additional code to a normal EP gen for the 4705 or 3705. It allows asynchronous interactive terminals, 3270 compatible terminals and BSC contention terminals (2780/3780) to access UTS through any packet switched network that uses X.25 link access procedures.

However, it is crucial to note that there is currently no distinction to UTS between a device connected through the network and a half-duplex (eg,tty) device directly attached to UTS. As far as UTS is concerned, it is talking to a common tty-type device and is not aware that the device is actually reached through a network. The PAD functions are handled entirely within the 4705.

This level of support is perfectly adequate for applications on UTS that are not concerned with the X.25 interface. In order to support applications that must recognize X.25 functions (particularly X3, X28 and X29 protocols), work is underway at Amdahl to provide these interfaces within UTS as well as a method of placing a call into the X.25 network (eg, generating a call request packet).

This current method of X.25 support does not require nor does it interface with the UTS/F software product.

4.2 Ethernet Network Interfacing

Another popular networking media is ethernet; however, it is new to the mainframe environment and hardware/software products are just now being developed for this class of processors.

Several companies market a channel-to-ethernet hardware device that can be used with UTS. In addition, there are several instances of software products that will interface such a device to a VM or other host operating system.

Work is underway at this time to interface such a device to UTS so that a basic internet protocol [RFC-791] is provided at the level 3, or network layer. Above this protocol will be both the Transmission Control Protocol [RFC-793], which performs validation checks on data, and the Universal Datagram Protocol [RFC-768], which does not perform validation checks.

Above the TCP layer will be such protocols as electronic mail [RFC-821], file transfer [RFC-765] and remote login support [RFC-764]. Through these functions, users will be able to connect to UTS over ethernet from a workstation, and UTS will be able to participate fully in existing ethernet networks.

---

† Comm-Pro Associates, 121 West Torrance Blvd, Redondo Beach, California, 90277.

4.3 Hyperchannel Network Interfacing

A third popular method of networking uses the Hyperchannel, † which can interface machines from different architectures through the use of a common bus and special hardware to interface to the bus. For example, a device exists to interface a standard ibm-style channel to the bus as well as another device to interface a DEC processor to the bus.

A common protocol (NETEX) is used to interface the various architectures; work is underway to provide this interface within UTS. Through it, mainframes of varying architectures will be able to communicate easily since many differing operating systems are supported.

This solution will be particularly attractive to installations that already have an installed base of mini and micro-processors yet want to bring in mainframe power while still operating the existing processors. The NSC family of communications products also supports satellite communications and remote devices (eg, remoted printers, etc).

5. CONCLUSION

This paper has examined in detail the major technical problems found in bringing full-duplex ASCII support to a mainframe environment that historically has not supported such a device. The UTS/F program product, which provides this support has been described, and several examples have been given of current work in network interfacing.

Through all these activities, it is now possible to provide the software portability and functionality of UNIX in the mainframe environment such that the end-user has an effective tool for creating and accessing corporate applications.

---

† Network Systems Corporation, 7600 Boone Avenue North, Brooklyn Park, Minnesota, 55428.

COSAC X400 NETWORK

AOUT 1985

C.Kintzig

CNET PAA/RDS/RVA
38 rue Gl Leclerc
92131 Issy-les-moulineaux
FRANCE

27/8/1985

- COSAC is a trade mark of CNET
- UNIX is a trade mark of Bell Labs

1. *PRESENTATION OF MHS-X.400*

CCITT has voted in October 1984 the X.400 series of recommendations. They define models, services and protocols allowing users to exchange electronic mail. The following presentation is an excerpt from the X.400 document, which may be referred to for further details.

1.1. DESCRIPTION OF THE MHS MODEL

1.1.1. Overview

A fonctionnal view of the MHS Model is shown in Figure 1. In this model, a user is either a person or a computer application. A user is referred to as either an originator (when sending a message) or a recipient (when receiving one). MH Service elements define the set of message types and the capabilities that enable an orinator to transfert messages of those types to one or more recipients.

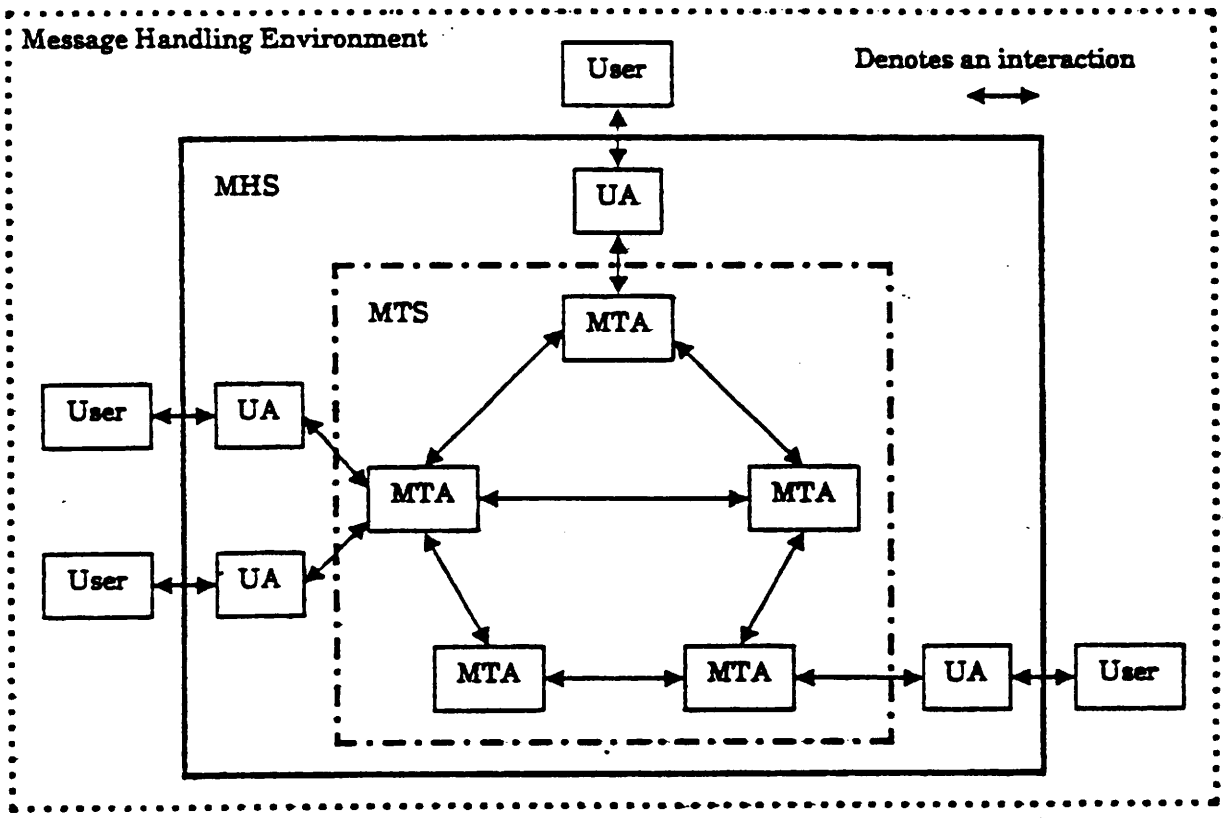


Figure 1 . Functional View of the MHS Model

An originator prepares messages with the assistance of his User Agent. A User Agent (UA) is an application process that interacts with the Message Transfert System (MTS) to submit messages. The MTS delivers to one or more recipient UAs the messages submitted to it. Functions perfomed solely by the UA and not standardized as part of the MH Service elements are called local UA fuAAA functions.

The MTS comprises a number of Message Transfer Agents (MTAs).Operating together, the MTAs relay messages and deliver them to the intended recipient UAs, which then make the messages available to the intended recipients.

The collection of UAs and MTAs is called the Message Handling System (MHS). The MHS and all of its users are collectively referred to as the Message Handling Environment.

The basic structure of messages is shown in Figure 2. The envelope carries information to be used when transferring the message. The content is the piece of information that the originating UA wishes delivered to one or more recipient UAs.

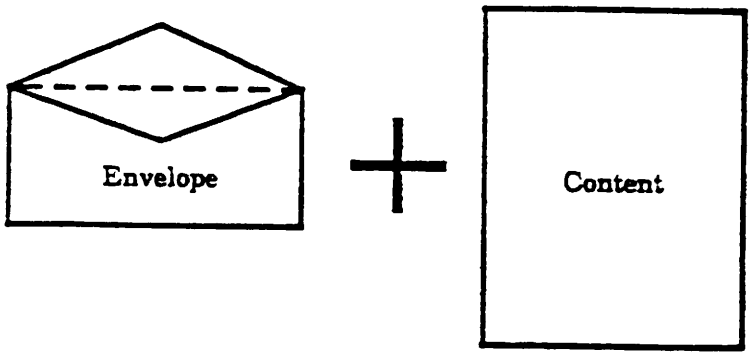


Figure 2 Basic Message Structure

1.1.2. Layered Description of the Message Handling System.

The Message Handling entities and protocols are located in the Application Layer (No 7) of the OSI Reference Model. This is done to permit the MH applications to use the underlying layers.

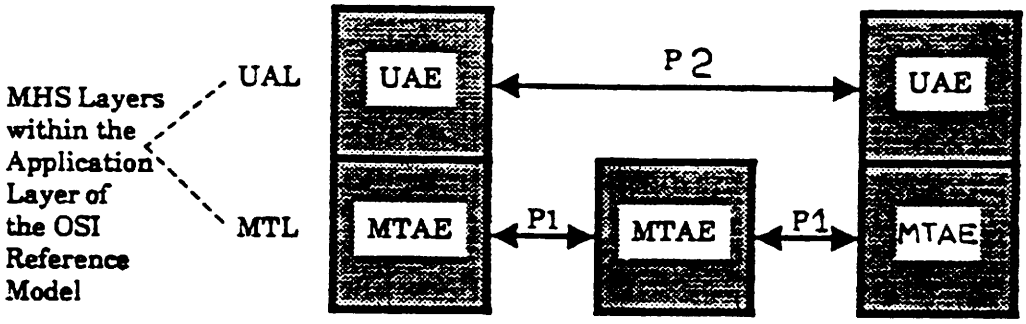


Figure : 3 Layered Description

1.1.3. The Message Transfert System

The MTS provides the general, application-independent, store-and-forward message transfert service.

1.1.3.1. Submission and Delivery

The MTS provides the means by which UAs can exchange messages. There are two basic interactions between MTAs and UAs.

1. The submission interaction is the means by which an originating UA transfers to an MTA the content of a message plus the submission envelope.

2. The delivery interaction is the means by which the MTA transfers to a recipient UA the content of a message plus the delivery envelope.

1.1.4. Relaying

Each MTA relays the message to another MTA until the message reaches the the recipient's MTA, which then delivers it to the recipient UA using the delivery interaction.

1.1.4.1. The User Agent

The UA uses the MT Service provided by the MTS. A UA is a set of computer application processes that, as a minimum, contain the functions necessary to interact with the MTS using the submission

and delivery procedures.

The IPM UA (Interpersonal Messaging UA) :

1. Provide the functions necessary to prepare messages.

2. Perform the submission interaction with the MTS.

3. Perform the delivery interaction with the MTS.

4. Perform the functions necessary to present messages to its user.

5. Provide functions to cooperate with other UAs in order to help its user deal with messages.

6. Perform additional message preparation and manipulation functions.

Optionally, the IPM UA may provide local UA functions not subject to standardization by CCITT. For example, it might provide word processing facilities, or database facilities for storing and retrieving previously received messages.

1.2. CONSTRUCTION OF O/R NAMES

It is an objective that an originator be able to provide a descriptive name for each recipient of a message using information commonly known about that user. This Recommendation specifies a set of standart attributes from which these O/R names can be construted.

1.2.1. Variantes of O/R Names.

An O/R Name may be of the four following variant.

Note:Attributes enclosed in square brackets are optional

Variant 1

O/R name consist of:
Contry name
Administration domain name
[Private domain name]
[Personal name]
[Organization Name]
[Organization Unit Name]
[Domain-defined attributes]

Note:At least one of Private Domain Name, Personal Name, Organization Name and Organization Unit Names must be selected.

Variant 2

O/R name consists of:
Contry name
Administration domain name
UA unique numeric identifier
[Domain-defined attributes]

Variant 3

O/R name consist of:
Contry name
Administration domain name
X.121 Address
[Domain-defined attributes]

Variant 4

O/R name consist of:
Telematic Address
[Telematic Terminal Identifier]

This form comprises the X.121 address and, optionally, a telematic terminal identifier.

1.3. SERVICE ELEMENTS

1.3.1. The Message Transfer Service

The features of the MT Service are listed in Table 1

Service Group	Service Elements
Basic	Access Management Content Type Indication Converted Indication Delivery Time Stamp Indication Message Identification Non-delivery Notification Original Encoded Information types Indication Registred Encodad Information Types Submission Time Stamp Indication
Submission and Delivery	Alternate Recipient Allowed Deferred Delivery Deffered Delivery Cancellation Delivery Notification Disclosure of Other Recipients Grade of Delivery Selection Multi-destination Delivery Prevention of Non-delivery Notification Return of Contents
Conversion	Conversion Prohibition Explicit Conversion Implicit Conversion
Query	Probe
Status and Inform	Alternate Recipient Assignment Hold for Delivery

Table 1. Message Transfer Service Elements

1.4. THE INTERPERSONAL MESSAGING SERVICE

The features of the IPM Service are listed in Table 2

Service Group	Service Elements
Basic	Basic MT Service Elements IP-message Identification Typed Body
Submission and Delivery and Conversion (MT service elements)	See Table 1
Coopering IPM UA Action	Blind Copy Recipient Indication Non-receipt Notification Receipt Notification Auto-forwarded Indication
Cooperting IPM UA Information Conveying	Originator Indication Authorizing Users Indication Primary and Copy Recipients Indication Expiry Date Indication Cross-referencing Indication Importance Indication Obsoleting Indication Sensitivety Indication Subject Indication Replying IP-message Indication Repy Request Indication Forwarded IP-message Indication Body Part Encryption Indication Multi-part Body
Query (MT service elements)	See Table 1
Status and Inform (MT service elements)	See Table 1

Table2 Interpersonal Messaging Service Elements

1.5. PROTOCOLS

The P1 protocol is used to exchange messages between MTA entities. The P2 protocol is used to exchange information between UA entities.

1.5.1. Principles

The P1 and P2 protocols are described in terms of the X.409 syntax. This syntax uses a BNF notation to describe protocol elements and octets exchanged between entities. Each protocol element is coded as a triplet (I,L,V) where I is an identifier, L a length and V a value. A protocol element is represented as a tree structure. N Nodes are (I,L,V) and contain (I,L,V). Leaves are (I,L,V). An advantage of this technique is that elements are not limited in length or number.

2. *THE COSAC EXPERIMENTAL NETWORK*

2.1. PRESENTATION

The experimental COSAC network (COmmunication SAns Connexion), which operates in message mode, is being tested at CNET. It allows users to exchange information of any type when there is no need for real time interaction. E.g. mail, file trafert Presently, COSAC offers a mail service. A file transfer service will be available at the end of 1985, and more services are being defined.

Early CCITT work on electronic mail, which matured into the X.400 series, was ta ken as the base for defining COSAC concepts.

2.2. CCITT ELECTRONIC MAIL MODEL

The CCITT model comprises two entities, both in the layer 7 of the OSI model. Th ese entities are:

- MTA : Message Transfer Agent (See 1)

- UA : User Agent (See 1)

This model was initially used for building the COSAC prototype, and then as a base for the experimental network.

2.3. THE COSAC NETWORK

The COSAC network is composed of computers implementing MTA and UA functions. MTA's handle P1 protocol elements regardless of their content. Thus, such elements could be used as well for file transfer, job transfer, and other applications. A network entity comprises MTA functions and specific US functions, e.g. mail or file transfer.

The functional diagram of this network (with only two MTA's) is represented in the following figure.

```
                Users                                      Users
/-------------------------\                  /-------------------------\

I----I    I----I    I----I                   I----I    I----I    I----I
I EM I    I TF I    I XX I                   I EM I    I TF I    I XX I
I----I    I----I    I----I                   I----I    I----I    I----I
 .          .          .                       .          .          .
   .        .        .                           .        .        .
     .      .      .                               .      .      .
       .    .    .                                   .    .    .
         I-----------I                                I----------I
         I           I                                I          I   P1 I----I
         I    MTA    I.......................P1.......I    MTA   I<....I AD I
         I           I                                I          I      I----I
         I-----------I                                I----------I
```

EM: electronic mail
TF: file transfer UA
XX: other service UA
AD: Adapter

SERVICES OFFERED BY COSAC

COSAC offers MTA functions. As UA functions are by and large provided by existing message systems, it was more efficient to put in COSAC adapters turning those closed systems into "open" systems in the OSI sense. Consequently, the interconnection of heterogeneous message systems is becoming possible. The COSAC software includes adapters for the following message systems:

- UNIX mail
- MULTICS mail
- MISSIVE (alias COMET)

- EUROKOM (alias KOM)

- FORUM

At this time, users of those message systems can exchange mail.

2.4. FILE TRANSFER

File transfer in message mode (TFMM) for the COSAC network has been specified at CNET. Ecole des Mines de St. Etienne is implementing a first version on UNIX (SM-90), which will be ported to other computers of the network (IBM, VAX, MULTICS). Services comprise primarily file send and receive, with multi-destinations, protection and attributes updating. The initial version of the protocol can transfer structured files (text lines, T.73 document structure) or binary files.

2.5. BULLETIN BOARDS INTERCONNECTION

Bulletin boards are similar to mailboxes by topics open for reading and writing to all users of one system. This service (e.g. "continuum" on MULTICS) is centralized. Several bulletin boards on the same topic may be interconnected via COSAC would have to be defined specific UA. In the meantime, for testing the concept, this service is offered by using dedicated mailboxes for passing messages between bulletin boards. Presently, a number of bulletin boards are interconnected.

2.6. PORTING COSAC ON IBM, VAX, MULTICS

The COSAC software is written in PASCAL and runs under UNIX. The experimental network described above comprises only SM-90's. To cater for the needs of the research community, some institutions (INRIA, CNUSC, CNAM, IMAG) have decided to port the COSAC software on their computers. Thus, the future COSAC network will include MULTICS, IBM, VAX and SM-90 computers. The first service will be electronic mail. The second service will be file transfer.

Other services are being defined and include, job submission and bulletin board interconnection.

3. PRESENT STATUS

The current software version is V3. The following mail systems are interconnected:

- UNIX mail

- MULTICS mail

- EUROKOM

- IBM UTS and CMS
- VAX VMS

Bulletin boards on MULTICS and UNIX are interconnected.

The V3 software handles a subset of P1 and P2. E.g.

Ex: address: country
administrative domain
private domain
person name

It does not contain UA as defined in X.400.

Message transfer between COSAC nodes uses asynchronous TTY emulators, PAD to PAD.

3.1. PRESENT DEVELOPMENTS

A new version V5 is under development. It contains MTA and UA functions. The P1 and P2 protocols are handled as defined in X.400, including address structures. It uses BAS session.

New nodes running the V5 version will start operating at the beginning of 1986, with mail and file transfer. They will be interconnected with V3 nodes.

The V5 software will be ported to IBM, VAX and MULTICS mid 1986, and will replace the V3 software.

Simultaneously, working groups are defining a distributed bulletin board service (CRMM) and job submission (STMM). MULTICS and UNIX implementation should become available during 1986.

CONCLUSIONS

The V3 version has allowed an experiment of basic MHS services. The V5 version is designed to offer more services (file transfer, job submission, bulletin board) on a larger set of heterogeneous systems, while putting the OSI model into concrete use.

ANSI C Standard

Mike Banahan

Technical Director
The Instruction Set

1. The ANSI C Standard

1.1 Abstract

This paper discusses, briefly, some salient points relating to the X3J11 C Standard effort. It considers the membership of the committee, then some of the more important changes and additions to the language and preprocessor. It does not even try to be a complete summary of the work of the committee.

1.2 The Committee

The X3J11 committee, like other ANSI committees, is made up of volunteer members who are interested enough to attend the meetings and can find the funds to pay for membership and attendance.

The committee meets quarterly for one week at a time, and currently only meets in North America. Unusually, a meeting is planned to be held in the UK at a future date, but such international mindedness is a rare event.

The bulk of this committee is made up of systems-oriented people. Many compiler suppliers and supporters are represented, with most of the major manufacturers fielding a team member. The remainder of the group is a mixture of users from various fields. The committee is split into three groups which are independently chaired: the language group, which considers the C that we know; the libraries group, defining standard libraries and header files - C is useless in an applications environment without libraries; and the environment group, who look at issues such as character sets, I/O translation mechanisms and other stuff that is usually taken for granted in a UNIX environment. The work done by the three groups will be mentioned in the following discussion, but my own interest is mainly in the language area, and that is where most of this paper will focus.

1.3 Committee Aims

The committee is relatively free to choose its own brief. However, wild schemes are tempered by the knowledge that the standard will eventually be voted on by the world at large (I think), and it is certainly at the front of everyone's mind that there is no point in defining a standard nobody wants to use.

The effect is that the committee feels bound to keep to the original spirit of C. They are prepared to introduce limited enhancements where a consensus feels that it would be a good idea, but the bulk of the work is aimed at reducing ambiguity and tightening the existing definition of the language, together with a well defined library set and I/O model.

It has certainly been a major influence that nobody wants to break existing programs unless they were "already broken"; for example by tightening a specification, there is bound to be a body of code that made use of a lax interpretation of the previous specification. Such code may no longer work.

1.4 Conformance

According to the standard, a conforming implementation of C must provide the mandatory syntax of the language, provide a definition of the way that it implements certain operations (for example sign-extend when right shifting an **int**.) Some things fall into the category of undefined behaviour - such as divide by zero - and the implementation can do what it wants in these cases; ranging from aborting the program to continuing with some unspecified value.

Associated with the idea of a conforming implementation, is the idea of conforming programs. There are actually three classes of program: Erroneous, which use undefined behaviour at some time; Conforming, which use only defined behaviour; Non-portable, which use both fully defined and implementation defined behaviour.

Implementations (by which is meant a compilation system or interpreter) are obliged to report syntax errors or the violation of a constraint, such as trying to declare an array of bitfields - a declaration permitted by the grammar of the language, but prohibited by a constraint.

1.5 The Language

In the following sections there is no attempt to reproduce the new language reference manual. If you want one of those, then send a cheque for US$ 20.- (requesting a copy of the 'C Language Information Bulletin', and stating your address clearly), to

X3 Secretariat, CBEMA,
311 First Street, NW,
Suite 500,
Washington, DC 20001.
U.S.A.
+1-202-737-8888

The reference manual has been completely rewritten and now gives a full grammar for the language (but there are quite a lot of rules), so YACC hackers can wander off and have a good time. Amazingly, the same has been done for the preprocessor too - though the grammars are not compatible - so your YACC will either need some systematic renaming or stuffing through a pipe.

The following will simply touch on interesting topics, give a flavour of what has been done, and leave you to refer to the full standard for more information.

1.5.1 External Declarations

This is a rich area; C has driven a coach and horses through traditional block structure by permitting appalling declarations like this:

```
static x;

f(){
        {
                extern x;       /* ho ho */
        }
}
```

What does that do? Even better, what is the scope of **x** if the **static x;** declaration is dropped? There are about four schools of thought, none of them compatible. There are now rules about that stuff, and a good thing too. Any code relying on that working before was written by a lunatic anyhow.

1.5.2 Identifiers

Up to 31 chars in internal identifiers, more may be used but not guaranteed to be checked.

Only 6 monocase characters in externals - blame DEC linkers (and others). This one issue absorbed more committee time than is spent drinking at an EUUG meeting.

1.5.3 Types

A sprinkling of new types have appeared, plus a new keyword **signed**. You can now declare **signed char**, **unsigned char** and **char**. The difference is that **signed char** guarantees to sign-extend on promotion to a longer type. There is also the interesting **long double** which is even longer than **double** for extra precision. This smacks of Algol-68 if I'm not mistaken, but isn't extendable to **long long double** etc.

1.5.4 Storage classes

Two new keywords: **const** and **volatile**. These are *not* opposites of each other, they are completely orthogonal. The aim of **const** is to allow storage of something in ROM: a **const** object may not be used as an lvalue.

To permit a whole class of optimisations not previously safe with C, it is now permissible to remove apparent loop-invariants out of loops. If they aren't really invariant - updated by interrupts perhaps - they should be declared to be **volatile**. Here are some examples:

```
const char s[]="hello";
const char *p;
const char *const cp = "hello"
volatile int *vp;
const volatile int *const timer = RTCLKADDR;
```

Strings are now of type **const char** and may not be modified. They are not guaranteed to be distinct any more, either. This should have very little impact on well-written code.

1.5.5 Void

This has been in UNIX for ages, but is now blessed. The rather odd type (**void** *) is now the universal pointer, and pointer-to-anything can be assigned to it and back both safely and without requiring an explicit cast: see below (which also uses a new feature of function declaration explained later).

```
void *malloc(size_t);

f(){
        int *p;
        p = malloc(sizeof(int));
}
```

1.5.6 Odds and ends

Arithmetic on purely **float** and shorter expressions is not forced to evaluate at **double** precision.

The misguided **enum** type is now officially **int** and that's the end of it. A shame it wasn't thrown out all together.

The separate name space of structure members is now blessed - though not many people can remember when this wasn't true.

The restriction that only static or external aggregates can be initialised has been lifted.

1.5.7 Function prototypes

We can now declare the number and types of arguments that a function takes. This is mainly courtesy of C++, so there is an existence proof that it can be made to work: essential for new ideas that are made a standard without testing (see the Ada tasking...). There is one constraint - if a function prototype is ever visible at the use or definition of a function, then an identical prototype *must* be visible at *every* use or definition. Examples:

```
double sqrt();
No argument specified

double sqrt(double arg);

double sqrt(double);

means that sqrt takes only one argument of type double

#include <math.h>

int i = sqrt(4);
```

Try passing 4 as the argument to current implementations of **sqrt**.

Also, the use of "..." meaning "unspecified arguments follow" together with a portable version of **varargs.h** allows a completely portable version of **printf** to be written.

```
#include <varargs.h>

printf(const char *fp, ... ){
```

The use of prototypes permits more optimisation:

```
int x(int, float);
```

allows a different calling sequence to be used and allows the passing of **float** arguments without forcing widening.

Function definitions can now be made to look like the prototype, since the declaration of formal parameters has been made to extend into the first block:

```
f(float f_arg, register i_arg){
```

1.6 Libraries

A whole bunch of standard library stuff has been incorporated, especially the stdio library. Most of this is based on the /usr/group standard, although there is some debate as to who should be allowed to define which routines. One useful thing is the guarantee that anything implemented as a macro will also exist as a function (so its address can be taken). All names starting with "_" have been reserved for library use, the same goes for macros starting "__".

There is also the concept of the freestanding environment - one without libraries - where the names are not reserved.

The libraries come with a set of header files containing important declarations, and some environmental/implementation enquiry facilities.

Though glossed over here, the library definitions may turn out to be the biggest single aid to portability in the whole effort.

1.7 I/O

The simple-minded UNIX view of files is unfortunately complicated by the real world and the stupidity of operating system designers who didn't exhibit the genius/naievity† of Ritchie when he designed the UNIX I/O mechanism. This has forced two file types onto the standard I/O library.

1.7.1 Text files

Basically, these may have to cope with translating \r\n to \n on input and vice versa on output. Even more bizarre convolutions are bound to be needed on some systems. (I shudder to think what would happen on CDC NOS files that are not in display format. But that's their problem.) The idea is that you still write code like before, but the library has to make the I/O look right to the underlying system. A restriction is imposed that this will only work on lines terminated by \n and less than 255 characters long, consisting of only printable characters, space, tab and form feed. Subject to that, if you write stuff out, it comes back looking the same.

1.7.2 Binary Files

These suffer no translation at all, but if you write text into a file, it may not actually look right if it is sent direct to a printer. Furthermore, since some systems don't even know how long their files are, binary files may contain an unspecified number of null characters beyond the high-water mark, and append mode may not work quite the way that you expected.

This whole area is tacky.

1.8 Preprocessor

Be warned - if you ever used more than

```
#define
#ifdef
#ifndef
#else
#endif
#if
```

---

genius/naievity Strike out which you think doesn't apply

and fancy tricks like name replacement in strings, then you have a rewrite to do.

The preprocessor has been defined in terms of a formal grammar and has been tightened up a lot. Techniques exist for building strings and concatenating tokens, as shown below (NB adjacent strings now merge):

```
#define debug(s, t) printf("x"#s"= %d, x" # t\
        "= %s", x ## s, x##t)
debug(1,2)
```

gives

```
printf("x""1""= %d, x""2""= %s", x1, x2)
```

then giving

```
printf("x1= %d, x2= %s", x1, x2)
```

1.9 Effects

What are the effects of all this? Well, it should considerably improve the portability of C programs, and allow for more efficient code generation.

The minus side is that the new keywords are bound to break some existing code. The preprocessor changes won't damage anything that wasn't portable anyway.

The X/OPEN Portability Guide

Jacques Febvre, Honeywell Bull
Mike Banahan, The Instruction Set

1. Introduction

The X/OPEN Developers' Guide to Portability is a recently published collection of definitions, intended to document a stable and substantial UNIX distribution. It is intended to be used as a basis for applications developers who wish to write applications software but are not sure what systems they should aim at. Current UNIX machines do not offer the full range of services needed for applications software. This reduces the confidence of applications writers who must have a large and stable market segment to aim at.

The X/OPEN document solves this problem by selecting industry standards and bringing them together under one common umbrella. This, plus a committment from X/OPEN member companies to support such an environment on a large number of their systems, will provide the software market with the sort of targets that they need. The end-user benefits from having a large number of portable applications packages available, and by being able to pick suppliers on criteria other than those imposed by the necessity to run one particular package.

The commercial arguments have been covered in other discussions. This paper describes the contents of the Portability Guide from a technical point of view.

1.1 Contents of the Guide

The Guide contains 6 sections, listed below.

- Introduction
- System V specification (system calls, libraries header files etc.)
- The C language definition.
- ISAM package
- Other programming languages, especially Fortran and Cobol.
- Source code transfer devices

The following sections describe the contents of the Guide, starting with the System V Specification.

1.2 System V specification

There are two contenders for the title of "UNIX Standard": the System V Interface Definition (SVID), from AT&T, and the forthcoming /usr/group standard. They are in fact very closely compatible, and the differences between them extremely small. X/OPEN chose, for commercial reasons, to ensure that the Interface Specification in the Guide should be compatible with the SVID. That is to say, that applications written to conform to the SVID should also be able to run on an X/OPEN system when recompiled. AT&T are also developing a verification suite of software which will check that a given system conforms to the SVID. The specification in the Guide was chosen so that the verification suite would be able to check, and agree, that X/OPEN systems also conform to the SVID.

Because the X/OPEN aim is to provide an environment for applications developers, the Guide is not an exact copy of the SVID. The reasons are explained below.

The first reason is simply one of usability. The SVID is organised as a Base and a number of extensions. The Guide has made its choice of the Base and Extensions, and (except for some slight changes required as a result of its choice), collects all of the SVID documentation, unchanged, into the X/OPEN Specification. There is no question of extensions, so the structure of the SVID is not appropriate. Instead, the X/OPEN specification is organised as manual pages which will be recognised by anyone familiar with traditional UNIX documentation. They are still presented as system calls, library routines and so on, in separate sections, but a comprehensive index is also supplied giving the name of every entry point and the manual page where it is described. This reorganisation is expected to make the document usable by programmers and designers on a day by day basis. The SVID is more of a reference document for system implementors.

The second reason is one of utility. The Guide has chosen to present a superset of the SVID functionality which more closely represents that found in current products. As a result, a number of library routines (in particular) not described in the SVID are to be found in the X/OPEN Specification. It was felt that their presence would be beneficial to a large number of applications developers.

The third reason is simply one of timing. The X/OPEN document was prepared later than the AT&T one. The chance has been taken to correct a small number of errors in the SVID and to adopt a number of "future directions" mentioned in the SVID. Most of these are simply the increased use of symbolic names, rather than absolute constants.

Finally some of the items described in the SVID as being parts of extensions have been brought into the Guide but marked optional. An example is *ptrace*, which is part of the SVID Kernel_Extension set. It is part of the X/OPEN Specification, but marked optional because not all systems may be able to support it. Its presence in the Guide means that if it *is* present in a system, then it will conform to the published specification. This prevents the name being used for other purposes and causing confusion.

It is important to realise that on every page in the specification, the differences between the X/OPEN Specification and the SVID have been noted in a special section. This means that applications developers can check to make sure that they are not using features in the X/OPEN system which would cause incompatibility with AT&T System V. The functions in the X/OPEN Specification which are not in the SVID are:

getgrent	brk
getlogin	cuserid
getpass	ecvt
getpwent	end
getut	getpw
l3tol	logname
monitor	ttyslot
putpwent	

1.3 C

As it is expected that a large number of applications will be written in C, the language is defined in full in the X/OPEN Guide by reprinting the AT&T System V.2 definition of the language. There are also a number of guidelines on portability given as notes to C programmers who want to make their programs as portable as possible.

Obviously, complete portability is hard to acheive and no guidelines will cover all cases. As the Guide says, the portability guidelines will be expected to grow in later releases of the document.

X/OPEN member companies are represented on the ANSI X3J11 C standard committee, and the group has echoed AT&T's committment to adopt that standard at an appropriate time. In the meantime, a number of guidelines are given on how to avoid incompatibilities between code written now and the eventual standard for the language. It is currently possible to write syntactically correct programs which will not compile under the proposed standard, because of the tightening up of some ambiguities in the language and the introduction of some new keywords. The new keywords are bound to cause problems to some existing code which has used the same words as identifiers.

1.4 Other Languages

1.4.1 COBOL

For large commercial applications, C is not the language in most common use. This is the province of COBOL, and it is unrealistic to expect applications developers to change to C overnight, or ever. For this reason, X/OPEN has chosen to specify MicroFocus Level II COBOL as the definition of the X/OPEN COBOL.

It is important to note that it is not the MicroFocus *product*, that has been specified; it is perfectly possible for some manufacturers to write their own compilers as long as the implementation provides the same *functionality* as the MicroFocus product. It can probably be reasonably expected that at least some of the X/OPEN manufacturers will choose to use the MicroFocus product; conversely we can also expect some to modify their existing compilers to conform.

The specification adheres relatively closely to ANSI standard X3.23, 1974, with extensions in the area of interactive processing. A full definition of COBOL is not given in the Guide, but a set of syntax diagrams shows the language that is supported. Interested readers should refer to the Portability Guide for full details.

1.4.2 FORTRAN

The standard for FORTRAN is FORTRAN 77, ANSI X3.9-1978. There are many certified compilers available, so X/OPEN has not felt it necessary to specify a particular product as the definitive implementation.

1.5 Data Management

As well as needing COBOL for commercial applications, it is also recognised that the file-handling primitives in UNIX are not at the level needed by application writers. To provide a higher-level view of file handling, an ISAM package has been defined. This is based on the RDS C-ISAM specification, with minor modifications to the documentation to delete references to implementation details. Like the COBOL, this is viewed as the definition of a set of interfaces, not a specific implementation.

The use of a standard package such as this is intended to increase the ease with which different applications can be integrated, and to encourage the use of data formats which will allow the data to be shared amongst applications.

1.6 Source Code Transfer

A considerable problem in actually porting software between machines is the difficulty of getting the source code from one machine to another one. For this reason, a definition of media and archive formats is given. The definitions are for 5¼ inch floppy disk and 9 track tape. Standard device names are defined for each of the variants of these, for example the preferred magnetic tape device is */dev/sctmtm*, which is phase encoded 1600 bpi, 512 bytes per record.

These devices are not necessarily present on every system sold - the specification is for systems which will be used in the porting of applications, not for every run-time system. Neither are these devices considered from the point of view of their suitability for file system back-up, although no doubt the 9 track tape device would probably also be used for this purpose if a machine has one.

It was not possible to define a cartridge tape format: there are too many conflicting standards in common use already.

The *tar* and *cpio* tools are the recommended ones for creating archives to be transferred between machines, with *cpio -c* being preferred from the point of view of portability.

1.7 Where to get the Guide

The X/OPEN Portability Guide is published by Elsevier Science Publishers B.V., P.O. Box 1991, 1000 BZ, Amsterdam, The Netherlands. It will also be for sale in various specialist bookshops ($75.-) Happy reading!

Development methods for high performance Commercial Unix systems

Paper presented at the EUUG autumn Conference
Copenhagen 11/9/1985

P.J.Cameron

Plessey Microsystems Ltd.
Water Lane, Towcester
Northants, NN12 7JN
England

Phone: Towcester (0327)50312
Eunet: mcvax!ukc!qtlon!pmllab!pjc

Introduction.

Within the computer industry, design teams are traditionally split into seperate hardware and software groups. Consequently hardware designers have given little or no consideration to the subsequent software integration, leaving the software designer to modify the software to fit, often to the extent of compromising its performance. This has occurred widely in the super-micro field, where the hardware and software are usually produced by different companies.

This paper attempts to show that there is a better way of designing hardware. That there should be a joint development program where software designers have substantial influence on the hardware design. The result of this co-operation is a better system design giving higher performance lower cost systems.

Plessey Microsystems approach.

Approximately two and a half years ago, Plessey Microsystems (a division of the UK. Electronics group Plessey) decided to enter the expanding market for Unix-based microcomputer systems. The decision was taken to initially buy in most of the cards used in the system. This allowed us an early entry into the market and acquisition of a substantial market share in the UK.

Once our leading market position was established, we began to consider product enhancements. Having identified the system components which should be improved, we evaluated available products in the marketplace. We discovered that all available cards either lacked facilities or gave a poor performance when used with Unix.

It was therefore decided that we should design our own card set. This has subsequently proved to be a very sound decision.

As our hardware designers had very little Unix experience, it was necessary for the software designers to take part in the hardware design process. It quickly became apparent that this was a better way of designing hardware, and our standard design philosophy is now based on this approach.

The design process.

For each system component we have a small multi-disciplinary design team comprising hardware designers, software designers and firmware designers. The multi-disciplinary composition of this team being the key to the design process.

We also have three design rules, these are:-

1) All components must be designed to implement completely one facet of the Unix system. The basic design should not be compromised for generality, though if it is generally useful, then this is a bonus.

2) There must be no modifications to the Unix Kernel or any of the utilities, so guaranteeing conformance with established standards (System V).

3) The hardware must be 100% standard bus compatible (Multibus or VMEbus).

In the commercial environment, it is necessary to adhere to defined standards such as Unix System V and Multibus. This gives customers the maximum possible flexibility in the choice of applications and peripherals.

<u>Manufacturers who do not hold to standards defeat the object of using them.</u>

The entire design process is now the reverse of that conventially used. Firstly it is necessary to look at the software interface rather than considering a hardware specification. This involves looking at the appropriate user program interface definition as described in the Unix manual, and the corresponding Unix kernel interface. From this a software interface that best matches that given by the Unix definitions is produced, this is the first approximation to the final design. At this stage, it is also necessary to look at existing hardware and associated software to see how well they have been integrated. This is often the most enlightening part of the design process, as it can show exactly how not to design the hardware.

Next it is necessary to look at the Unix kernel to see how efficiently it handles the task that is being considered. Surprising as it may seem to some members of the Unix community, some areas of the kernel use techniques unsuitable for todays requirements. In particular the handling of the c-lists (buffering and character translation routines) in System V is very inefficient (we are not the first to notice this, Berkeley changed this completely in one of their first distributions). Accurate kernel performance evaluation on systems running real applications is the key to this analysis, and here the multi-disciplinary approach again pays dividends, as oscilloscopes and logic analysers can quickly provide much more information than simple kernel profiling.

From the information obtained, it is possible to produce a software (device driver) interface specification. It is at this point, that the greatest care must be exercised; this specification is the key to the whole project. If this is correct then the product will be good, if it is poor, the final product will also be poor. Thus it is important that there is good interchange of ideas between the members of the design team, to ensure that the process of refining the initial design ideas produces the best possible design specification. Because of the importance of this stage of the design, it often takes longer than the actual design of the final piece of hardware and software. It is well worth the effort. It is also necessary to keep the whole system in mind at this stage of the design to ensure that the operation of one part does not interfere with another.

The next task is to produce a specification of the hardware to firmware interface (if there is any firmware). This is usually much easier than the software interface specification, as most of the important decisions have already been made.

The final stage of the design process is to produce the hardware, firmware and software. Because there is a firm specification of each interface, these are relatively simple tasks, and can be done simultaneously.

Example 1: The Plessey Inteligent I/O Co-processor.

The first product on which this design process was used was the Plessey Intelligent I/O processor. This is a processor controlled eight channel serial I/O card, with associated firmware and device driver.

By looking at the stages in the design of this card, it is possible to see how the design method was used. Firstly the non-intelligent I/O card that had been used in our first system was studied. This showed us several of the main areas that were in need of improvement.

Our first analysis showed that for a typical systems, output exceeds input by a factor of between 30 and 100 times. Consequently, it is much more important to optimise the output side of serial I/O than the input side.

It was also found that most output was passed to the device driver not as single characters, but as large blocks of characters. These blocks are typically 512 bytes, or one line of output. However, as the number of single character blocks was similar to the number of large blocks, poor handling of these would also seriously degrade system performance. The conclusion was that it was more important to optimise multi character output, but that it was also necessary to ensure that single character output was reasonably efficient.

Timings of the various stages in sending characters from a process to the I/O card were taken. It was found that when outputing single characters, 52% of the total time was taken up by the c-list routines and 35% handling the interrupt. For 512 character blocks, these figures were 14% and 85% respectively. Thus these two areas were those that needed optimisation.

Using these results, the software and firmware interfaces were designed. It was relatively easy to design a system that allowed both multi-character and single-character I/O to be efficient, by ensuring that the firmware on the card had circular buffers for output.

The problem of the overhead on c-list processing was more difficult. Because no modifications to the kernel were allowed, it was not possible to optimise the code. Thus the bold decision was made to re-code the whole of the output c-list handling in firmware. Because of this, different algorithms could be used which were tailored to our design and did not have to be portable, making the code much more efficient. This solution retained the full functionality of c-lists, so ensuring that the software interface still conforms to the System V standard.

The next problem, that of the interrupt overhead, had only one obvious solution: avoid interrupts! Careful design of the interface eliminated interrupts on output in all but the most heavily loaded situations. In the worst possible case, each channel generates one interrupt per thousand characters (ie. one per second at 9600 baud).

The input side of the system was also optimised. The c-lists are still used, though incoming x-on x-off sequences are processed in the firmware. Input interrupts have been optimised by restricting them to twenty per second. This moves some of the processing work onto the device driver, as it must deal with a block of characters on receipt of each interrupt, rather than one character per interrupt. The figure of twenty interupts per second was chosen as optimal, since it reduces the number of interrupts to a manageable level, without causing an undue delay (a 1/20 second delay is not noticeable).

The net result is that the main processor overhead on output has been reduced to as little as 0.009ms per character for 512 character blocks (as against .71ms for the non-intelligent card). The card will cope with all eight channels sending output at 9600 baud without slowing the output on any channel. All channels being driven at 19200 baud should be possible, when terminals that can properly support this become available.

Example 2: The Plessey Cache Processor Card.

The second design example is the Plessey Cache Processor Card. The design brief for this was to make a cpu card that would allow a 12.5MHz 68000/68010 to run as near to the maximum 12.5MHz (the fastest available) as possible.

The first investigation centred on the best way of accessing memory to minimise the number of wait states required by the card. Because standard bus structures were used, it was not possible to access memory directly at this speed. Therefore it was necessary to look at alternatives, and consider how they would fit into the memory usage of a Unix system. Three approaches were examined:

1) A local memory bus. This was the most obvious approach as it allowed the use of large amounts of high speed memory. However, the memory would have to be dual-ported onto the main system bus to allow the DMA transfers that Unix requires for fast disk operation. The use of dual ported memory would have increased the cost of the system and, with arbitration time, would not have been much faster than standard memory on the system bus. This approach was therefore rejected.

2) Local memory. This was another attractive approach as it potentially allowed zero wait state memory. However the amount of on-card memory would be limited and almost all of this memory would be used by the kernel. User programs would have to reside in slow memory on the main bus. This approach was also rejected.

3) A memory cache. This was initially the least favoured approach since it was the most complicated and potentially expensive. However, from tests it seemed likely that this would give the highest performance of any approach and would enable the use of standard low cost memory throughout. This was the approach adopted.

Cache architecture can be expensive both in terms of board area and cost. Studies showed that an 8k byte tagged cache would be a feasible size. It was now necessary to decide whether to use a single 8K cache, or to subdivide it into several parallel cache sets. Analysis of Unix and application software showed that a single set cache would not be very efficient, and that a multi-set cache would be better. A two set cache with 4K bytes per set appeared to give potential hit rates of 80% to 90%, more sets would have required more board area than was available. Hence two 4K byte sets were used. When the prototype had been built, experiments with different cache arrangements were carried out, and the original choice confirmed as the best.

Next the hardware facilities needed to support Unix were considered. The most important requirement is a heartbeat timer. This must be accurate, and at 50 Hz to 60 Hz; this was provided. Some Unix systems have a Floating Point arithmetic unit on the cpu card. As the system was being designed primarily for non-technical customers, it was decided not to include this.

Unix System V has the option of automatically starting up in multi-user mode. However, whilst doing this, it will ask for the time and date. In order to allow a truly turnkey system to be made, the cpu card was specified with a battery backed up clock from which the Unix clock could be automatically initialised.

Having made these design decisions, it was possible to write the design specification for the cpu card. This was now a simple task, since the system architecture was fully defined. The hardware, firmware and software development could procede. The actual hardware design proved a little more difficult, since a large number of components had to be fitted onto the board. When this was finally achieved, the board performance proved to be outstanding.

The card has a measured average cache hit rate of 92%, giving an average of 0.24 wait states on each memory read cycle using low cost memories with full error correction. This card has been independently benchmarked, and found to have better performance than a VAX 11/750 for non-floating point applications. Because of the cache design, the processor also provides optimal performance from a standard bus based system by reducing the processor demands on limited bandwith bus structures, leaving this free as a disk I/O bus.

Conclusion.

Both of the cards given as examples are now in production, and form part of the Plessey Microsystems Mantra Unix based system. These cards are also being sold to other manufacturers both in the UK. and Europe for inclusion in their own Unix based products. Because of the way they were designed, these cards are sold as a package, the I/O co-processor comes complete with the firmware and a device driver, and the Cache Processor card with a complete Unix System V port.

Viewing system components from both the software and hardware angles gives Plessey Microsystems the ability to produce low cost, high performance, versatile, Unix-based computer systems. These are excellent both as software development machines and as general-purpose business machines. They are not only very fast, but support a full standard version of Unix System V and standard bus structures, and hence maintain compatibility with the widest range of other systems and components.

Address list of speakers that submitted papers
for these proceedings:

Ms. Miriam Amos
Digital Equipment Corp
Ultrix Engineering Group, Berkeley Division
P.O. Box 40220
Berkeley, Ca., 94704, U.S.A.
+1 415 643 6448
mcvax!ucbvax!miriam

Mike Banahan
The Instruction Set Ltd.
152 - 156 Kentish Town Road
London NW1 9QB, Great Britain
+44 1 482 2525
mcvax!ukc!inset!mikeb

Charles Bigelow
Bigelow & Holmes
15 Vandewater Street
San Francisco, Ca., 94133, U.S.A.
+1 414 788 8973
mcvax!cab@su-ai.arpa

Mike Burrows
Churchill College
Cambridge CB3 0DS, Great Britain
+44 223 277041
mcvax!mb@cl-steve.cam.ac.uk

P.J. Cameron
Plessey Microsystems Ltd
Water Lane
Towcester, Northants.,NN12 7JN, GB
+44 327 50312
mcvax!qtlon!pmllab!pjc

Ms. Julia Dain
University of Warwick
Dept. of Computer Science
Coventry CV4 7AL, Great Britain
+44 203 24011 x 2363
mcvax!ukc!warwick!julia

Martijn de Lange
ACE Associated Computer Experts
N.Z. Voorburgwal 314
1012 RV Amsterdam, The Netherlands
+31 20 245444
mcvax!ace!martijn

Theo de Ridder
IHBO "de Maere"
P.O. Box 1075
7500 BB Enschede, The Netherlands
+31 53 324247
mcvax!im60!ridder

Bob Duncanson
Unix Europe Ltd.
27a Carlton Drive
London SW15 2BS, Great Britain
+44 1 785 6972
mcvax!ukc!uel!rld

Kevin J. Dunlap
Digital Equipment Corp
Ultrix Engineering Group, Berkeley Division
P.O. Box 40220
Berkeley, Ca., 94704, U.S.A.
+1 415 643 6449
mcvax!ucbvax!kjd

Jacques Febvre
Bull Sems
Boite Postale 208
38432 Echirolles Cedex, France
+33 76 39 75 00
mcvax!vmucnam!echbull!xopen

Jim Hughes
Summit Operations / UNIX Systems Development
Amdahl Corporation
Room E-123
190 River Road
Summit N.J. 07901, U.S.A.
+1 201 522 6039
mcvax!seismo!ihnp4!attunix!jfh

Brian W. Kernighan
Bell Laboratories
600 Mountain Avenue
Murray Hill, N.J. 07974, U.S.A.
+1 201 582 3445
mcvax!research!mrkos!bwk

Douglas P. Kingston III
Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam, The Netherlands
+31 20 5929333
mcvax!dpk

Claude Kintzig
CNET PAA/RDS/RVA
38 Rue du General Leclerc
92131 Issy-les-Moulineaux, France
+33 1 6385047
mcvax!ogesml0!reseau
Dest:rva/kintzig (this must be provided as
an extra address line in all electronic messages)

Sylvain Langlois
Projet Rose
Bull, PC 33-05
68 Route de Versailles
78430 Louveciennes, France
+33 902 5060
mcvax!vmucnam!lvbull!sylvain

Georg P. Philippot
NCR Education Nordic Area
P.O. Box 2685 St. Hanshaugen
N0131 Oslo 1, Norway
+47 2 201200

Conor Sexton
Motorola International Software Development Centre
Mahon Industrial Estate
Blackrock, Cork, Ireland
+353 21 357101

Bjarne Stroustrup
Bell Laboratories
600 Mountain Avenue
Murray Hill, N.J., 07974, U.S.A.
+1 201 582 3445
mcvax!research!wild!bs

David Tilbrook
Imperial Software Technology Ltd.
60 Albert Court
Prince Consort Road
London SW7 2BH, Great Britain
+44 1 581 8155
mcvax!ukc!ist!dt

EUUG European UNIX systems User Group
Owles Hall, Buntingford
Herts. SG9 9PL
United Kingdom
+44 763 73039
mcvax!inset!euug

The Secretary
European UNIX† systems User Group
Owles Hall
Buntingford
Herts. SG9 9PL.
Tel: Royston (0763) 73039