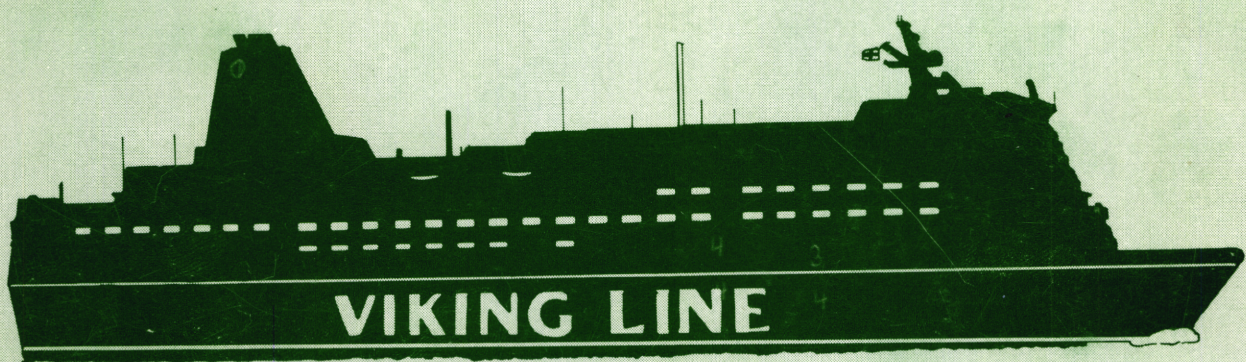
EUUG

European UNIX® systems User Group

CONFERENCE PROCEEDINGS



"UNIX® GROWS UP"

SPRING 1987 Conference
Finland & Sweden

®UNIX is a Registered Trade Mark of AT&T in the USA and other Countries

SPRING'87

CONFERENCE PROCEEDINGS

On board M/S Mariella sailing between
Helsinki and Stockholm

May 12-14 1987

This volume is published as a collective work. Copyright of the material in this document remains with the individual authors or the author's employer.

Further copies of the proceedings may be obtained from:

EUUG Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

UNIX is a registered trademark of AT&T in the USA and other countries.

ACKNOWLEDGEMENTS

Many people have contributed to the production of this volume. It is not possible to thank them all individually, but the following deserve special note:

Programme Chair:

Ms Jean Wood
Digital Equipment Europe

Local Organisers:

Mr Bjorn Eriksen (Sweden)
Mr Pekka Nikander (Finland)

Programme Committee:

Mr Kim Biel-Nielsen
Mr Johan Helsingius
Mr Hans Albertsson
Mr Neil Todd

These proceedings have been produced with the kind help of IST London.

Thanks to all the authors who submitted their paper in time and in the proper format to allow complete typesetting with *troff* and the associated processors and macros.

Finally it is appropriate to thank the many authors who submitted very good papers, but were not selected for this conference.

Due to the nature of the venue it was impossible to fit every paper into the programme. This meant that the Programme Committee had to perform a task that, as the standard of submitted papers rises, becomes harder each year.

The committee would have accepted more papers, had the time to present them been available. We hope that the unsuccessful authors will consider submitting further papers to future EUUG conferences.

CONTENTS

UNIX Conferences in Europe 1977-1987 & EUUG Meetings 7

Technical Programme Timetable 8

Mike Forsyth .... Vorlich - A Regular Expression Processor 9

Rob Pike .... Structural Regular Expressions 21

David J. Brown and
Jonathan P. Bowen .... The Event Queue 29

Dominic Dunlop .... I Come to Bury Unix.... And to Praise It 53

Pascal Beyls & Bertram Halt ... Now UNIX Talks to Me in My Language 63

Dr. G. Kruse .... UNIX in Manufacturing 81

Peter S. Langston .... UNIX and Entertainment Why and How? 93

Dale Shipley .... Distributed UNIX in Large-Scale Systems 103

Douglas V. Larson .... UNIX for Real Time 107

Andrew S. Tanenbaum .... MINIX: A UNIX Clone with Source Code 117

Brian E. Redman .... A User Programmable Telephone Switch 127

R. Grafendorfer .... UNIX in Banking 141

Osmo Hamalainen and
Markus Rosenstrom .... A Digital Selective Calling System for Use in the Maritime Mobile Service (DSC) 145

Alan Chantler .... Using the UNIX Operating System to Market Selected Information 153

Philip H. Dorn .... Is There a Future for UNIX in the World of Commercial Computing? 165

Ernst Janich .... Automating Administration of UNIX-Systems with Thousands of Users 175

Doug Michels .... The Development of a Standard UNIX System for Intel-based Microcomputers: A Technical Perspective 181

Bjarne Stroustrup .... Multiple Inheritance for C++ 189

Christian Tricot .... MuX: A Lightweight Multiprocessor Subsystem Under UNIX 209

Martin D. Beer .... Using a UNIX Engine as an Intelligent Information Server 227

Dr. Rolf Strothmann .... A UNIX SVID Compatible System Based on PXROS 233

RESERVE PAPER

Nick Nei .... Integrating the Apple Macintosh in a UNIX Environment 247

NAMES AND ADDRESSES OF SPEAKERS 261

UNIX Conferences in Europe 1977–1987

UKUUG/NLUUG meetings

1977 May	Glasgow University
1977 September	University of Salford
1978 January	Heriot Watt University, Edinburgh
1978 September	Essex University
1978 November	Dutch Meeting at Vrije University, Amsterdam
1979 March	University of Kent, Canterbury
	Brian Kernighan & Ken Thompson
1979 October	University of Newcastle
1980 March 24th	Vrije University, Amsterdam
	Steve Johnson
1980 March 31st	Heriot Watt University, Edinburgh
1980 September	University College, London

EUUG Meetings

1981 April	CWI, Amsterdam, The Netherlands
	Dennis Ritchie
1981 September	Nottingham University, England
1982 April	CNAM, Paris, France
1982 September	University of Leeds, England
1983 April	Wissenschaft Zentrum, Bonn, Germany
1983 September	Trinity College, Dublin, Eire
1984 April	University of Nijmegen, The Netherlands
1984 September	University of Cambridge, England
1985 April	Palais des Congres, Paris, France
1985 September	Bella Center, Copenhagen, Denmark
1986 April	Centro Affari/Centro Congressi, Florence, Italy
1986 September	UMIST, Manchester, England
1987 May	On board M/S Mariella sailing between Helsinki and Stockholm

TECHNICAL PROGRAMME TIMETABLE

	Tuesday	Wednesday	Thursday
09:00		* Cray Title to be announced	EUUG business
09:30		Shipley Distributed UNIX in large scale systems	Janich Automating administration of UNIX systems with thousands of users
10:00		Larson UNIX for real(time)	Michels Standard UNIX System for Intel-based Micros
10:30	Coaches leave Hotel Dipoli	BREAK	BREAK
11:00	Embarkation	Tanenbaum MINIX: A UNIX Clone for the IBM PC	Stroustrup Multiple inheritance for C++
11:30			Tricot muX: a lightweight multiprocessor subsystem under UNIX
12:00		Free	Beer Using a UNIX machine as an intelligent information server
12:30			Strothmann A UNIX SVID Compatible System Based on PXROS
13:00	Official Opening†	Time	LUNCH
13:30	* Bill Joy† Computer workstation architecture: 1982-1992		
14:00			Disembarkation starts
14:30	BREAK	in	Coach 1 to Hotel Dipoli
15:00	Forsyth Vorlich - A regular expression processor for UNIX systems		Coach 2 to Hotel Dipoli
15:30	Pike Structural regular expressions	Stockholm	
16:00	Brown & Bowen The Event queue - An extensible input system for UNIX workstations	Redman A user programmable telephone switching system	
16:30	BREAK Access to cabins	* Kutscha A study of portability problems arising in software development for optimised goods (rail-road) traffic	
17:00		Grafendorfer UNIX in Banking	
17:30	Dunlop I come to bury UNIX.. and to praise it	Osmo Hamalainen A Digital Selective Calling System for use in the Maritime Mobile Service (DSC)	
18:00	Beyls & Halt Now UNIX talks to me in my language	BREAK	
18:30	Kruse UNIX in manufacturing	Chantler Using the UNIX operating system to market selected information	
19:00	BREAK	Dorn Is there a future for UNIX in the world of commercial computing ?	
19:30	Langston UNIX in entertainment		
20:00		DINNER	
20:30	DINNER		
21:00			
21:30		PANEL "Has UNIX grown up ?	
22:00	BOF System V issues		
22:30		BOF C++	
23:30			

All events are in the main auditorium except those marked † which are in the nightclub.

* Paper not included

Vorlich - A regular expression processor

Paul Cockshott, Mike Forsyth and Patrick Foulk
MEMEX
Edinburgh, Scotland (UK)

ABSTRACT

Vorlich is custom designed hardware to perform serial searching using regular expressions, similar to the utility **GREP** under **UNIX**+, at disk transfer speeds. The searching is thus offloaded from the host processor of the system. The hardware has a defined instruction set which makes it programmable for a wide set of applications. This paper describes the use of regular expressions in searching and the architecture and instruction set of the **Vorlich** processor.

+**UNIX** is a Trademark of Bell Laboratories.

Introduction

Vorlich is a programmable regular expression processor with the functional ability to -

Search free text for words - the OR of 40 to 50 terms
Perform range, proximity and ordered searches
Wildcard searching
Fixed and free format records or structured searching.

The processor board sits on the bus of the host system, scans data sent to it for a given regular expression and returns the byte addresses of hits within the data.

Whereas **GREP**, and **UNIX** utilities using regular expressions, delimit their search area by a newline, **Vorlich** may be programmed to allow a regular expression to define the bounds of the search area. Thus a lexical area such as sentence or paragraph, rather than a single line, may be used to delimit where the query may be satisfied.

The problem of searching is that of recognising a language, the query, in an input stream, the database. Linguistic science provides us with a well established theory of language and from this we examine grammars.

Grammars.

A language can be defined formally as below. Note that a **Symbol** is not formally defined, it is an abstract entity similar to **Point** and **Line** in Geometry.

An **alphabet T** is a finite set of symbols.

A finite sequence of symbols t1,t2,t3... from an alphabet **T** is called a **string**.

The **length** of a string is the number of symbols in the string.

Concatenation is the operation of writing one string after another, thus
a1a2a3 concatenated with **b1b2b3**
forms
a1a2a3b1b2b3.

The **null string** has length 0 and is denoted by **Nil**. Any word concatenated to **Nil** is equal to the word.

T* is the set of all strings over the alphabet **T** including the empty symbol. This is known as the **Kleene Closure**.

A **language L**, over the single alphabet **T**, is a subset of **T***.

A language consisting of a finite number of strings can be defined by simply exhaustively listing those strings. However infinite languages cannot be defined as such and are defined by using a **grammar**.

A **grammar** consists of:

A finite set of **Non-Terminal** symbols N (variables)
A finite set of **Terminal symbols** T (note that $N \cap T = \emptyset$)
A **start symbol** S which is a Non-Terminal.
A **set of productions** of the form
u -> v
where u and v can be Non-Terminal and Terminal symbols and v may be empty.

There are four classifications of grammars, introduced by **Chomsky** [5], these are classified by restrictions on their productions -

Class 0 - Unrestricted

No restrictions on productions, this is the most powerful and is equivalent to the general type of computational mechanism known as the **Turing machine**. Languages generated from this grammar are those whose sentences can be generated by any deterministic computational machine.

Class 1 - Context Sensitive

Productions are of the form u -> v with the restriction that the length of u must be less than or equal to the length of v.

Note that u and v can be either Non-terminal symbols or Terminal symbols. Languages generated from this grammar are those whose sentences can be recognised by a deterministic computational machine using an amount of storage proportional to the length of the input - this is known as a **Linear Bounded Automaton.**

Class 2 - Context Free

Productions have the form A -> v where

A is restricted to be a Non-terminal symbol.
v may be either a Nonterminal or Terminal symbol.

This is more familiar in computer notation as Backus-Naur Form (BNF), which is used for defining some programming languages for a compiler and forms the basis of the Syntax Analysis stage of the compilation process. A **Push-Down Automaton** or **Stack** is required.

Class 3 - Regular

Productions may take one of two forms -

1. **Right Linear**
   A->tB or A->t

2. **Left Linear**
   A->Bt or A->t

where t is a terminal symbol and A and B are non-terminal symbols.

We are particulary interested in the Regular grammar because it is the simplest machine for the searching application. The regular language can be defined with a regular expression and can be easily represented using state transition diagrams or finite state machines. The state transition diagram is simply a directed graph.

Regular expressions are the basis of the Lexical Analysis stage of a compiler and they are the method employed for string searching by **UNIX** utilites such as **GREP**, **AWK** and the editors **VI** and **ED**. Finite state machines have been used in a branch of mathematics known as Automata theory, this has been used as the basis of much of the theory of computation. The state transition network provides the frame of a recogniser in that it describes a language, this can be restricted to that of a specific pattern providing a query.

A Regular language satisfies the properties of a context free language thus a regular language is also a context free langauge. They are a simple but non-trivial subclass of context free languages.

Regular grammars are conveniently represented by the use of transition diagrams. The construction of the transition diagram is described at length elsewhere [6],[7]. It is to be noted that there is a fault in the algorithm given in [7], it does not handle backtracking correctly and will not, for example, find the string "**mississippi**" when given "**issip**".

The transition diagram produced represents a Non-Deterministic Finite State machine (or automaton) - the nodes are known as states and the arcs as transitions. The number of nodes is finite hence the name Finite State.

The Non-Deterministic description refers to the operation of the machine in that at certain points when traversing the graph non-deterministic choices must be made - there are several possible traverses from a given state.

Instead of using the transition diagram to represent a regular language one may more conveniently use a **Regular Expression.**

A **regular expression** over the Alphabet T and the language denoted by that regular expression is defined as-

1. ø is a regular expression denoting the empty set.
2. Ω is a regular expression denoting { Ω }.
3. **a** where **a** is a Terminal symbol is a regular expression denoting {**a**}.
4. If P and Q are regular expressions denoting the languages Lp and Lq then -

   (P+Q) is a regular expression denoting Lp U Lq. (Alternation)
   (P.Q) is a regular expression denoting Lp.Lq (Concatenation)
   (P*) is a regular expression denoting
   { Ω } U Lp U Lp U Lp.... (Kleene Closure)

A regular expression denotes a language - but there may be several regular expressions that denote the same language. For example the regular expressions (a+b)* and (a*b)*a are equivalent as they both describe a set of strings containing a's and b's.

Examples of regular expressions may be seen by using the **UNIX** utility programs, for example the **EGREP** program searches for strings in text, thus

EGREP "example" database	(using **Concatenation** of e.x.a.m.p.l.e)
EGREP "jack|jill" database	(Using both **Concatenation** and **Alternation** of j.a.c.k and j.i.l.l)
EGREP "t*" database	(Using **Closure** to match 0 or more t's)

Deterministic Recognition with transition networks.

With a Non-Deterministic machine we are faced with several solutions to the problem of multiple transitions from a given state. The solution we took was to convert the network into a Deterministic machine so that there is no choice. This conversion is performed using software before the recogniser is presented with its input stream. The conversion algorithm is described elsewhere [6],[7]. Thus the Deterministic machine is performing the recognition and the Non-Deterministic machine is only a step in the conversion process.

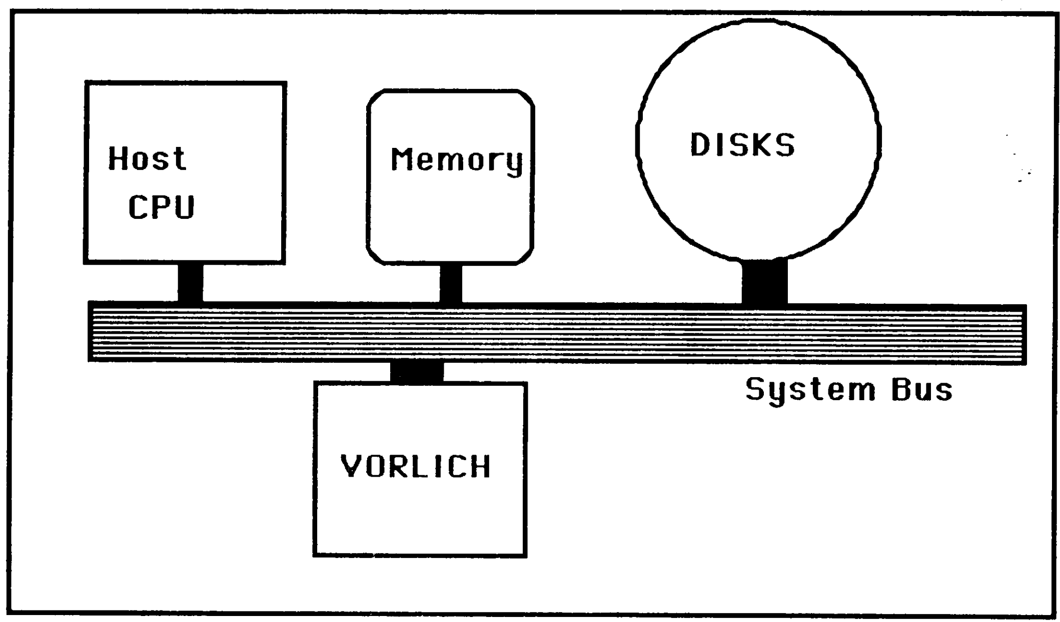
Another approach would be to use the Non-Deterministic machine for searching and to perform Parallel searching. Thus on reaching a state with multiple transitions the processor takes all of the transitions at once. On reaching the final state the other searches may be terminated. This solution is more appropriate to a parallel processing environment.

The second strategy is to follow one of the transitions and on failing to Backtrack taking a different alternative. Although the work done by the Parallel search is **linearly proportional** to the sentence being recognised, the processing is never larger than the number of states in the network, the backtracking algorithm is **exponentially related** to the length of the sentence.

The solution we chose was that of converting from a Non-Deterministic machine to the Deterministic machine. The time taken to perform this conversion is dependent on the length of the query but the search time is constant. It is the least expensive solution from the hardware costs, simplifying the hardware significantly. The hardware implements what is known as a **Moore machine** - a finite automata with output on states. Output is given on reaching final states otherwise a NULL output is performed.

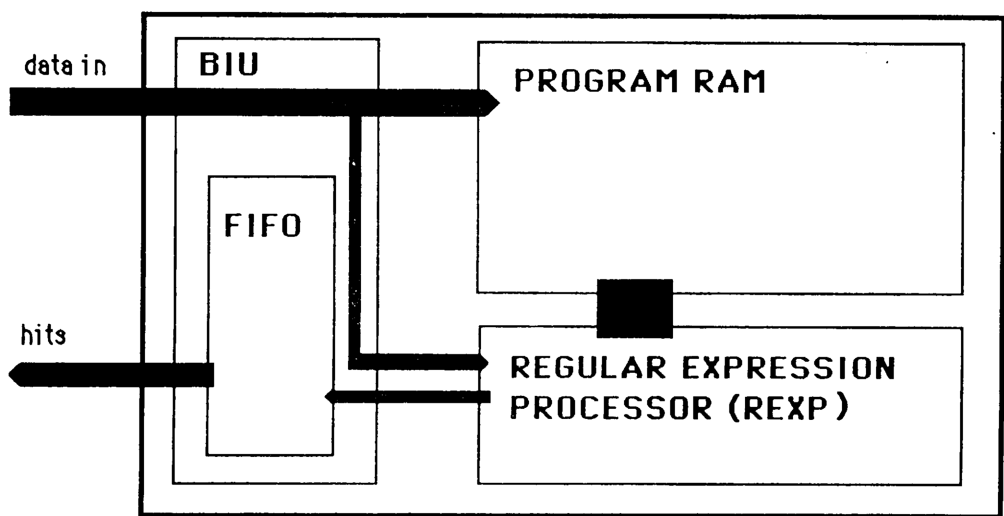
Architecture of Vorlich.

The Vorlich board sits on the host system bus reading data from disk or memory under control of an application program and driver. All searching is done outwith the host CPU as **Vorlich** acts as a searching co-processor on the system.



It may connect to the system bus via an interface board or directly onto the system bus allowing DMA from disk to **Vorlich** direct. It is not built into a disk controller to ease portability between different systems and to take advantage of faster disk controllers as they become available, without redesign.

The Vorlich board itself consists of three functional units - BIU, REXP and RAM.



1. Regular Expression Processor (REXP)
   A PAL programmed to execute a defined instruction set and alter the data path.

2. Bus Interface Unit (BIU)
   Handles communications between system bus
   or bus interface and the REXP. This also holds a FIFO
   for buffering the results from the REXP.

3. Program Memory
   64K RAM holding program for REXP to execute.

Instruction Set.

It is simpler to describe the architecture and operation of the REXP by providing a description of the instruction set. The following register set of the REXP is referred to

C	- Hold current character
PC	- Program Counter
CNT	- Character Counter
HL	- Hit Latch
FIFO	- Hit Memory

The output of the regular expression compiler is normally the machine code for the machine. For ease of testing and to provide diagnostics for the board an assembler was written. This may also provide an application interface at a lower level than the regular expression.

The instruction set as provided for the assembler is described below.

MUSTBE operand address

The operand field contains the character that the input stream must match. If it matches with C then the **PC** is incremented otherwise the **PC** is set to the address field, equivalent to a jump on failure. On success a character is read from the input stream into C.

MUSTNTBE operand address

The operand field contains the character that the input stream must not match. If it does not match with C then the **PC** is incremented otherwise the **PC** is set to the address field. On success a character is read from the input stream into C.

JUMP address

Set **PC** to address.

FIRST

Latch current character count **CNT** in **HL** - this is used to set the hit address. This extends the bounds of a search area as compared with **GREP** which uses the newline.

OUT operand address

The **FIFO** status byte, operand and latched character counter **HL** are put into the **FIFO**, in that sequence. A character is read from the input stream into C and then the **PC** is set to the address.

READI

Read a character from the input stream into C and increment **PC**.

HALT

Halt the processor.

CLASSJUMP operand address

This is a class jump instruction. The operand field holds the class table number (0-255) and the address is to the start of a jump table. The jump table is a sequence of contiguous locations in program RAM filled with **JUMP** instructions to other areas of the program. The class tables are located at the fixed address 8K -16K. They contain an offset in the jump table. The jump table is located at the address given in the address field and the PC is set to **classtables[operand][C] + address.**

This implements the character class construct, notationally equivalent to a multiple OR.

CTS opcode

Signifies the start of the class table given in the opcode. This corresponds to the opcode field in a classjump instruction naming the table. Any **cte** instructions which follow will be associated with this table.

CTE opcode classnumber

Class Table Entry sets character entry given by **opcode** with **classnumber**. The classnumber provides the address in a sequence of jump instructions as described in the **CLASSJUMP** instruction.

Memory Allocation.

There is 64K of 32 bit words of program memory on the board. There is however 8K of this pre-allocated for class tables at 8K - 16K.

Example program: **grep 'issip' US_Rivers**

0 readi	read in a character
1 mustbe i 0	if it is not an 'i' goto 0
2 classjump 0 16	many way branch on input character
3 readi	
4 classjump 0 32	
5 readi	
6 mustbe i 0	
7 classjump 1 48	
8 readi	
9 first	mark current character position
10 out 'H' 1	output to FIFO with token 'H' and goto 1
11 halt	
16 jump 1	Jump Table at location 16
17 jump 2	
18 jump 4	
32 jump 1	Jump Table at location 32
33 jump 2	
34 jump 6	
48 jump 1	Jump Table at location 48
49 jump 2	
50 jump 9	
51 jump 4	
cts 0	Start of class table definitions
cte i 1	These provide offsets within a jump table
cte s 2	for the given character - 0 is the default
cts 1	
cte i 1	
cte p 2	
cte s 3	

Note that the jump table for the classjump must start on a 16 instruction boundary and consequently the addresses for the jump table must be
16,32,48,64.......

The program memory is 1/4 Megabyte which provides 56K instruction programs. Note that the class jump instruction uses up 17 instructions by having to preallocate 16 entries in a jump table. These entries can be re-used by the compiler when it discovers they are free. The class tables are preallocated 8K which cannot be reclaimed.

Performance and Enhancements.

Currently Vorlich can search at disk transfer rates, from a Berkeley 4.2 system it searches at about 600K/s. The board itself is capable of searching in excess of 1 Megabyte per second.

To speed up disk transfers we developed a Contiguous Filesystem under UNIX which allows several cylinders to be the minimum allocation unit on the system. This resides along with the UNIX filesystem on the host machine. With this filesystem we can effect a transfer rate of 1 Megabyte per second from the disks.

Depending on the host machine data may have to be transferred into buffers in memory. In this case double buffering is found to be effective in hiding this delay. On other machines the board may be interfaced directly to the bus and we can DMA direct to the board.

The board has the capability of handling multiple users/queries by segmenting the program RAM and class tables. A scheduler in the host computer can allocate the users/queries program space and reload the Program Counter of Vorlich. The hits returned from the board can be tagged using the operand field of the OUT instruction to allow an application program to sort out which hits belong to which user.

Although programs for searching free text tend to consume little program space, other applications may require more than the 56K instruction space. The successor to Vorlich will have extendible memory boards and wider registers, for applications in Linguistics that require very large state machines.

Applications of regular expressions.

We have programmable hardware capable of searching at disk transfer rates for regular expressions, here we describe some simple applications which can use the speed of searching through free and structured text. Taking the UNIX syntax for regular expressions as an example we can use regular expressions in free text searching -

Concatention

c.o.m.p.u.t.e searches for the text "**compute**"
It will also find any words with compute embedded
within such as "micro**compute**rs".

Alternation

a|d searches for the characters '**a**' or '**d**'

Combinations

computer|crime will use both concatenation and
alternation to search for the strings
"**computer**" OR "**crime**".

Closure

3450* searches for the number 345 with or
without trailing zeroes.

Character Class

[A-Za-z] searches for alphabetic characters. The character class may be used to identify the delimiters of the terms searched for, so that by concatenating these delimiters to the text string the word "**compute**" would be found and not the word "**microcomputers**".

Numerics

>1985 can be represented by the regular expression -

198[5-9] | 199[0-9] | [2-9][0-9][0-9][0-9] |
0*[1-9][0-9][0-9][0-9][0-9][0-9]*

Similarly a range such as 1976-1994 can be represented as

197[6-9] | [198[0-9] | 199[0-4]

Word delimiters have been omitted for readability.

Structured Searches.

For record based data we can identify 3 general methods of structuring data - Fixed length field records, variable length fields using unique tags and variable length fields using non-unique tags. These can all be handled using regular expressions.

For example given a fixed field record structure of the following

```
struct name { char surname[20];char initials[5];char first[20]};
struct address { char street[20];char city[20];char zipcode[8]; }

struct personnel {
        struct name Name;
        struct address Address
};
```

and a query looking for all personnel with the surname of "MacBeth" who live in the city of "Edinburgh" we could construct the following regular expression -

"MacBeth.+13.+45Edinburgh.+11.+28"

where . represents a character class of every character (dont care) and +N represents the fixed number N of the preceding character. The .+13 after MacBeth for example simply skips the remainder of the surname field. On failure of the regular expression Vorlich assembler can be written to read in the remaining characters in the record to synchronise the search.

If the fields of a record are not fixed then tags may be inserted which let a program know the start or end of a particular field. For the record structure name this may be of the form

RS FS surname **FS** initials **FS** first **RS** ....

where **RS** is a record seperator and **FS** is a field seperator. The field seperators may or may not be unique. A regular expression for searching concatenates the seperator to bind the search to the particular field it is looking within.

Conclusion.

Vorlich was designed as a programmable tool for free text searching allowing flexibility in searching without the need of inverted files. It can be thought of as a hardware **GREP** searching at disk transfer rates and freeing the host CPU for other applications.

The hardware provides a performance enhancement to many applications which use searching and is not restricted to free text. Application program writers have access to Vorlich via both the regular expression interface and an assembly language. Although employing serial searching gives a linear search time for a file, that search time becomes prohibitive for GigaByte files. A coarse index may be built upon the system cutting down the search space considerably.

It has proved useful as a tool to speed up **GREP** on our **UNIX** systems and can provide the mechanism of similarly speeding up other applications.

References.

1. Aho, A.V and M.J. Coraisick "Efficient string matching: An aid to bibliographic search" - Comm ACM,18, 333-340 (1975)

2. Boyer, R.S and J. Strother Moore "A fast string searching algorithm"
Comm ACM,20, 762-772 (1975).

3. Knuth, D.E, J.H. Morris and V.R. Pratt "Fast pattern matching in strings"
SIAM Journal of Computing,6,323-350 (1977).

4. Thompson K, "Regular expression search algorithm"
Comm. ACM,11, 419-422 (1968)

Books:

5. Language as a cognitive process - T. Winograd : Addison-Wesley.
6. Syntax of Programming Languages - R.C. Backhouse :Prentice-Hall.
7. Principles of Compiler Design - Aho, Hopcroft , Ullman : Addison-Wesley.

Structural Regular Expressions

Rob Pike

Bell Laboratories
Murray Hill
New Jersey 07974
U.S.A.

ABSTRACT

The current UNIX text processing tools are weakened by the built-in concept of a line. There is a simple notation that can describe the 'shape' of files when the typical array-of-lines picture is inadequate. That notation is regular expressions. Using regular expressions to describe the structure in addition to the contents of files has interesting applications, and yields elegant methods for dealing with some problems the current tools handle clumsily. When operations using these expressions are composed, the result is reminiscent of shell pipelines.

The Peter-On-Silicon Problem

In the traditional model, UNIX text files are arrays of lines, and all the familiar tools — **grep**, **sort**, **awk**, etc. — expect arrays of lines as input. The output of **ls** (regardless of options) is a list of files, one per line, that may be selected by tools such as **grep**:

```
ls -l /usr/ken/bin | grep 'rws.*root'
```

(I assume that the reader is familiar with the UNIX tools.) The model is powerful, but it is also pervasive, sometimes overly so. Many UNIX programs would be more general, and more useful, if they could be applied to arbitrarily structured input. For example, **diff** could in principle report differences at the C function level instead of the line level. But if the interesting quantum of information isn't a line, most of the tools (including **diff**) don't help, or at best do poorly. Worse, perverting the solution so the line-oriented tools can implement it often obscures the original problem.

To see how a line oriented view of text can introduce complication, consider the problem of turning Peter into silicon. The input is an array of blank and non-blank characters, like this:

```
    #######
   #########
  ####  #####
 ####    ####  #
 ####     #####
####       ###
########   #####
#### #########
#### #  # ####
## #  ###   ##
###    #  ###
###    ##
 ##   #
  #   ####
  # #
##   #   ##
```

The output is to be statements in a language for laying out integrated circuits:

```
rect minx miny maxx maxy
```

The statements encode where the non-blank characters are in the input. To simplify the problem slightly, the coordinate system has x positive to the right and y positive down. The output need not be efficient in its use of rectangles. **Awk** is the obvious language for the task, which is a mixture of text processing and geometry, hence arithmetic. Since the input is an array of lines, as **awk** expects, the job should be fairly easy, and in fact it is. Here is an **awk** program for the job:

```
BEGIN{
    y=1
}
/^/{
    for(x=1; x<=length($0); x++)
        if(substr($0, x, 1)=="#")
            print "rect", x, y, x+1, y+1
    y++
}
```

Although it is certainly easy to write, there is something odd about this program: the line-driven nature of **awk** results in only one obvious advantage — the ease of tracking **y**. The task of breaking out the pieces of the line is left to explicit code, simple procedural code that does not use any advanced technology such as regular expressions for string manipulation. This peculiarity becomes more evident if the problem is rephrased to demand that each horizontal run of rectangles be folded into a single rectangle:

```
BEGIN{
    y=1
}
/^/{
    for(x=1; x<=length($0); x++)
        if(substr($0, x, 1)=="#"){
            x0=x;
            while(++x<=length($0) && substr($0, x, 1)=="#")
                ;
            print "rect", x0, y, x, y+1
        }
    y++
}
```

Here a considerable amount of code is being spent to do a job a regular expression could

do very simply. In fact, the only regular expression in the program is ^, which is almost irrelevant to the input. (Newer versions of **awk** have mechanisms to use regular expressions within actions, but even there the relationship between the patterns that match text and the actions that manipulate the text is still too weak.)

Awk's patterns — the text in slashes // that select the input on which to run the actions, the programs in the braces {} — pass to the actions the entire line containing the text matched by the pattern. But much of the power of this idea is being wasted, since the matched text can only be a line. Imagine that **awk** were changed so the patterns instead passed precisely the text they matched, with no implicit line boundaries. Our first program could then be written:

```
BEGIN{
	x=1
	y=1
}
/ /{
	x++
}
/#/{
	print "rect", x, x+1, y, y+1
	x++
}
/\n/{
	x=1
	y++
}
```

and the second version could use regular expressions to break out complete strings of blanks and #'s simply:

```
BEGIN{
	x=1
	y=1
}
/ +/{
	x+=length($0)
}
/#+/{
	print "rect", x, x+length($0), y, y+1
	x+=length($0)
}
/\n/{
	x=1
	y++
}
```

In these programs, regular expressions are being used to do more than just select the input, the way they are used in all the traditional UNIX tools. Instead, the expressions are doing a simple parsing (or at least a breaking into lexical tokens) of the input. Such expressions are called *structural regular expressions* or just *structural expressions*.

These programs are not notably shorter than the originals, but they are conceptually simpler, because the structure of the input is expressed in the structure of

the programs, rather than in procedural code. The labor has been cleanly divided between the patterns and the actions: the patterns select portions of the input while the actions operate on them. The actions contain no code to disassemble the input.

The lexical analysis generator **lex** uses regular expressions to define the structure of text, but its implementation is poor, and since it is not an interactive program (its output must be run through the C compiler) it has largely been forgotten as a day-to-day tool. But even ignoring issues of speed and convenience, **lex** still misses out on one of the most important aspects of structural expressions. As the next section illustrates, structural expressions can be nested to describe the structure of a file recursively, with surprising results.

Interactive Text Editing

It is ironic that UNIX files are uninterpreted byte streams, yet the style of programming that most typifies UNIX has a fairly rigid structure imposed on files — arrays of not-too-long lines. (The silent limits placed on line lengths by most tools can be frustrating.) Although the **awk** variant introduced above does not exist, there is an interactive text editor, **sam**, that treats its files as simple byte streams.

The **sam** command language looks much like that of **ed**, but the details are different because **sam** is not line-oriented. For example, the simple address

```
/string/
```

matches the next occurrence of "string", not the next line containing "string". Although there are shorthands to simplify common actions, the idea of a line must be stated explicitly in **sam**.

Sam has the same simple text addition and modification commands **ed** has: **a** adds text after the current location, **i** adds text before it, **d** deletes it, and **c** replaces it.

Unlike in **ed**, the current location in **sam** need not be (and usually isn't) a line. This simplifies some operations considerably. For example, **ed** has several ways to delete all occurrences of a string in a file. One method is

```
g/string/ s///g
```

It is symptomatic that a substitute command is used to delete text within a line, while a delete command is used to delete whole lines. Also, if the string to be deleted contains a newline, this technique doesn't work. (A file is just an array of characters, but some characters are more equal than others.) **Sam** is more forthright:

```
x/string/d
```

The **x** ('extract') command searches for each occurrence of the pattern, and runs the subsequent command with the current text set to the match (not to the line containing the match). Note that this is subtly different from **ed**'s **g** command: **x** extracts the complete text for the command, **g** merely selects lines. There is also a complement to **x**, called **y**, that extracts the pieces *between* the matches of the pattern.

The **x** command is a loop, and **sam** has a corresponding conditional command, called **g** (unrelated to **ed**'s **g**):

```
g/pattern/command
```

runs the command if the current text matches the pattern. Note that it does not loop, and it does not change the current text; it merely selects whether a command will run. Hence the command to print all lines containing a string is

```
x/.*\n/ g/string/p
```

— extract all the lines, and print each one that contains the string. The reverse conditional is `v`, so to print all lines containing 'rob' but not 'robot':

```
x/.*\n/ g/rob/ v/robot/p
```

A more dramatic example is to capitalize all occurrences of words 'i':

```
x/[A-Za-z]+/ g/i/ v/../ c/I/
```

— extract all the words, find those that contain 'i', reject those with two or more characters, and change the string to 'I' (borrowing a little syntax from the substitute command). Some people have overcome the difficulty of selecting words or identifiers using regular expressions by adding notation to the expressions, which has the disadvantage that the precise definition of 'identifier' is immutable in the implementation. With `sam`, the definition is part of the program and easy to change, although more long-winded.

The program to capitalize 'i's should be writable as

```
x/[A-Za-z]+/ g/^i$/ c/I/
```

That is, the definition of `^` and `$` should reflect the structure of the input. For compatibility and because of some problems in the implementation, however, `^` and `$` in `sam` always match line boundaries.

In `ed`, it would not be very useful to nest global commands because the 'output' of each global is still a line. However, `sam`'s extract commands can be nested effectively. (This benefit comes from separating the notions of looping and matching.) Consider the problem of changing all occurrences of the variable `n` in a C program to some other name, say `num`. The method above will work —

```
x/[a-zA-Z0-9]+/ g/n/ v/../ c/num/
```

— except that there are places in C where the 'identifier' `n` occurs but not as a variable, in particular as the constant `\n` in characters or strings. To prevent incorrect changes, the command can be prefixed by a couple of `y` commands to weed out characters and strings:

```
y/".*"/ y/'.*'/ x/[a-zA-Z0-9]+/ g/n/ v/../ c/num/
```

This example illustrates the power of composing extractions and conditionals, but it is not artificial: it was encountered when editing a real program (in fact, `sam`). There is an obvious analogy with shell pipelines, but these command *chains* are subtly — and importantly — different from pipelines. Data flows into the left end of a pipeline and emerges transformed from the right end. In chains, the data flow is implicit: all the commands are operating on the same data (except that the last element of the chain may modify the text); the complete operation is done in place; and no data actually flows through the chain. What is being passed from link to link in the chain is a view of the data, until it looks right for the final command in the chain. The data stays the same, only the structure is modified.

More than one line, and less than one line

The standard UNIX tools have difficulty handling several lines at a time, if they can do so at all. `Grep`, `sort` and `diff` work on lines only, although it would be useful if they could operate on larger pieces, such as a `refer` database. `awk` can be tricked into

accepting multiple-line records, but then the actions must break out the sub-pieces (typically ordinary lines) by explicit code. **sed** has a unique and clumsy mechanism for manipulating multiple lines, which few have mastered.

Structural expressions make it easy to specify multiple-line actions. Consider a `refer` database, which has multi-line records separated by blank lines. Each line of a record begins with a percent sign and a character indicating the type of information on the line: A for author, T for title, etc. Staying with **sam** notation, the command to search a `refer` database for all papers written by Bimmler is:

```
x/(.+\n)+/ g/%A.*Bimmler/p
```

— break the file into non-empty sequences of non-empty lines and print any set of lines containing 'Bimmler' on a line after '%A'. (To be compatible with the other tools, a '.' does not match a newline.) Except for the structural expression, this is a regular **grep** operation, implying that **grep** could benefit from an additional regular expression to define the structure of its input. In the short term, however, a 'stream **sam**,' analogous to **sed**, would be convenient, and is currently being implemented.

The ability to compose expressions makes it easy to tune the search program. For example, we can select just the *titles* of the papers written by Bimmler by applying another extraction:

```
x/(.+\n)+/ g/%A.*Bimmler/ x/.*\n/ g/%T/p
```

This program breaks the records with author Bimmler back into individual lines, then prints the lines containing **%**T.

There are many other examples of multiple-line components of files that may profitably be extracted, such as C functions, messages in mail boxes, paragraphs in `troff` input and records in on-line telephone books. Note that, unlike in systems that define file structures *a priori*, the structures are applied by the program, not the data. This means the structure can change from application to application; sometimes a C program is an array of functions, but sometimes it is an array of lines, and sometimes it is just a byte stream.

If the standard commands admitted a structural expression to determine the appearance of their input, many currently annoying problems could become simple: imagine a version of `diff` that could print changed sentences or functions instead of changed lines, or a `sort` that could sort a `refer` database. The case of `sort` is particularly interesting: not only can the shape of the input records be described by a structural expression, but also the shape of the sort key. The current bewildering maze of options to control the sort could in principle be largely replaced by a structural expression to extract the key from the record, with multiple expressions to define multiple keys.

The awk of the future?

It is entertaining to imagine a version of **awk** that applies these ideas throughout. First, as discussed earlier, the text passed to the actions would be defined, rather than merely selected, by the patterns. For example,

```
/#+/ { print }
```

would print only # characters; conventional **awk** would instead print every line containing # characters.

Next, the expressions would define how the input is parsed. Instead of using the restrictive idea of a field separator, the iterations implied by closures in the expression

can demarcate fields. For instance, in the program

```
/(.+\n)+/ { action }
```

the action sees groups of lines, but the outermost closure (the + operator) examines, and hence can extract, the individual lines. **ed** uses parentheses to define sub-expressions for its back-referencing operators. We can modify this idea to define the 'fields' in **awk**, so **$1** defines the first element of the closure (the first line), **$2** the second, and so on. More interestingly, the closures could generate indices for arrays, so the fields would be called, say, `input[1]` and so on, perhaps with the unadorned identifier `input` holding the original intact string. This has the advantage that nested closures can generate multi-dimensional arrays, which is notationally clean. (There is some subtlety involving the relationship between `input` indices and the order of the closures in the pattern, but the details are not important here.)

Finally, as in **sam**, structural expressions would be applicable to the output of structural expressions; that is, we would be able to nest structural expressions inside the actions. The following program computes how many pages of articles Bimmler has written:

```
/(.+\n)+/{    # break into records
     input ~ /%A.*Bimmler/{     # is Bimmler author? (see text)
          /%P.*([0-9]+)-([0-9]+)/{     # extract page numbers
               pages+=input[2]-input[1]+1
          }
     }
}
END{
     print pages
}
```

Real **awk** uses patterns (that is, regular expressions) only like **sam**'s **g** command, but our **awk**'s patterns are **x** expressions. Obviously, we need both to exploit structural expressions well. This is why in the program above the test for whether `input` contains a paper by Bimmler must be written as an explicit pattern match. The innermost pattern searches for lines containing two numbers separated by a dash, which is how `refer` stores the starting and ending pages of the article.

This is a contrived example, of course, but it illustrates the basic ideas. The real **awk** suffers from a mismatch between the patterns and the actions. It would be improved by making the parsing actions of the patterns visible in the actions, and by having the pattern-matching abilities available in the actions. A language with regular expressions should not base its text manipulation on a `substr` function.

Comments

The use of regular expressions to describe the structure of files is a powerful and convenient, if unfamiliar, way to address a number of difficulties the current UNIX tools share. There is obviously around this new notation a number of interesting problems, and I am not pretending to have addressed them all. Rather, I have skipped enthusiastically from example to example to indicate the breadth of the possibilities, not the depth of the difficulties. My hope is to encourage others to think about these ideas, and perhaps to apply them to old tools as well as new ones.

Acknowledgements

John Linderman, Chris Van Wyk, Tom Duff and Norman Wilson will recognize some of their ideas in these notes. I hope I have not misrepresented them.

The Event Queue
An Extensible Input System for UNIX Workstations

David J. Brown

Cambridge University Computer Laboratory
Corn Exchange Street, Cambridge CB2 3QG, England
djb@uk.ac.cam.cl
&
Digital Equipment Corporation
Workstation Systems Engineering
100 Hamilton Ave, Palo Alto California 94301, USA

Jonathan P. Bowen

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road, Oxford OX1 3QD, England
bowen@uk.ac.oxford.prg

ABSTRACT

A design is given for an event-based input system for use on graphics workstations that overcomes some problems associated with earlier designs. An interface is specified that allows a client to select events to be queued from the set of those available, read events from the queue, put events into the queue, flush the queue, and connect a set of events to a queued event. The system has been designed with the intention of being implementable on a number of systems and has also been formally specified using the Z specification language developed at Oxford. The current implementation effort focuses on UNIX† workstations

Motivation

Many existing input systems have been based on the design of a particular device or subsystem and/or have been specialized to the needs of a particular window system. This approach has several problems that are discussed toward the end of the paper. The event queue system is independent of any particular application or window system, and is not committed to any particular physical device or subsystem. It provides a set of general purpose interfaces and is extensible to accommodate new input devices and event types. This design results in a system that is useful in a diverse range of applications.

Formal Specification

The Z ("zed") specification language [Morg84, Sufr86, Spiv86a, Spiv86b, Wood86, King87, Haye87], based on set theory and developed at the Programming Research Group in Oxford, has been used to formally describe the operation of the system [Bowe87]. A condensed version of the formal specification is included as an appendix of this paper. This notation is sufficiently readable to be used in

† UNIX is a registered trademark of AT&T in the U.S.A. and other countries

documentation for those familiar with it. The approach is to present the abstract state of the system and to describe the initial state. The effect of individual operations (library routine calls in this case) on the state is then described. In a Z specification, the formal text is mingled with informal prose. These should complement each other, reinforcing the reader's concept of how the system works. For readers unfamiliar with Z, the informal text should be sufficient to describe the system. If desired, the formal text can be removed from a final copy of the formal documentation to give an informal presentation. Should any ambiguity exist however, the formal description is the final arbiter in the operation of the system.

In general, even if the formal specification of a system is not included in the final description, the formal specification process can only be beneficial to designing and documenting a system. The formal treatment helps to ensure that areas which may have otherwise been ignored are covered, and to detect incompleteness or inconsistency in an informal design. The event queue system took only a day to formally specify from the original informal description [Brow86]. The exercise highlighted a number of points which have subsequently been improved and clarified in the system described here.

Ideally, formal specification should be used as a design tool rather than as a post-hoc description language. Formal specification encourages the design of simple well-understood systems which can be formally reasoned about before the implementation stage. Having a more rigorous and better understood design before beginning implementation helps to reduce the number of iterations around the expensive development loop: design-implementation-testing.

The Event Queue

An event queue is a FIFO buffer of events between the system and the client. Activity on an input device or an action by a client can cause an event to be entered in the event queue. The Event Queue System under UNIX implements a number of such event queues, as there may be several independent applications that each require an event queue, or a single application which requires more than one. The succeeding few sections describe the client's interface to an event queue. The system implementation is described in a following section.

Event Types and Value Semantics

An event is very simple. It contains a device identifier and a value:

```
typedef unsigned short qDevice
typedef short qValue
```

The device identifier: **qDevice** refers to the object that generated the event or equivalently to the type of event.[1] The **qDevice** field provides a namespace for event types. An action by a client or an activity on an input device in the system can cause an event. When this occurs, the source of the event uniquely identifies the type of event with **qDevice**. A client which generates events may use the device identifier namespace in an application specific way. However, we reserve certain device identifiers in order to define a number of system-provided devices that we expect will be useful in many applications. The event value is the value associated with the device (object) at the time the event occurred. The meaning of the value depends on the event type. The value has fixed size.[2]

---

[1] It is important not to confuse the use of the term 'device' in the context of the Event Queue with what is meant by a UNIX device. The choice of the term 'device' is unfortunate in this respect.

[2] In this design the goal was to keep events very small, simple and of fixed length. The **qValue** type could be made larger if it were decided that there was a need for additional range. However, it is possible to obtain additional precision within the scope of this design by using multiple events and **connectevent()** (see below). Variable length events will be considered in a subsequent design.

At the present time a **qDevice** is in one of three classes: button, valuator or keyboard. Button devices have boolean values. The **qValue** associated with a button event will be either 0 indicating that the button went up, or 1 indicating that the button went down. Valuator devices have a range value associated with them. For example **MOUSEX, TABLETY, KNOB15,** or **CLOCK** are valuator devices. Valuator devices may be relative or absolute. For example we might have a **MOUSEX** device which provides absolute mouse x positions, or a **MOUSEDX** device which provides relative mouse x positions (deltas), or we might have both. Keyboard devices return character code values. For example, the device **ASCIIKEYBOARD** is a device which returns ASCII values in the low order 7 bits of **qValue**.

All events contain a device identifier. It is possible to determine the class of the device that caused the event from this identifier with the following requests:

```
boolean isbutton(device)
  qDevice device;

boolean isvaluator(device)
  qDevice device;

boolean isrelative(device)
  qDevice device;

boolean isabsolute(device)
  qDevice device;

boolean iskeyboard(device)
  qDevice device;
```

The class of the device that caused the event is important to the client since the event value semantics depend on this. The client will know how to interpret the event value based on the class of the device or perhaps the specific event type. Detailed knowledge of event value semantics is deferred to the client. The Event Queue System itself has minimal knowledge about value semantics. The extensibility of the system derives in part from this fact.

Selecting and Deselecting Events

The client can select a device to have its events queued (entered in the event queue) with a **qdevice()** request.

```
int qdevice(device)
  qDevice device;
```

When the value associated with a selected device changes, an event is entered in the event queue. We also refer to a selected device as an 'active' device. One **qdevice()** request must be issued for each device we wish to make active, but any subset of the available devices may be active at once. A device must be selected by **qdevice()** for events associated with it to be placed in the event queue. Devices that have not been selected (inactive devices) will not cause events to be entered in the event queue.

Devices that are presently selected to enter events in the queue (active) may be deselected (made inactive) with the **unqdevice()** request.

```
int unqdevice(device)
  qDevice device;
```

Event Filtering

Certain valuator devices such as a knob, mouse x or y position may be 'noisy' (i.e. frequently have a small change of value) and others such as a time of day clock may be very high resolution. Devices of these sorts may thus generate more events than the client actually wishes to see. The **threshold()** interface allows the client to specify that a minimum change must occur in a device's value before an event is entered in the queue.

```
int threshold(device, delta)
  qDevice device;
  int delta;
```

If the device *device* has been selected by **qdevice()** then a change of at least *delta* must occur on its associated value before an event will be placed in the queue[3].

Reading Events

The client obtains the next input event from the event queue with the **getevent()** request.

```
qDevice getevent(pValue)
  qValue *pValue;
```

Recall that an input event consists of two values: one identifying the device that generated the event, and one giving the value associated with the device at the time of the event. **getevent()** returns the device identifier, and places the value associated with the device for this event in the location pointed to by *pValue*. The client's usage is:

```
qValue value;
qDevice device;
...
device = getevent(&value);
```

For efficiency we are often interested in reading a number of events in one procedure invocation. The **getevents()** request performs this function.

```
qDevice getevents(EventArray, count)
  int count;
  struct {
    qDevice Device;
    qValue Value;
  } EventArray[/* count */];
```

getevents() returns the device associated with the first event and returns an array of *count* events in *EventArray*. **getevents()** does not return until all *count* requested events have been returned. The usual application of this interface is to acquire a set of events at once when we expect them to be entered contiguously in the event queue. (See the Binding Events section below).

[3] This interface makes sense only for valuator type devices. It returns an error code if *device* is an inapplicable type. For a relative (vs. absolute) valuator device the system accumulates relative changes until the absolute value of the total accumulated change exceeds *delta*. (For implementation purposes this implies that state must be kept in the system pertaining to the present value of active valuator devices).

Posting Events

Events may be 'posted' (i.e. placed) at the end of the event queue with the **postevents** request.

```
int postevents(EventArray, count)
   int count;
   struct {
      qDevice Device;
      qValue Value;
   } EventArray[/* count */];
```

count events are entered in the event queue with device id's and values taken from *EventArray* (passed in by the client). **postevents()** allows a client to simulate physical or virtual events generated by some other device, or to implement a pseudo-device to synthesize some new class of events (e.g. for a window manager or other software process). It is important that the client be able to enter several events into the queue as an atomic action. **postevents()** assures that the list of events passed in will be placed contiguously in the event queue, uninterrupted by other events that may occur on physical devices during the call.

Testing the Queue

Since the **getevent()** request is synchronous (i.e. it will suspend the caller until there is an event in the input queue to be read), the client may not wish to use it. The **qtest()** request allows the client to determine whether there is an event in the queue before issuing a **getevent()**.

```
qDevice qtest()
```

qtest() returns the device id associated with the first input event in the queue, or 0 if the queue is empty. The client may use **qtest()** to provide a non-blocking use of the input queue. This is implemented by using **qtest()** to see that there is an event and only then performing a **getevent()** or **getevents()** if input is available.

Binding Events

At the time a particular event occurs it is often necessary to observe the state (present value) of several other devices. Use of the **connectevent()** request allows us to annotate an event on an active device with an event containing the value of another device.

```
int connectevent(active_device, bound_device)
   qDevice active_device, bound_device;
```

Whenever *active_device* generates an event, an event associated with *bound_device* will also be placed in the event queue. The event queue system produces an event from the bound device by examining its value at the time the active device generates an event. We refer to the event produced from the bound device as an 'annotation' event. The annotation event is an event just like one we would get if *bound_device* was itself active and generated an event. The only distinction drawn by an annotation event is the manner in which that event was caused to be entered in the queue: The device was bound to an active device which changed value. This caused the event queue system to examine the value of the bound device and synthesize this event.

A device must be active before devices can be bound to it with **connectevent()**. If an attempt is made to bind a device to an inactive device (i.e. a device not presently selected to enqueue events) with **connectevent()**, no binding is made and an error code is returned. An active device may have any other device bound to it, and the bound device may be either active or inactive. That is, bound devices themselves do not have to be selected to enqueue events.

Multiple Bindings

The **connectevent()** request may be issued more than once to bind more than one device to a specified device[4]. When an event occurs on *active_device*, one event for that device will be placed in the event queue, and one event for each device bound to it by **connectevent()**. Annotation events are entered contiguously in the event queue immediately after the event for *active_device*, without any intervening events. Annotation events appear in the event queue in the order established by the invocations to **connectevent()**. Each annotation event contains the value of a bound device at the time the event from the active device occurred. **connectevent()** returns 0 as an error code if it was unable to honor the request, non-zero if successful.

The Binding Relationship

The association created between *active_device* and *bound_device* by **connectevent()** is directed. That is, if we bind **TABLETX** to **BUTTON156** with:

connectevent(**BUTTON156**, **TABLETX**);

the system will produce a **TABLETX** event when **BUTTON156** changes value, but not the converse. This relationship is illustrated in figure 1.

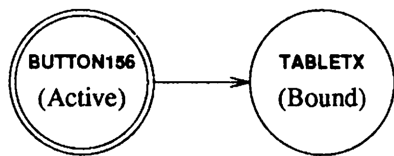


Figure 1 - Binding Relationships are Directed

We may of course establish the other relationship with:

connectevent(**TABLETX**, **BUTTON156**);

The binding relationship is a directed graph. Each *active_device* is the root node of a linked list of zero or more bound devices.

Binding Example

An example of the use of **connectevent()** is as follows: we wish to note the cursor's x and y position whenever the mouse's left button is depressed. We do not wish to know the cursor's position at any other time but when the left button is depressed, and therefore do not wish to select **CURSORX** or **CURSORY** to enqueue events. As a result we will select to enqueue mouse left button events and then bind the cursor x value and cursor y value to them with **connectevent()**:

[4] There is an implementation limit on the number of devices which can be bound to another device with **connectevent()**. This is based on the fact that there is a limited amount of storage available for data structures used to maintain the binding information and by the performance considerations of examining a large number of bound devices. An error code is returned if the request fails due to such a limit. These static implementation limits of the event queue system may be changed as application requirements and system performance direct. This is a compile-time parameter in the present implementation.

```
qdevice(MOUSELEFT);
connectevent(MOUSELEFT, CURSORX);
connectevent(MOUSELEFT, CURSORY);
```

The relationship established is shown in figure 2. Notice that since the first invocation of **connectevent()** referred to CURSORX and the second to CURSORY the CURSORX precedes CURSORY in the list of devices bound to MOUSELEFT.

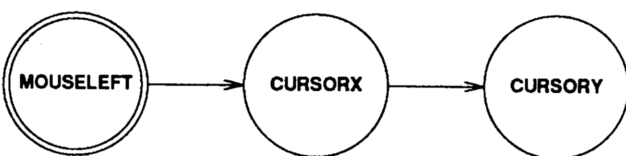


Figure 2 - Cursor Bound to Left Mouse Button

Subsequent to these requests, events containing the cursor x value and cursor y value will also appear in the event queue whenever there is a left mouse button event. The cursor x and cursor y events will appear immediately after the left mouse button event in the event queue and will contain the values of CURSORX and CURSORY at the time the mouse button event occurred. This arrangement is quite useful when the mouse button is being used to make a selection and the cursor x and y positions locate the selection. The mouse button is the device we want to have notify us (cause an event) so we can execute the procedure associated with selection. The cursor location is the annotation we need on the event so we can decide what was selected when the mouse button was pressed.

Another useful application of **connectevent()**, is that of timestamping certain events (i.e. the time of day clock is bound to a device). This can be used to note the absolute time that an event occured, or the duration between successive events. This can be used to implement the mouse button "double click" interface popular in some systems.

Unbinding Devices

It may be necessary to undo the binding between two devices. The **disconnectevent()** request performs this function.

```
int disconnectevent(active_device, bound_device)
    qDevice active_device, bound_device;
```

The request returns non-zero on success, and 0 for failure. Failure will be due to the fact that the *bound_device* referred to was not bound to the *device*.

Bindings are only maintained while a device is active. If an active device is made inactive with the **unqdevice()** request, all its bound devices are unbound. Bindings must be reestablished if an active device is made inactive and then reactivated.

Cursor

A cursor is a two dimensional locator, typically used to mark a point of interest on a display device. Although we may more readily think of the cursor in the output context, the cursor is also a subject closely related to the event queue mechanism. The cursor's two dimensional location is formed from a pair of valuator devices. These devices (CURSORX and CURSORY) are abstract devices which derive their current values from two concrete devices of valuator type (e.g. tablet x and y, knobs, mouse x and y, or the like).

The **attachcsr()** request allows the client to determine which pair of valuators will be used to determine the cursor position.

```
int attachcsr(xdevice, ydevice)
  qDevice xdevice, ydevice;
```

CURSORX and CURSORY are devices in their own right and may be queued with **qdevice** like any other device. The distinction between these devices and certain other ones is that there is no physical device which generates events of type CURSORX or CURSORY. They are effectively an abstraction of a chosen pair of valuators, and have additional special output semantics associated with another (the display) subsystem[5].

Overview of Implementation

The event queue system is implemented as a pseudo device. It is a system module analogous to the UNIX tty subsystem or the Berkeley line disciplines. It presents a standard UNIX device interface at the user level, but is not a driver for any particular physical device. Instead, UNIX device drivers which work with the event queue system can be caused to dispatch their low level physical input to the event queue module with the **queInput()** interface. The higher level semantics of the event queue are implemented centrally - in the Event Queue system module, rather than in the physical device drivers. This guarantees a uniform and consistent behavior to the client across the range of devices which work with the Event Queue System. The system structure is illustrated in figure 3.

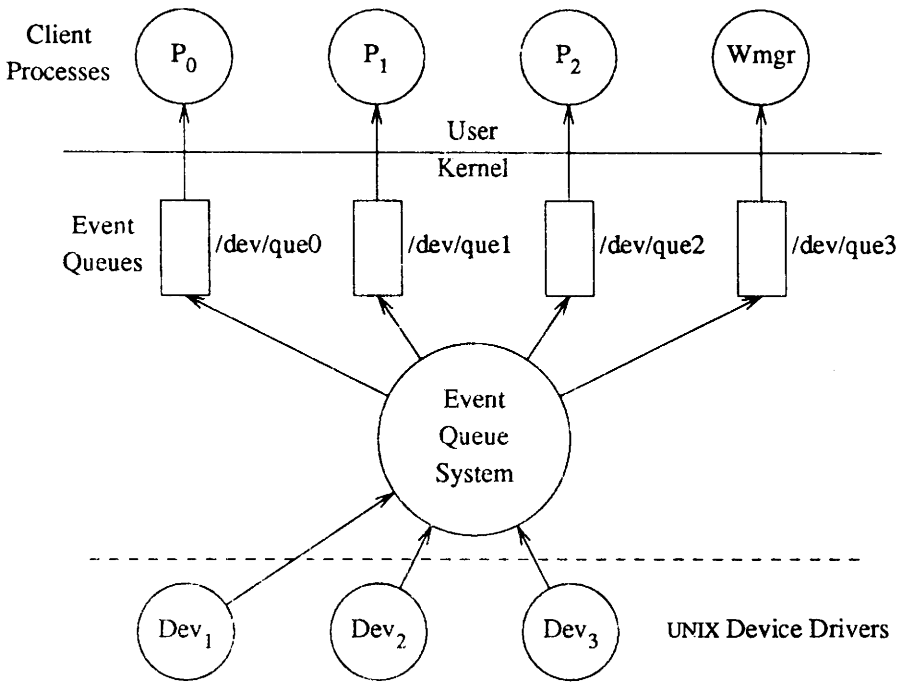


Figure 3 - Event Queue System Implementation Structure

The relationship of a physical device which works with the event queue to the event queue subsystem is highly similar to the relationship of a serial line multiplexer to the UNIX tty subsystem. In both cases, higher level uniform semantics are imposed across a variety of different physical devices in a common

[5] The cursor involves the interaction of two subsystems: the event queue and the display. This interaction is an interesting topic but is outside the scope of the present paper.

system module, not the device driver. In the same way that we do not wish to reimplement the terminal semantics of the tty system in the driver for each new serial line multiplexer, or the semantics of a network protocol in the driver for each new ethernet controller, we wish to avoid reimplementing the semantics of the event queue in each relevant physical device driver.

The event queue differs from the UNIX tty system and the line disciplines in that there is a many to one relationship between physical devices and an event queue. Multiple physical devices can cause events to be entered in a single queue. One event queue coalesces the input from many physical devices. However, it also imposes a uniform structuring (events) on the typically unstructured physical input as well as filtering and the other event queue system features described above in the design based on this structured input. In the tty system, there is a one to one relationship between a physical device at the bottom (the terminal), and the device interface that the client sees. In the event queue, there are many physical devices which enter events in the queue, and the client sees a single device interface (the event queue) at the top.

The event queue subsystem provides a set of procedures and data structures which a device driver, or client (user process) may access. An event queue provides a set of logical devices with fixed semantics. A given logical device abstracts from the large variety of physical devices which may possibly implement it. The event queue subsystem implements several pseudo devices - each an independent event queue. These are implemented as different minor devices of the event queue major device. The event queue system has been implemented in the Ultrix system on a large VAX[6] system (11/785) with a large variety of attached input devices and more recently on MicroVAX II workstations.

Extensibility

The Event Queue is easily extensible to accommodate new types of devices and events. New event types can be added to the queue either by adding a new physical or 'virtual' device which puts these events in the event queue. This is described in the following sections.

New Physical Devices

The Event Queue System provides a single canonical interface by which device drivers dispatch input to it. This interface: **queInput()** is analogous to the **ttyInput()** routine in the UNIX tty system module. With **queInput()** however, an *event* is dispatched to the Event Queue System. Recall that *ASCII characters* are dispatched to the UNIX tty system with **ttyInput()**. The usage of **queInput()** in a driver or device dependent module which dispatches input to the Event Queue System is very similar to the way that a serial line driver dispatches a character to the Unix tty system.

If a new input device can be attached to a serial line interface which works with the UNIX terminal system it requires no change to the existing standard device driver. Instead a simple line discipline for the new input device is written. This discipline understands the protocol sent by the input device over the serial line and dispatches events to the event queue by calling **queInput()**. The line discipline for an input device is activated in the usual way using the **ioctl(2)** interface. In the case of an input device connected via a special hardware interface board, a device driver must be written. This driver dispatches input directly to the Event Queue System with **queInput()**.

[6] VAX, MicroVAX and Ultrix are trademarks of Digital Equipment Corporation.

New Virtual Devices

A 'virtual device' is a client process that synthesizes one or more event type. Such a client uses the **postevent()** request for this purpose. This client interface resolves to the same function within the Event Queue System as the bottom end interface **queInput()**.

An example of a virtual device is a window manager which enters events in the input queue relating to windows: window entry/exit, exposure, iconization, and the like. The event queue allows any client process to enter a sequence of events in the queue as an atomic operation. This ensures proper synchronization of these events with events being generated by other devices in the system.

Retrospective

The UNIX terminal system

The designers of UNIX recognized the value of modularizing and abstracting fundamental system components. A successful system component is high level enough to be independent of the various physical devices that it can support, but low level enough to serve the needs of a variety of clients and applications. In UNIX a large part of the terminal input system is implemented as a general purpose system module which layers on top of serial line device drivers. The terminal system then extends a higher level interface to clients which they may then build further upon to meet specialized application needs.

The UNIX system addresses the traditional problem of interactive timesharing. The associated terminal input system addresses the problem of interactive terminal handling for ASCII character cell displays. Device drivers reconcile their input to a simple canonical interface provided by the bottom end of the tty system, and more sophisticated high level semantics are imposed by the same code independent of the hardware device involved. This design separates device function from the higher level terminal input abstraction nicely: The device driver's job is kept simple, and consistent terminal semantics are guaranteed across devices by sharing the implementation of the higher level functions in the tty module.

Workstation Input Systems

On contemporary workstation systems, the input problem is more general than that on traditional timesharing systems. There is a greater diversity of input devices, and this makes it somewhat more difficult to get a unifying abstraction for input. Although most workstation input system designs have embraced the event queue paradigm at some level as a unifying paradigm, none have effectively isolated this subsystem from a particular set of physical devices on the one hand or a high level application (typically a window system or manager) on the other, or both.

On many workstations to date it has been possible to escape with a specialized input system whose design or implementation is committed to a particular hardware subsystem or high level application. This is design approach is motivated by workstations where a limited number of input devices are possible and they are all attached to the system via a single hardware interface. This approach however, has the significant problem that it does not decouple the important component of the system - the event queue, which provides the unifying abstraction for input, from the low level device-specific components. Much of the value of the event queue abstraction is lost if this function must be duplicated or modified each time we wish to implement the system on a new physical device.

Principal Problems and their Resolution

The Event Queue System is based on the premise that interactive input is a fundamental system function, and that as such, a general purpose system component should be available to address this function. The principle problems of many previous designs are their specialization to the needs of a given

application, committment to a particular hardware interface or subsystem or their presumption of a predetermined set of input devices. The Event Queue System overcomes these problems by separating out a large general purpose component of the input system - the event queue itself, and a set of abstract functions relating to the handling of events and the queue. The separation of application specific function from the Event Queue System allows it to be used in many different applications. The exclusion of device specific function and the provision of an abstract bottom-end interface that devices call, means that the Event Queue System need not be changed either to incorporate input from new input devices or to accommodate new hardware interfaces.

The event queue system has been implemented in the way described above, with the vast majority of the event queue code in a system module, and a small amount of preprocessing/interface code in device-specific modules. As a result it has been possible to incorporate new physical input devices with minor effort and without modification to the event queue system module. Device-specific modules for a new device fabricate events which they dispatch to the bottom end event queue input interface. Device dependent modules do not implement the event queue itself or support the associated client level functions directly.

Related Work

An event queue of some form is found in any respectable workstation system. We have looked in varying detail at those in X version 10 [Gett86], Sun Windows version 2 [Sun 85], MS Windows [Micr85a, Micr85b, Micr85c], the Apollo Domain release 8 [Apol84], and the Silicon Graphics Iris version 2 [Sili84] to mention some. The design presented here derives a great deal from the input system on the Iris workstation[7]. That design was in turn based on ideas developed still earlier by Bob Sproull [Newm79] and used in the Alto and subsequent D machine workstations at the Xerox Palo Alto Research Center. Some close exposure to the input system in the X window system (versions 7 through 10) raised several problems. A desire to resolve these problems was the principal stimulation for this design.

Acknowledgements

We are indebted to a number of people for designs that motivated the present one, and for the ideas which it derives from them. The software engineering group at Silicon Graphics was principally responsible for several interfaces which have been absorbed with little change in the present design. The present design however, extends that design in several ways, and the implementation context and strategy is fundamentally different.

This work was started at the DEC Workstation Systems Engineering lab in Palo Alto. Ray Drewry, Brian Kelleher and Mike Bidun of that group provided valuable early discussion of the ideas. John Danskin used the first prototype implementation of the system and made helpful suggestions for its improvement. We are especially grateful to Paul Haeberli of Silicon Graphics, who has provided an enduring interest in this and related work and given us many valuable systems ideas. Special thanks to Steve Bourne, also at DEC, for creating an environment for this sort of work, and for continued support and encouragement.

[7] Iris is a trademark of Silicon Graphics Inc.

References

[Apol84] Apollo Computer Inc., *Programmer's Guide to DOMAIN Graphics Primitives*, Apollo Computer Inc., Chelmsford, Massachussetts USA, April 1984.

[Bowe87] Jonathan P. Bowen and David J. Brown, *Formal Specification of the Event Queue: An Extensible Input System for UNIX Workstations*, Programming Research Group, Oxford University, Oxford, England, March 1987.

[Brow86] David J. Brown, *Input: Workstation Systems Technical Memo No. 2*, DEC Workstation Systems Engineering, Palo Alto, California USA, March 1986.

[Gett86] J. Gettys, R. Newman, and A. DellaFera, *Xlib - C Language X Interface, Protocol Version 10*, MIT Project Athena, Cambridge, Massachussetts USA, January 1986.

[Haye87] I. J. Hayes, *Specification Case Studies*, Prentice-Hall International Series in Computer Science, 1987.

[King87] S. King, I. Sorensen, and J. Woodcock, *Z: Concrete and Abstract Syntaxes Version 1.0*, Programming Research Group, Oxford University, Oxford, England, January 1987.

[Micr85a] Microsoft Inc., *Microsoft Windows Programmer's Guide, Beta Release*, Microsoft Inc., Bellvue, Washington USA, May 1985.

[Micr85b] Microsoft Inc., *Microsoft Windows Reference Manual, Beta Release*, Microsoft Inc., Bellvue, Washington USA, May 1985.

[Micr85c] Microsoft Inc., *Microsoft Windows Adaptation Guide, Beta Release*, Microsoft Inc., Bellvue, Washington USA, May 1985.

[Morg86] C. C. Morgan and B. A. Sufrin, "Specification of the UNIX file system," *IEEE Transactions on Software Engineering*, vol. 10, no. 2, March 1984.

[Newm79] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics 2nd Ed.*, McGraw Hill, New York, N.Y. USA, 1979.

[Sili84] Silicon Graphics Inc., *Iris Workstation Programmer's Guide*, Silicon Graphics Inc., Mountain View, California USA, 1984.

[Spiv86a] J. M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Programming Research Group, Oxford University, Oxford, England, 1986. D Phil Thesis

[Spiv86b] J. M. Spivey, *The Z Library - A Reference Manual*, Programming Research Group, Oxford University, Oxford, England, August 1986.

[Sufr86] B. A. Sufrin, *Z Handbook Draft 1.1*, Programming Research Group, Oxford University, Oxford, England, March 1986.

[Sun 85] Sun MicroSystems Inc., *Programmer's Reference Manual for SunWindows*, Sun MicroSystems Inc., Mountain View, California USA, April 1985.

[Wood86] J. Woodcock, *Structuring Specifications - Notes on the Schema Notation*, Programming Research Group, Oxford University, Oxford, England, October 1986.

EUUG conference, May 1987, Helsinki.

Title of paper:

THE EVENT QUEUE: An Extensible Input System for UNIX Workstations

Names and addresses of authors:

Jonathan P. Bowen
Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road, Oxford OX1 3QD, England
Tel: +44-865-273852 (Sec: +44-865-273840)
`bowen@uk.ac.oxford.prg` (JANET)

David J. Brown
Cambridge University Computer Laboratory
Corn Exchange Street, Cambridge CB2 3QG, England
Tel: +44-223-354606 (Sec: +44-223-354600)
`djb@uk.ac.cam.cl` (JANET)
&
Digital Equipment Corporation
Workstation Software Engineering
100 Hamilton Avenue, Palo Alto, California 94301, USA
Tel: +1-415-853-6723 (Sec: +1-415-853-6700)
`djb@decwrl.dec.com` (ARPA)

APPENDIX

The Z specification language is used throughout this appendix to formally describe the operation of the system. An abstract state of the system is presented, and then the effect of individual library routines on the state is given. The corresponding C declarations in the main part of the paper may be compared to aid those familiar with the C programming language. The amount of English description which would normally be included in a typical Z document has been reduced here on the assumption that the reader will have read the main part of the paper first.

Basic Concepts

Activity on input devices can cause entries to be made in the event queue. An event is represented by a device identifier and an associated value.

```
┌ Event ──────────────────
│  device : qDevice
│  value  : qValue
└─────────────────────────
```

At present there are three types of device: button, valuator and keyboard. Valuators may be absolute or relative.

```
Button, RelValuator, AbsValuator, Keyboard : qType

Valuator  ≙  { AbsValuator, RelValuator }
```

The system includes a number of devices, each of which has a `type` and a `value`. There is also a `delta` resolution which is only meaningful for valuator devices.

```
┌ Device ─────────────────
│  type  : qType
│  value : qValue
│  delta : short
└─────────────────────────
```

All devices return a `qValue`. Button devices return `boolean` (true/false) values. Valuator devices (e.g. a mouse or clock) have a range value associated with them. They may be relative or absolute. For example a mouse could have two associated devices returning relative and absolute positions respectively. Keyboard devices return character code values.

Abstract State

The state of the system includes a finite number of devices. These devices may be enabled and disabled. A sequence of events awaits processing by the client. A number of devices may be bound to a device. A pair of devices may be associated with the x and y coordinates of the cursor. Zero is used to indicate an invalid device.

```
┌ State ─────────────────────────────────────
│  devices  : qDevice ⇸ Device
│  enabled  : 𝔽 qDevice
│  events   : seq Event
│  bindings : qDevice ⇸ seq qDevice
│  cursor   : qDevice × qDevice
├──────────────────
│  0 ∉ dom devices
│  enabled ⊆ dom devices
│  dom bindings = dom devices
└────────────────────────────────────────────
```

After initialization, much of the state is zero or empty. A number of devices are configured in the system, but there are no enabled or bound devices.

```
┌ InitState ──────────────
│  State'
├──────────────────
│  devices' ≠ ∅
│  enabled' = ∅
│  events'  = ⟨⟩
│  ran bindings' = {⟨⟩}
└──────────────────────────
```

Operations change the state of the system. However the device types always remain the the same. They are set at initialization time.

```
┌ ΔState ─────────────────────────────────────────
│  State
│  State'
├──────────────────
│  dom devices' = dom devices
│  ∀ dev:qDevice | dev ∈ dom devices •
│      devices'(dev).type = devices(dev).type
└─────────────────────────────────────────────────
```

Some operations do not affect the state. (Note that in practice the actual device values change asynchronously but this does not affect the abstract specification).

```
≡State  ≙  ΔState | θState' = θState
```

For most operations, only a small part of the state is changed. The following *framing schemas* are useful of the definition of subsequent operations, by specifying that all but one of the state components is unaffected.

```
ΦDevice    ≙  ΔState ∧ ≡State\devices
ΦEnable    ≙  ΔState ∧ ≡State\enabled
ΦEvent     ≙  ΔState ∧ ≡State\events
ΦBinding   ≙  ΔState ∧ ≡State\bindings
ΦCursor    ≙  ΔState ∧ ≡State\cursor
```

Asynchronous Events

Device values may change asynchronously. Button values flip between 0 and 1. For valuator devices, the change is not less than the delta resolution of the device. Keyboards only return ASCII characters.

```
┌ AsyncChange ──────────────────────────────
│  ΦDevice
│  device : qDevice
│  Device
│  Device'
├──────────────
│  Device = devices(device)
│  type' = type
│  delta' = delta
│  type = Button       ⇒ value' = not(value)
│  type = AbsValuator ⇒
│             abs(value'-value) ≥ delta
│  type = RelValuator ⇒ value' ≥ delta
│  type = Keyboard     ⇒ value' ∈ char
│  devices' = devices ⊕ {device ↦ Device'}
└───────────────────────────────────────────
```

If the device is enabled, this causes an event. This is added to the end of the event queue together with any associated enabled bound device events. The corresponding current device value is recorded in each event.

```
┌─ QueueEvent ──────────────────────────────
│  ΦEvent
│  Event
│  bound_devs   : seq qDevice
│  bound_events : seq Event
├──────────────
│  device ∈ enabled
│  value = devices(device).value
│  events' = events ⁀ ⟨Event⟩ ⁀ bound_events
│  bound_devs = bindings(device) ↾ enabled
│  #bound_events = #bound_devs
│  ∀ i:ℕ | i ∈ dom bound_devs •
│      bound_events(i).device = bound_devs(i) ∧
│      bound_events(i).value =
│          devices(bound_devs(i)).value
└──────────────────────────────────────────
```

An asynchronous event consists of an asynchronous change in a value of a device followed by the addition of the event and bound events if the device is enabled.

$$\mathsf{AsyncEvent} \;\widehat{=}\; \mathsf{AsyncChange} \;\mathbin{;}\; (\Xi\mathsf{State} \oplus \mathsf{QueueEvent})$$

The notation $\mathsf{AsyncEvent}^*$ is used to denote an arbitrary number of consecutive asynchronous events connected using *schema composition*. We can define

$$\begin{aligned} \mathsf{AsyncEvent}^0 &\;\widehat{=}\; \Xi\mathsf{State} \\ \mathsf{AsyncEvent}^1 &\;\widehat{=}\; \mathsf{AsyncEvent} \\ \mathsf{AsyncEvent}^2 &\;\widehat{=}\; \mathsf{AsyncEvent} \mathbin{;} \mathsf{AsyncEvent} \end{aligned}$$

and so on. Using these definitions,

$$\mathsf{AsyncEvent}^* \;\widehat{=}\; \mathsf{AsyncEvent}^0 \lor \mathsf{AsyncEvent}^1 \lor \mathsf{AsyncEvent}^2 \lor \ldots$$

These asynchronous events may be considered to occur in sequence with requests invoked by the client. Each schema operation may be considered as being *atomic*.

Device Types

Each device has an associated type identifier. It is possible to determine the type of a device using of a number of requests. A framing schema may be used to capture the general features of these requests. Then each request is defined in terms of this

schema.

```
┌─ φIs ──────────────────────────────────────────────
│   ΞState
│   device? : qDevice
│   qtype   : 𝔽 qType
│   is      : boolean
├──────────────────
│   devices(device?).type ∈ qtype ⇒ is = TRUE
│   devices(device?).type ∉ qtype ⇒ is = FALSE
└────────────────────────────────────────────────────
```

```
isbutton    ≙  φIs[isbutton!/is]    | qtype = {Button}
isvaluator  ≙  φIs[isvaluator!/is]  | qtype = Valuator
isrelative  ≙  φIs[isrelative!/is]  | qtype = {RelValuator}
isabsolute  ≙  φIs[isabsolute!/is]  | qtype = {AbsValuator}
iskeyboard  ≙  φIs[iskeyboard!/is]  | qtype = {Keyboard}
```

Queuing & Dequeuing Input Events

When a device value changes, an event is entered in the input event-queue if this device has been selected for queueing by the client with a `qdevice` request.

```
┌─ qdevice ──────────────────────────
│   φEnable
│   device? : qDevice
├──────────────────
│   enabled' = enabled ∪ {device?}
└────────────────────────────────────
```

Once a device has been selected by `qdevice`, an event will be placed in the event queue whenever the device changes value. Devices which have not been queued will not cause events to be entered in the event queue. Device events which are currently being queued may be disabled.

```
┌─ unqdevice ────────────────────────
│   φEnable
│   device? : qDevice
├──────────────────
│   enabled' = enabled \ {device?}
└────────────────────────────────────
```

Event Filtering

Certain devices (such as a clock or mouse) may be high resolution or 'noisy' and can thus generate more events than the client actually wishes to see. The `threshold` request allows the client to specify that a minimum change must occur in the device's value before an event is entered in the queue.

```
┌ threshold ─────────────────────────────
│  ΦDevice
│  device? : qDevice
│  delta?  : short
│  Device
│  Device'
├─────────────────
│  Device = devices(device?)
│  type' = type
│  value' = value
│  delta' = delta?
│  devices' = devices ⊕ {device? ↦ Device'}
└─────────────────────────────────────────
```

Reading Events

The client obtains the next input event from the event queue with the `getevent` request.

```
┌ GetEvent ──────────────────
│  ΦEvent
│  pValue!   : qValue
│  getevent! : qDevice
│  Event
├─────────────────
│  Event = head events
│  pValue! = value
│  getevent! = device
│  events' = tail events
└────────────────────────────
```

This blocks until an event is available. Hence at least one event must be available in the queue.

```
getevent  ≙  [AsyncEvent* | #events' ≥ 1] ⨾ GetEvent
```

For efficiency we are often interested in reading a number of events in one procedure invocation. The getevents request performs this function.

```
┌ GetEvents ──────────────────────────────────
│  ΦEvent
│  count?      : int
│  EventArray! : int ⇸ Event
│  getevents!  : qDevice
├──────────────────
│  EventArray! = succ ⨾ (1..count? ◁ events)
│  getevents! = head(events).device
│  events' = (count?+1..#events) ↿ events
└─────────────────────────────────────────────
```

Getevents returns the device associated with the first event and an array of events. The request does not return until all count? requested events have occurred. Hence, similarly to getevent, a number of events may be necessary beforehand.

```
getevents  ≙  [AsyncEvent* | #events' ≥ count?] ⨾ GetEvents
```

Posting Input Events

Events may be 'posted' (i.e. placed) at the end of the input queue. A list of devices and associated values must be supplied. These are added atomically and in the order given.

```
┌ postevent ──────────────────────────────────────────
│  ΦEvent
│  count?   : int
│  pDevice? : int ⇸ qDevice
│  pValue?  : int ⇸ qValue
├──────────────────
│  events' = events⁀{ s : seq Event | #s = count? ∧
│                      ∀ i:1..count? •
│                        s(i).device = pDevice(i-1) ∧
│                        s(i).value  = pValue(i-1)}
└─────────────────────────────────────────────────────
```

Testing the Input Queue

Since the getevent request is synchronous (i.e. it will suspend the caller until there is an event in the input queue to be read), the client may not wish to use it. The qtest request allows the client to determine whether there is an event in the queue.

```
┌ qtest ─────────────────────────────────────
│ ΞState
│ qtest! : qDevice
├──────────
│ events ≠ ⟨⟩ ⇒ qtest! = head(events).device
│ events = ⟨⟩ ⇒ qtest! = 0
└────────────────────────────────────────────
```

Qtest returns the device id associated with the first input event in the queue, or 0 if the queue is empty. The client may use qtest to provide a non-blocking use of the input queue. This is implemented by using qtest to see that there is an event and only then performing a getevent or getevents if input is available.

Grouping Input Events

It is often desirable to observe the state of several devices when a particular event occurs. With the connectevent request the client directs the system that another input event is to be placed in the event queue when a particular event occurs.

```
┌ connectevent ──────────────────────────────
│ ΦBindings
│ device?,
│ bound_device? : qDevice
│ connectevent! : int
│ bound_devs    : seq qDevice
├──────────
│ bound_devs = bindings(device?)
│ #bound_devs < MaxBindings ⇒
│     bindings' = bindings ⊕
│         {device? ↦ bound_devs ⁀ ⟨bound_device?⟩} ∧
│     connectevent! < 0
│ #bound_devs ≥ MaxBindings ⇒
│     bindings' = bindings ∧
│     connectevent! > 0
└────────────────────────────────────────────
```

When an event occurs on device?, an event will be placed in the event queue for that device, and for each bound device in the order that the devices were originally bound by successive calls to connectevent. connectevent returns an error code less than 0 if it was unable to honor the request and positive non-zero if successful. Subsequently, it may be necessary to undo the binding between two devices.

```
┌─ disconnectevent ──────────────────────────────
│  ΦBindings
│  device?,
│  bound_device?      : qDevice
│  disconnectevent!  : int
│  bound_devs        : seq qDevice
├──────────────────
│  bound_devs = bindings(device?)
│  bound_device? ∈ ran bound_devs ⇒
│       bindings' = bindings ⊕ {device? ↦
│            bound_devs↾(qDevice\bound_device?)} ∧
│       disconnectevent! ≠ 0
│  bound_device? ∉ ran bound_devs ⇒
│       bindings' = bindings ∧
│       disconnectevent! = 0
└────────────────────────────────────────────────
```

The request returns non-zero on success, and 0 for failure, if the bound_device? was not bound to the device? given.

Cursor Position

The cursor is closely related to the event queue mechanism. The cursor must derive its current position based on two input devices (e.g. the x-y coordinates of a mouse). The attachcsr request allows the client to specify which pair of valuators will be used to determine the cursor position.

```
┌─ attachcsr ─────────────────────
│  ΦCursor
│  xdevice?,
│  ydevice? : qDevice
├──────────────────
│  cursor' = (xdevice?, ydevice?)
└─────────────────────────────────
```

Function Definitions

The following non-standard functions are assumed in the specification given:

```
│   not,
│   abs : Z → Z
├──────────────────────
│   ∀ i:Z •
│         i=0 ⇒ not(i) = 1
│         i≠0 ⇒ not(i) = 0
│         i<0 ⇒ abs(i) = 0-i
│         i≥0 ⇒ abs(i) = i
```

Type Definitions

The types used in this Z specification are all ranges of integers and are currently implemented as follows:

FALSE ≙ 0
TRUE ≙ 1
boolean ≙ { FALSE, TRUE }

char ≙ $0..2^{7}-1$
short ≙ $-2^{15}..2^{15}-1$
int ≙ $-2^{31}..2^{31}-1$

unsigned_char ≙ $0..2^{8}-1$
unsigned_short ≙ $0..2^{16}-1$
unsigned_int ≙ $0..2^{32}-1$

qDevice ≙ unsigned_short
qValue ≙ short
qType ≙ unsigned_char

A more flexible approach would be to allow qValue to either be a single value, or a sequence of values. We could define qValue as a labelled disjoint union and update the specification given accordinging.

```
qValue   ::=  val <<short>> | str <<seq char>>
```

In practice, this could consist of boolean flag followed by either a single value or a length and a string.

Z Glossary (*Not* comprehensive)

a, b	identifiers	f, g	functions
x, y	terms	m, n	numbers
p, q	predicates	s, t	sequences
A, B	sets	S, T	schemas
Q, R	relations		

Notation	Meaning
a ≙ x	Syntactic definition
a::=b<<x>>|...	Data type definition
¬p	Logical negation
p ∧ q	Logical conjunction
p ∨ q	Logical disjunction
p ⇒ q	Logical implication (¬p∨q)
p ⇔ q	Logical equivalence
∀a:A;...|p•q	Universal quantification
∃a:A;...|p•q	Existential quantification
x = y	Equality of terms
x ≠ y	Inequality (¬(x=y))
x ∈ A	Set membership of a set
y ∉ A	Non-membership (¬(x∈y))
∅	Empty set
A ⊆ B	Set inclusion
A ⊂ B	Strict set inclusion
{x, y, ...}	Set of elements
{a:A;...|p...}	Set comprehension
(x, y, ...)	Ordered tuple
A×B×...	Cartesian product
ℙ A	Power set (set of subsets)
𝔽 A	Set of finite subsets
A ∩ B	Set intersection
A ∪ B	Set union
A \ B	Set difference
#A	Size of a finite set
A ↔ B	Relation (≙ ℙ A×B)
a ↦ b	Maplet (≙ (a, b))
dom R	Domain of relation
ran R	Range of relation
id A	Identity function
Q ⨾ R	Foward relational composition
A ◁ R	Domain restriction
A ⩤ R	Domain corestriction
A ▷ R	Range restriction
A ⩥ R	Range corestriction
R⦇A⦈	Relational image
R^{-1}	Inverse of relation
R^{*}	Reflexive transitive closure
R^{+}	Transitive closure
R^{n}	Relation composed n times
A ⇸ B	Partial function
A → B	Total function
A ⇻ B	Finite partial function
f(x)	Function application
f ⊕ g	Overriding ((dom g⩤f) ∪ g)
ℕ	Set of natural numbers
ℤ	Set of integers
succ n	Function {0↦1, 1↦2, ...}
m + n	Addition ($succ^{n}$ m)
m − n	Subtraction
m ≥ n	Greater than or equal
m . . n	Range {i:ℕ|n≥i≥m}
seq A	Set of finite sequences
⟨⟩	Empty sequence
⟨a, b, ...⟩	Sequence {1↦a, 2↦b, ...}
s ⁀ t	Sequence concatenation
head s	First element (s(1))
tail s	All but head of sequence
N↿s	Index restriction (N⊆ℕ)
s↾A	Sequence restriction
(schema box: S / a:A;... / p∧...)	Vertical schema definition. New lines denote ";" and "∧". Name and predicates are optional.
S≙[a:A;...|p...]	Horizontal schema
[T...|...]	Schema inclusion
z.a	Selection (given z:S)
θS	Tuple of components
S′	Decoration (after state)
S|p...	Extra predicate(s)
S;a:A...	Extra declaration(s)
S[a/b, ...]	Renaming (new/old)
¬S	Schema negation
S ∧ T	Schema conjunction
S ∨ T	Schema disjunction
S\(a, ...)	Hiding of component(s)
pre S	Precondition of schema
S ⊕ T	Overriding ((S∧¬preT)∨T)
S ⨾ T	Schema composition

```
┌ S ──────────
│ a:A;...
├────────────
│ p∧...
└────────────
```

I Come to Bury UNIX... And to Praise It

Dominic Dunlop

Sphinx Ltd.

ABSTRACT

The UNIX™ operating system is being used increasingly as a vehicle for the delivery of application software. As such, it is usually almost completely buried: only when a user answers the `login:` prompt are they responding to a UNIX utility — all other interaction is with an application layered on top of UNIX.

But how deep and impervious should this layering be? What aspects of UNIX can form a useful part of an application while still avoiding the need to expose the user to such self-evident command strings as `lp -dletter -onb -oletenv -n2`?

This paper, using examples taken from actual applications packages, explores a range of responses to this question.

THE CHANGING ROLE OF UNIX

UNIX has come a long way since all its users were so pleased to get a slice of a computer to play with that they were prepared to put up with its terse — not to say obscure — command language because of the power that it put at their fingertips. And it's come a long way since users who found that terseness and obscurity too much could be sure of ready access to a guru who could solve their every problem with a few esoteric keystrokes. These days, the majority of UNIX systems are sold to users who expect a computer to solve their problems, not present them with new ones. Show these users a dollar-sign prompt, and they'll be on the phone to their supplier demanding assistance. That dollar sign can be taken as an indicator of support costs!

Thus, although UNIX is an excellent vehicle for the delivery of many types of application system, it must be buried in order that it does not frighten users. This paper takes a number of products which are successful in the UNIX environment and examines the methods they use to hide UNIX, and how successful those methods are. Table 1 lists the products, and summarises their functions. Note that almost all of them are available in several environments, decreasing the reliance that their authors can place on the availability of UNIX tools.

TABLE 1. Example Products

Package	Operating systems	Description
Informix-4GL	PC-DOS UNIX VMS[1]	C-like language for the development of data-based applications requiring menus, screen forms, and report generation
Informix-SQL	PC-DOS UNIX VMS[1]	Data base manager with subsystems for screen forms, report generation, and simple menus
Q-Office	PC-DOS[2] UNIX	Office automation package providing word-processing, and many (sometimes optional) extras such as menu and forms handling, electronic mail, free text retrieval...
Tetraplan	PC-DOS UNIX	Reasonably comprehensive British accounting package (seventeen modules, and growing fast)
Uniplex-2 Plus	UNIX	Office automation package incorporating word-processing, database management, spreadsheet, electronic mail and configurable menus

APPLICATIONS AREN'T LIKE UTILITIES

The first problem that all of these applications have to face is that, unlike most UNIX utility programs, they are screen, rather than line oriented. What's more, they have to be terminal-independent because no package that assumes it's talking to one particular type of terminal is going to get very far in today's diverse markets. Of course, UNIX has a means of solving this problem. It's called *termcap.* Or maybe *terminfo*. And it's seven bits wide, and can't do colour or draw lines, whatever it's called. Faced with these uncertainties and shortcomings, and with the fact that smart-looking screens are a selling point, it's not surprising that most application writers have supplemented the system's basic facilities. Table 2 shows how.

Of the example products, only **Informix-SQL** confines itself confines itself to the capabilities described by *termcap*. As a result, **Informix-SQL**'s screen displays are less attractive than those of **Unify**, a major competitor with its own screen handling library, and sales have been lost as a result. Perhaps this is why **Informix-4GL**, a more recent product, requires that the *termcap* database is augmented in order that additional display features can be accessed.

1. Under development
2. Word-processor only

TABLE 2. Screen-Oriented Features

Product	**Description source**			**Additional features**			**Additional source**
	termcap	*terminfo*	**Other**	**Colour**	**Boxes**	**Other**	
Informix-4GL	•			•	•		*termcap* tweaks
Informix-SQL	•						–
Q-Office			•	•	•	•	–
Tetraplan	•				•	•	"box files", *termcap* tweaks
Uniplex-2 Plus	•	•				•	Configuration file

Like most products, **Informix-SQL** does not yet use *termcap*'s successor, *terminfo*. While *terminfo* undeniably improves start-up time, the fact that description entries must be compiled threatens application writers with headaches that they'd rather avoid if at all possible. Among the listed products, only **Uniplex-2 Plus** uses *terminfo* where it is provided. **Q-Office**, the other office automation (OA) product, dispenses with UNIX solutions entirely, using a private database which describes terminals in considerable detail, but which is totally incompatible with either *termcap* or *terminfo*. In general, the "enhancements" which allow different products to access additional terminal capabilities are all mutually incompatible, making system maintenance tricky if more than one product is installed.

Where does this leave us? UNIX has been the target of standardisation efforts for a number of years [1] [2] [3], yet we've only examined one relatively minor aspect of UNIX, and already we're faced with a situation where real software products need facilities which are years away from being standardised[3].

APPLICATION INTERFACE TO UTILITIES

There are other areas where, at first sight, UNIX provides standard tools which should preclude the need for application writers to reinvent wheels. It usually turns out that real life isn't that simple. Print spooling is a case in point.

a. It used to be done by *lpr*, which was woefully inadequate (unless you had the Berkeley version)

b. Now it's done by *lp*, which is competent, if limited

c. There is no standard way of describing printers' capabilities in the way that *termcap* describes terminals' capabilities (*printcap* never caught on), meaning that each application has to incorporate a back-end which worries about the details of printer control

3. In fact, by the time two-dimensional screen handling is standardised, most software products will probably want two-and-a-half — windows...

d. Some manufacturers ship proprietary print spoolers of greater or lesser competence with their systems, but these can't be relied upon by portable applications.

TABLE 3. Access to Print Spooler

Product	Spooler		Configurable by	Multiple destinations	Back ends
	Own	System			
Informix-4GL	None	Required	Programmer	Maybe	None
Informix-SQL	None	Required	Administrator	Via shell	None
Q-Office	Prefered	With difficulty	Administrator	Yes	Many
Tetraplan	None	Required	Administrator	Via shell	None
Uniplex-2 Plus	Optional	Prefered	Programmer	Yes	Several

Table 3 shows that there's a fair degree of agreement among software products on how this problem should be addressed. They simply punt, leaving the job of tuning the interface between the product and a system's spooler to the person who gets to install the product. Only the OA products acknowledge the issues of multiple destinations and of printer capabilities. Indeed **Q-Office** is delivered with a complete spooling subsystem, which works well on dedicated OA applications, but which rapidly becomes unsupportable if **Q-Office** is co-resident with other packages which expect the print spooler to be called *lp* and to behave appropriately.

ESCAPING FROM APPLICATIONS

If you want to get fancy with printing from the other products, it may be possible to enter a shell command line to get the desired effect, but this flexibility allows users them to shoot themselves in the foot — or the system in the back.

This brings us on to the matter of shell escapes. It's very nice for programmers to be able to enter something like `!ps -ef` in response to a prompt from a menu-driven program, dropping temporarily into shell command language to have a quick poke around the system. Many software products allow an escape to the shell — indeed, some require it in order that they can kick off subordinate programs[4]. It's here that the problem arises: an open-ended shell escape can expose timorous users to things they don't want or need to understand, and can allow malicious users unintended access to things they know only too well. Table 4 shows that, of the products which offer a shell escape, only the OA products allow it to be inhibited by the person who configures the system. With **Informix-SQL** and **Tetraplan**, you're stuck with the shell escape whether you want it or not. (**Informix-4GL** is a

4. You discover this when you try to inhibit the shell escape by setting `SHELL=/bin/true` (or something even more creative) in the environment, and everything stops working.

TABLE 4. Applications as Self-Contained Environments

Product	Escape to UNIX			Suitable as shell?	Menu facility
	Allowed	Required	Optional		
Informix-4GL	Yes	No	Yes	Yes	Tool kit
Informix-SQL	Yes	Yes	No	Yes	Limited
Q-Office	Yes	No	Yes	Yes	Full language
Tetraplan	Yes	Yes	No	Yes	Limited
Uniplex-2 Plus	Yes	No	Yes	Yes	Full language

programming language, so it's up to the programmer whether applications built with it have a shell escape.)

The table also shows that many products can themselves be used as shells. Generally, this means that they must be invoked from a *.profile* which sets up an environment for the product, and has as its last line something like

```
exec Qoffice
```

This works very well until a word processing operator, browsing through the list of documents in their main folder, notices something strange called *.login*, and corrupts it in the process of trying to find out what it is. The next time that person tries to log in, the system won't work for them.

Of course, possibility of this type of error can be minimised if the package follows the UNIX convention for hidden files (both OA products do this), and avoided completely if some tricks are played in *.profile*:

```
...
HOME=/somewhere/else
export HOME
cd
exec Qoffice
```

KICKING OVER THE TRACES

The point is that the user must be prevented from tripping over little bits of UNIX: it must be completely buried. With many application products, UNIX is only half-buried by the package as delivered: it still takes somebody who knows what they're doing to complete the job.

This is particularly true for back-up strategies. As table 5 shows, few applications suppliers even address the issue.

Instead, they tacitly assume that back-ups are somebody else's problem. A reasonable excuse for this state of affairs is the total lack of agreement among system suppliers about back-up methods. Some manufacturers put forward (more or less hacked versions of) *tar* or *cpio*, while others favour *dump*[5], *volcopy*, or

TABLE 5. Provisions for Security Back-Up and Recovery

Product	Back-up method provided	Recovery method provided
Informix-4GL	None	Transaction replay
Informix-SQL	None	Transaction replay
Q-Office	Various	Various Keystroke replay
Tetraplan	*tar*	*tar*
Uniplex-2 Plus	None	None

something proprietary of their own devising. And there's even less agreement about device names. Consequently, it's usually up to a UNIX programmer to implement a back-up strategy and, if necessary, tie it into a menu system so that a turn-key system's users can run it themselves. This is even true for **Q-Office** and **Tetraplan**, the two listed products which include a back-up method. Such is the diversity of system configurations in the field that the person installing the software generally has to fill in some blanks on a menu, or hack a small shell script before back-ups work properly.

UNIX GETS IT RIGHT

So far, we've explored areas where application developers have addressed real or perceived shortcomings in the facilities that UNIX implementations can be relied upon to offer. What about areas where there should be little argument that UNIX can do an adequate job? How well do software products take advantage of such facilities?

An example concerns file access permissions and system security. If the idea is that each file or program should have exactly those permissions which allow selected users to do their job, while preventing others from accidentally or maliciously from treading on their toes, table 6 indicates that most products do a pretty poor job.

(Notice that system utilities *lp* and *mail* have been slipped into the table, showing that it's easy to make mistakes.)

A two-pronged strategy seems to be at work:

1. If you need to get around permissions, use set-uid *root*: it's bound to work!
2. If there's a chance that restricting permissions may stop users from doing anything, leave everything wide open.

---

5. The old *dump* that dumps file systems, not the new *dump* which displays segments of COFF modules...

TABLE 6. Security Characteristics

Product	User for set-uid	User for set-gid	Approach	File permissions	
				installed	user
Informix-4GL	root	informix	Selective	Correct	Safe
Informix-SQL	root	informix	Selective	Correct	Safe
lp	lp	sys	Dumb	–	Fairly safe
mail	–	mail	Smart	–	Safe
Q-Office	root	various	Shotgun	Affected by root's *umask*	Determined by user's *umask*
Tetraplan	–	–	–	Affected by root's *umask*	Wide open
Uniplex-2 Plus	root	informix	Selective	Affected by root's *umask*	Determined by user's *umask*

This is a recipe for creating unsafe and insecure systems, as has been pointed out in [4] and [5], but application developers seem to keep right on doing it. Only the Informix products break out of this sloppy mould[6], using set-uid *root* briefly to push System V's annoying *umask* file size limit out of sight, then set-gid *informix* to create database files accessible only to their owner or users running Informix programs. (These programs check the data dictionary for further access restrictions.)

BUT FIRST, YOU'VE GOT TO INSTALL IT...

It's also interesting to note that Informix products are among the small minority which incorporate a script to perform comprehensive checks of file ownership and access permissions when they are installed. Almost all products have to be installed by the super-user, and too many tacitly assume that the *umask* in effect is sufficiently lax to allow the creation of files with general read and execute permissions.

As with back-up procedures, some of the blame for this situation can be laid at the doors of system manufacturers. While many suppliers have supplemented their UNIX implementations with (more or less friendly, and more or less bomb-proof) methods for installing software packages, there's very little incentive for software authors to use them, unless they're working on a fat OEM contract for a particular manufacturer[7]. The problem is that recommended methods differ so much from

6. **Uniplex-2 Plus** uses the back-end from **Informix-SQL** for database services, and */bin/mail* for mailbox access.

7. Even then, Informix Software's implementations for the 3B1 use AT&T's recommended installation procedure, whereas those for the 3B2 don't...

system to system that most software authors don't think it worthwhile to work out how to use them, or to educate their support staff on recovery from problems in a dozen different procedures. Instead, most packages rely on *tar* or *cpio*, and assume that the person installing the product can enter something like

```
mkdir /usr/informix
cd /usr/informix
cpio -ivcBd < /dev/rfd0ssdd
./install
```

without making too many mistakes[8].

Inevitably, mistakes are made, the phone is picked up, and support is demanded, but that represents a cost suppliers have decided, for the moment, to live with.

CONCLUSIONS

This paper has concentrated on the problems of building real applications on top of UNIX, and so could depress or frighten those who aren't aware of the power of the tools that UNIX provides. In fact, it's that power which gives rise to some of the problems; UNIX presents reasonably complete solutions to many problems not even addressed by other operating systems. As a result, smart programmers build on the foundations that UNIX provides, saving themselves the trouble of implementing subsystems from the ground up.

It's clear that further standardisation of certain aspects of UNIX would greatly aid application developers. Areas highlighted in this paper are

- Access to advanced terminal features
- Multi-stream spoolers with support for printer capabilities
- Back-up methodologies
- Installation procedures for optional software

But developers too must play their part by educating themselves about the facilities that UNIX can provide, and the correct way to use them. Above all, they must bury UNIX deep if users are to see it as a fertile base for applications, rather than as something they'd prefer to avoid.

REFERENCES

1. *IEEE Trial Use Standard 1003.1, Portable Operating System for Computer Environments*, Wiley-Interscience, 1986

2. *UNIX System V Implementation Definition*, second edition, Volumes 1-2, AT&T, 1986

8. It is left as an exercise for the reader to pinpoint the several pitfalls in even this short sequence

3. *X/OPEN Portability Guide*, Volumes 1-5, Elsevier Science Publishers BV, 1987

4. *UNIX System Security*, Patrick H Wood & Stephen G Kochan, Hayden, 1985

5. *How to Write a Setuid Program*, Matt Bishop, ;login: vol. 12, no. 1 (January/February, 1987)

Trademark acknowledgements:

Informix-SQL, Informix-4GL: Informix Software; PC-DOS: International Business Machines Corporation; Q-Office: Quadratron; Tetraplan: Tetra Business Systems; Uniplex-2 Plus: Redwood International; UNIX: AT&T; VMS: Digital Equipment Corporation.

Now UNIX[1] Talks To Me In My Language

Pascal BEYLS
BULL
1, rue de Provence
38432 Echirolles
FRANCE

Bertram HALT
SIEMENS
Charles de Gaulle Strasse 2
D8000 Munchen 83
GERMANY

ABSTRACT

BULL and SIEMENS, 2 major European companies, have jointly achieved the internationalization of UNIX, as defined by the X/OPEN[2] group. This document describes the choices and the implementation.

An approach similar to TERMINFO makes it possible for the user to define his own sets of characters and sort sequences easily and comfortably. The characters are now 8 bits long and encompass all the national characters. Compliance with local conventions such as printing the date, is also provided.

An original solution, based on a new section in the COFF, has made it possible to eliminate any multilingual problems, so that the user may now work in his own language. Naturally, progress still remains to be made in this field.

It is noteworthy that it is two European, non-English companies which are offering a truly European, if not international, UNIX.

1. Unix is registered trademark of AT&T in the USA and other countries.
2. X/OPEN is a licensed trademark of the X/OPEN Group Members.

1. Introduction

UNIX quickly finds certain limitations when used in a non-English environment, such as that found in most European countries.

For example, a French user is immediately bothered by the use of (or rather the absence of) accents. He can either ignore this problem (the most common solution), or try to find a make-shift work-around. The documentation, the print-outs, the error messages are all in English. Even if this does not bother our friends the hackers too much, the same thing cannot be said of the end user, especially when he has bought a machine from a computer manufacturer from his own country. It is a hard pill to swallow that a national constructor does not supply its machines with a user interface in its own language.

Let's take the example of a beginner wanting to print out his first message:

"Bonjour, Hélène"

as others might print out "hello, world". This unlucky person would have enough problems finding a keyboard with accented letters on it, and when he did, he would have problems programming in C, as the two letters é and è take the place of the standard ASCII characters { and }. If he was using nroff, he would have to write:

"Bonjour, H*(e'l*(‘ene"

He would soon decide that those letters really aren't all THAT important and would give up the idea of using them.

Nothing is provided to help the translation process, either. Any software sold in another country must have the source massaged in order to change the messages and then recompiled. Not to mention the fact that date and decimal formats change between countries also. The American 03/10/87 (March 10 1987) is not the same as the German 03/10/87 (October 3 1987). In France, the equivalent of the American $12,345.67 is $12.345,67. (note that the characters "." and "," have opposite meanings!!)

All this contributes to slow down development of UNIX in non-English speaking countries. And yet, there is enormous potential in some of these countries, such as Japan, Europe and the Middle East.

There are 3 principle areas where research is being done.

- enlarging the character sets to include accented characters
- taking into account each language's particularities (word sorting with 'special' characters, for example).
- printing out messages in the user's language

Although the ASCII character set is universal, the same cannot be said of extended character sets (especially 8 bit character sets). Several are already in use:

- ISO 8859/1 for Western Europe
- ISO 8859/2 for Eastern Europe
- ISO 6937 used by the CCITT
- the IBM-PC has 2 character sets
- etc...(let's not forget the Japanese character sets)

These character sets are evidently not compatible and the ordering of the letters (*collating sequence*) is not ascending, as one would hope. For example, é is not between e and f, which means that sorting of text can no longer use *strcmp*(3X) which sorts by ascending bit value, but must use a sort algorithm based on tables, where the order of

each character is given. Sorting should also take into account other particularities, such as the double letter ll in Spanish (the order is l, ll, m).

The way of handling messages in several languages is a group of tools made available to the programmer that lets him build international software. The text printed out by the program can be translated, without modifying the original program. This toolbox to help the programmer must fill the following conditions:

— The work of internationalizing an application should not change the way of implementing a program and should be limited to a small number of new things to do. Thus, a software house should be able to internationalize its products in many languages without having to modify extensively the program source.

— The way of printing out messages in several languages should be transparent to the user.

— Several people on the same system should be able to speak to the computer in their own language simultaneously.

— The toolbox should make it easy to manipulate and translate messages, as well as to exchange message "databases" between users.

1.1 What impact will this have on UNIX ?

The internationalization should be seen as a set of tools and databases that are added on top of UNIX. Several modifications are still necessary to fully implement internationalization:

— the masking of the 8th bit. Since the ASCII codeset uses only 7 bits for encoding characters, utilities use the 8th bit for internal purposes (e.g., *sh*(1) uses this 8th bit to keep track of single quoted strings to prevent evaluation. This work is known as "8th bit clean up".

— Multiple codeset support : New codesets are now available and intend to cover all the possible characters and signs. These are often based on 8 bits.

— Support for national or cultural conventions: the major areas are:

- Collating sequence
- Date and time format
- Printout of numbers and currency units.
- Hyphenation.

— Message presentation : The message presentation is a mechanism which permits program messages to be stored separately from the logic of the program, translated into different languages and retrieved at runtime according to the language of the user. An environment variable *LANG* defines the language, territory and codeset used.

1.2 The X/OPEN definition

The X/OPEN group has defined and published a set of interfaces, defined as "XVS INTERNATIONALIZATION", which contains mainly an enhanced interface definition for standard C library functions and an announcement mechanism [XOPEN87]. This important specification was the basis of the common implementation realised by BULL and SIEMENS. As the X/OPEN group is only committed to defining programming interfaces, the implementation is left up to the individual companies.

2. The Database INTLINFO

The INTLINFO database contains information on the following topics in order to provide the necessary functionality:

1. Information on the code set used,
2. Information on the properties of every code in the code set (character classification),
3. Information that allows a program to correctly sort/collate user data,
4. Information on how to do conversions, for example, the standard conversions *toupper* and *tolower,*
5. Information that can be accessed by some library routine to correctly interpret things like date formats, number representation, and so on.

In order to allow access to all this information in an efficient manner, it is desirable that the database be organized in some fashion and that a program accessing the database has to do as little interpretation of data as possible.

It is therefore necessary to design a language that allows easy entry of all this information and to design and implement a compiler that will translate this source into a binary that fulfills the requirements of fast access and compact storage.

2.1 The Data Base Source

The following general considerations were followed in the design of the language for the database:

1. The database source should, as a first step, only contain ASCII characters because currently UNIX systems supporting any code set other than ASCII are very hard to find. At a later stage, when tools like editors to edit non ASCII source files are commonly available, this restriction may be dropped allowing for a more comfortable entry of data into the database.
2. The database source should be free format. This especially means that "white space" shall have no significance other than as separator for tokens in the input language. This requirement comes mainly from the bad experiences that every UNIX user has had with languages where this is not the case (for example: make, termcap and terminfo sources!).
3. Comments should be allowed wherever possible. This is one of the reasons why the C language pre-processor is used in the current implementation: it gives the user the freedom to intersperse his source with C-style comments and imposed no additional workload on the compiler designer/implementor.
4. The source should be as self documenting as possible.

The source for this database consists of two major parts:

1. The definition of the code set used.
2. The definition of property tables, collation sequences, string tables and conversion tables.

2.1.1 The CODESET section

The definition of the code set used is introduced by the keyword **CODESET** or **EXTENDED CODESET** followed by a name for the code set. **CODESET** implies

an eight bit codeset with the possible extension of some special, non contiguous sixteen bit codes named **double letters**. **EXTENTED CODESET** implies a sixteen bit codeset with no double letters. The name given to the codeset will become the name of the binary file once compilation has succeeded. It must therefore adhere to some convention on the system so that the runtime routines can find the database. As the runtime routines expect the database to be named "$LANG_$TERRITORY_$CODESET" this should be the name given here. Optionally the name of the codeset may be given on the command line using the -o option. This command line name will override the name of the codeset in the database source.

Following this introduction each code is defined by assigning the code's value to an identifier, which may be used to reference the code from then on. This assignment has the form:

Identifier '=' value_list [':' Properties] ';'

The value_list is a comma separated list of values, a value may be given as C-style character constant, in octal, hexadecimal, decimal or ISO notation, or by giving the name of a previously defined code.

Codes may be subdivided in two classes: **simple** and **combined** codes. There are several restrictions that have to be observed when defining codes in the CODESET section:

1. The list of simple codes must contain all codes from code value 0x0 up to and including the code with the highest value defined. This is necessary because the simple codes are not stored in the INTLINFO data base and the runtime access routines assume the existence of all simple codes for speed reasons. The order of definition is free because all values are sorted into ascending machine collation order when the whole code set definition has been read.
2. The list of simple codes may not contain codes with duplicate code values. This is necessary for the same reasons as above.
3. If the codeset is not an EXTENDED CODESET there may be an arbitrary number of definitions for multi-byte codes up to 2^15 or memory limit, whichever is smaller. Combined codes need not have contiguous code values and will be sorted in ascending machine collation order and construct the "double letter table" in the INTLINFO database binary.
4. If the codeset is an EXTENDED CODESET there may be no combined codes (i.e., double letters) and all codes defined are simple codes. (This especially means that rule 1) and 2) apply!)

As the generation of a valid database without the afore mentioned restrictions is not possible all these conditions are thoroughly checked by the compiler and abort compilation after the codeset section has been parsed.

The optional properties part in the above code definition serves to assign default properties to a code. If it is not given, the code is assumed to be defined but illegal. (This is useful for languages that do not know about some letters defined in a standard code set.) Properties take the form of a comma separated list of keywords.

There is a third kind of statement allowed in this section: the (re-)assignment of default properties to an already defined code. This statement takes the form of

Identifier ':' Properties ';'

The CODESET section is terminated by the two terminal symbols END '.'.

The use of the '#include' facility provided in the language because of the use of the C pre-processor is strongly recommended as most of the codes considered contain common code (for example, ASCII or IS 646 or IS 8859) in their lower half and therefore using a common include file will reduce the risk of errors and provide a common name basis for the remainder of the source.

There may only be one definition of a code set and the definition must be the first item in the source file.

2.1.2 The Data Table Section

The data table section consists of a sequence of property, collation, string and conversion tables, containing at least:
1) a default collation table,
2) a default string table,
3) the two code conversion tables "toupper" and "tolower".

2.1.2.1 The Property Table

A "property" is the membership in a character class and can be accessed at runtime by the *nl_ctype*(3C) library functions.

There can be more than one property table. Each property table is introduced by the keyword **PROPERTY**. The default property table, built along with the code set, has the predefined name PROP_DFLT. It is an error to redefine this property table. Names of property tables must be unique throughout the source.

A statement in the property table takes the form of:

Ident ':' Properties ';'

where Ident designates a defined code and Properties is a comma separated list of properties.

Some properties also have effects on the interpretation of characters by various other International UNIX library routines. For example, the property DIPHTONG must be set for diphtongs to collate correctly as diphtongs, the property DOUBLE must be set to correctly recognize the first of a double letter sequence.

The following properties are known:

ILLEGAL	The corresponding code is defined but is not a legal code.
CTRL	The corresponding code is a control code.
NUMERAL	The corresponding code is a number.
UPPER	The corresponding code is an upper case letter.
LOWER	The corresponding code is a lower case letter.
HEX	The corresponding (letter) code represents a hexadecimal digit.
PRINT	The corresponding code is printable.
PUNCT	The corresponding code is a punctuation mark.

SPACE — The corresponding code is a character that prints as space.

DIACRIT — The corresponding code is a diacritical sign. If combined with either UPPER or LOWER: the corresponding code is a diacriticalled letter.

ARITH — The corresponding code is an arithmetic sign.

FRACTION — The corresponding code represents a fraction.

DIPHTONG — The corresponding character is a diphtong. The meaning of diphtong here is somewhat different from the definition used in the grammar of languages having diphtongs. Diphtong for the purposes of the international UNIX database is a defined as a character for which "one to two" collation must be used. (This implies an interdependence with the collation tables.)

DOUBLE — The corresponding code is constructed from two other single byte codes but is to be treated as a single code. (Note: This allows two things: one is the expansion of 8 bit character sets to include double letters (e.g. Ll, ll in Spanish) that collate two to one and the other is the handling of 8/16 bit codes like IS 6937/1.). This property may only be used if the codeset is not a true sixteen bit (EXTENDED) codeset.

GRAPHIC — The corresponding code prints as a graphic symbol. (e.g. arrows, playing card symbols, ...)

CURRENCY — The corresponding code is a currency symbol.

The two most interesting properties in the list above are the properties DIPHTONG and DOUBLE and their combinations for normal and extended codesets. To clarify the interdependence here are some examples (both assuming an underlying 8 bit codeset with extensions, i.e., IS 6937):

1. Definition of a double character
   Assuming the underlying codeset is IS 6937, the German umlaut will be represented by two eight bit codes: The nonspacing diacritical mark "diaresis" followed by the code for the character "a". If this "character" should be collated "two to one" e.g. for standard German dictionary order the correct definition is:

   dia_a = diaresis, a : LOWER, DIACRIT, DOUBLE, PRINT;

   which means that the code "dia_a" has the 16 bit value "c861" (diaresis == c8, a == 61), is a lowercase diacriticalled letter that collates "two to one" and is printable.

2. Definition of a double diphtong character
   Again taking the IS 6937 codeset and the German umlaut , assume that this character should be collated as the letter "a" followed by the letter "e" (German telephone book ordering). Now we have the case that a double letter should collate as two simple letters, essentially a "two to two" collation. The correct definition for this code now is:

   dia_a = diaresis, a : LOWER, DIACRIT, DOUBLE, DIPHTONG, PRINT;

IS 6937 is the typical example of an eight bit codeset that needs to be extended by a noncontiguous range of pseudo sixteen bit characters (not all combinations of diacritical marks and other codes are defined!).

A code with no defined property will be listed as ILLEGAL in the resulting property table.

2.1.2.2 The Collation Table

A collation table starts with the keyword **COLLATION** followed by the name of the collation table. Exactly one nameless collation table must exist. This is the default collation table bearing the internal name "COLL_DFLT". Names of collation tables must be unique throughout the source file and there can be more than one collation table.

The order of statements in the collation section is significant, as every statement (except the last in the list of forms below) opens up a new class of codes with the same primary weight and the primary weight (starting at 1) increases with each statement encountered. This way for example, the fourth statement in the collation section assigns the primary weight four to all the codes named therein.

A statement in the collation section may take one of the following forms:

PRIMARY ':' Ident_list ';'

PRIMARY ':' Ident '-' Ident ';'

PRIMARY ':' REST ';'

Ident '=' '(' Ident ',' Ident ')' ';'

PROPERTY ':' Property_table_name ';'

The meaning of the first construction is to assign the codes designated in the identifier list the same primary weight and ascending secondary weights from left to right. If the identifiers are to be sorted in machine collation order the second form of the collation statement may be used, which will assign ascending secondary weights for ascending machine collation order.

The third type of statement will set the primary weight of codes not explicitly named in the collation section to the ordinal number of the statement in the collation section, the secondary weight will ascend in ascending machine collation order. This is a convenient notation for defaulting unspecified codes to collate after or before all others.

The fourth form of the collation statement is reserved for the collation of diphtongs (one to two collation) and implies the following: The left hand side code collates as if it were the first right hand code followed by the second right hand code. Please note that in order for the diphtong collation to work correctly, the code named on the left hand of the statement **must** be marked as DIPHTONG in at least one property table and that this property table, if it is not the default table must be given by means of the fifth statement which allows the runtime routines to load a collation only property table if this collation is chosen.

If a code is not given weights in the collation section this code is treated as if it had the (otherwise illegal) primary and secondary weight zero. The net effect of this is that such a code will collate as if non existent. (For example, if '-' is such a code then "abc" and "a-b-c" will collate equal!)

To solve the problem of two to one collation (i.e., two letters collate as one) the double letter must be named in the code set and then can be given a weight in the collation section.

2.1.3 The String Table

There can be more than one string table in a source file.

A string table starts with the keyword **STRINGTABLE** followed by the name of the string table. Exactly one nameless stringtable must exist. This is the default string table, internally named "STRG_DFLT". Names of string tables must be unique throughout the source.

Each statement in a string table looks like

Ident '=' value_list ';'

where Ident is an identifier, i.e., the name of the string and the value list is a comma separated list of strings, character constants and Identifiers designating codes. This allows inclusion of non ASCII codes in any string table value by giving the name of the code in the value_list.

The compiler will check the existence of all codes in the value list, compiling escaped characters in string constants in the process.

If the advice of using include files in the code set section was followed and the names for characters are the same in the include files, the string table can be copied (or again included) in several sources.

2.1.4 The Conversion Tables

In an environment where more than one code set is supported the possibility to convert from one codeset to another becomes of prime importance.

There are essentially two types of possible conversions:

1. conversion within the codeset itself. This type of conversion will hereafter be called a **code conversion**. Examples are the standard conversions "toupper" and "tolower".
2. conversions from one code set into another. This type will hereafter be called a **string conversion**. Examples include the conversion from ASCII to EBCDIC or from IS 8859/1 to IS 6937 and more.

A conversion table either starts with the keywords **CODE CONVERSION** or **CONVERSION**, or **STRING CONVERSION**. The first two are equivalent. Each conversion must be given a name with which it will be accessed at runtime. The two code conversions **toupper** and **tolower** are mandatory. Names of conversions must be unique throughout the source file.

2.2 The Compiler ic

The **ic** compiler was realised using the powerful tools available in UNIX

- C-like comments, file inclusion and defining as well as conditional compilation are handled through use of the standard C pre-processor (usually */lib/cpp*) which can be called either as a separate pass or connected to ic via a pipeline (-DPIPE compile time option).
- The lexical analyser was written using the *lex* lexical analyser generator.
- The parser was written using the UNIX tool *yacc*. This assures that changes in the syntax of the compiled language can be easily accommodated, which is of great value for a prototype compiler.

The C subroutines doing the actual work are distributed over thirteen source files (one for each section [cod.c, prp.c, col.c, frm.c, cnv.c], three for message/error handling [message.c, yyerror.c, yywhere.c], one for I/O [io.c], generally used subroutines [subr.c], symbol table handling [sym.c] and value handling [val.c]) all collected in a library [libic.a].

The main program [main.c] does the necessary initialisations and then calls the yacc generated parser yyparse.

The general strategy employed to construct the INTLINFO data base is the following: To enable the compiler to run even on small machines, each section, except for the CODESET, is written to a temporary file as soon as it is completely parsed. Identifiers and values only used locally in the table are deleted.

This results in a uniform structure throughout the parser: Each section has its own initialisation routine, some functions that are called for each statement to build the table and a finish-up routine that writes the table to the temporary file and does the cleanup necessary.

If no errors have occurred during compilation the final routine in the compiler then writes out the database header and the code set and appends all intermediate files generating the indices for the sections in the process.

Generally the error handling in yacc generated parsers is quite unsatisfactory (as can be seen in pcc, make, awk...). To improve on the error messages the technique described in [SCH84] was used, which resulted in a fairly stable and reliable product, which usually will give quite concise messages ([warning], [error nn]) for the most common errors and recover well from almost any error. One class of errors ([fatal error]) intentionally leads to a premature termination of the compilation: Specific errors like missing or duplicate codes in the CODESET section would result in so many follow up errors in the other sections that termination seems appropriate. There also is a "this cannot happen" error type ([fatal bug]) which also results in a premature termination of the compilation process, leaving temporary files in /tmp that then may be used for debugging.

Special attention was given to the debugging facilities in the compiler. There are several global variables which when set will give tracing/debugging information up to a very deep level. The debug code is included by setting -DEBUG and/or -DYYDEBUG at compile time. The routines necessary for debugging are found in the file dbg.c and, for specific subsections in the appropriate file for the section.

2.3 Example of the input to the database

```
/*
 * partial example of source for
 * an International UNIX database
 */
CODESET CH_ASCIIPLUS :
        /* ^ This will be the name of the INTLINFO file */
#include "ISO646"
        /* ^ include predefined ASCII code definition */

        /*
         * additional definitions for demonstration purposes:
         *
         * first we have a range of secondary control codes.
```

```
 * This is not enforced by the ic compiler nor by
 * the language but is a common IS 2022 style
 * code set extension technique. Note that because
 * there are no properties defined below all these
 * codes are defined but not legal.
 */
sc00 = 0x80; sc01 = 0x81; sc02 = 0x82; sc03 = 0x83;
sc04 = 0x84; sc05 = 0x85; sc06 = 0x86; sc07 = 0x87;
sc08 = 0x88; sc09 = 0x89; sc0a = 0x8a; sc0b = 0x8b;
sc0c = 0x8c; sc0d = 0x8d; sc0e = 0x8e; sc0f = 0x8f;

/*
 * now come some more useful code definitions. These
 * definitions are taken from the IS 8859/1
 * definition. Note the convention of writing upper
 * case letters in all upper case, lower case
 * letters and special codes in all lower case.
 * Here the codes are defined directly from their
 * ISO notation.
 */
A_GRAVE = 12/0    : UPPER, PRINT;
A_AIGU = 12/1     : UPPER, PRINT;
A_CIRCON = 12/2   : UPPER, PRINT;
A_TILDE = 12/3    : UPPER, PRINT;
DIA_A = 12/4: UPPER, PRINT;
A_CIRCLE = 12/5   : UPPER, PRINT;
/*
 * The below declaration of AE as a diphtong enables
 * the correct treatment of diphtongs (one to two
 * collation!) in the default collation.
 */
AE = 12/6    : UPPER, DIPHTONG, PRINT;

/*
 * lower case equivalents of the codes defined
 * in the last block
 */
a_grave = 14/0       : LOWER, PRINT;
a_aigu = 14/1 : LOWER, PRINT;
a_circon = 14/2      : LOWER, PRINT;
a_tilde = 14/3: LOWER, PRINT;
dia_a = 14/4  : LOWER, PRINT;
a_circle = 14/5      : LOWER, PRINT;
ae = 14/6     : LOWER, DIPHTONG, PRINT;

/*
 * special double letters for Spanish
 * Note that these "characters" are not defined by
 * any standard! They represent an extension
 * useful to handle the following problems:
 *       - two to one collation
 *       - conversions toupper and tolower
 */
Ll = L, l : DOUBLE, UPPER, PRINT;
ll = l, l : DOUBLE, LOWER, PRINT;

END.
```

```
/*
 * Collation table that shows most of the possible
 * problems in collation but does not make very much
 * sense in the real world:
 *
 * Upper and lower case letters are intermixed and
 * within one letter the upper case comes before the
 * lower case letter.
 *
 * Accented characters sort after their corresponding
 * non accented base character.
 *
 */
COLLATION :
        PRIMARY : A, A_GRAVE, A_AIGU, A_CIRCON, A_TILDE,
                DIA_A, A_CIRCLE;
        PRIMARY : a, a_grave, a_aigu, a_circon, a_tilde,
                dia_a, a_circle;
        PRIMARY: B; PRIMARY: b; PRIMARY: C; PRIMARY: c;
        PRIMARY: D; PRIMARY: d; PRIMARY: E; PRIMARY: e;
        PRIMARY: F; PRIMARY: f; PRIMARY: G; PRIMARY: g;
        PRIMARY: H; PRIMARY: h; PRIMARY: I; PRIMARY: i;
        PRIMARY: J; PRIMARY: j; PRIMARY: K; PRIMARY: k;
        PRIMARY: L; PRIMARY: l;

        /*
         * TWO TO ONE COLLATION:
         *
         * For Ll and ll Spanish collation rule says that
         * this has to be collated after L or l.
         */
        PRIMARY: Ll; PRIMARY: ll;

        PRIMARY: M; PRIMARY: m; PRIMARY: N; PRIMARY: n;

        /*
         * ONE TO TWO COLLATION:
         *
         * The following two codes are diphtongs, that is
         * codes that collate as two characters.
         */
        AE = (A, E);                    ae = (a, e);

        /*
         * The rest of the codes defined in the codeset will
         * collate as if they were non existent.
         */

END.

STRINGTABLE :
END.

/*
 * The next two sections are the required code conversions
 * tolower and toupper. For both conversions the manual for
 * ctype(3) says that a code with no lower/upper case
```

```
 * equivalent will be returned unchanged.
 */
CONVERSION tolower :
        DEFAULT -> SAME;
        A - Z -> a - z;
        A_GRAVE - AE -> a_grave - ae;
        Ll -> ll;
END.

CONVERSION toupper :
        DEFAULT -> SAME;
        a - z -> A - Z;
        a_grave - ae -> A_GRAVE - AE;
        ll -> Ll;
END.

STRING CONVERSION to ISO6937 :
END.
```

3. Message Presentation

The basic requirement is that the mechanism should not modify the way a programmer writes programs, nor their associated makefiles. It should limit the amount of extra work necessary for a programmer to make a multi-lingual program.

3.1 Basic principles

The mechanism of message presentation is integrated with the development utilities: *cc(1)*, *as(1)*, and *ld(1)*. It is made up of:

— an evolution of some of their constituent parts

— or a set of pre/post processors inserted into the development chain.

The mechanism of internationalization is invoked by an option to cc. The programmer does not need to manipulate an intermediary work file.

- All the messages in the program in the same language are grouped in the same section as defined in an extended COFF format. The message section is included in the executable (a.out) file. There are STYP_NL types (a flag in the header of the a.out(4) section)
- An extension to the loader *exec*(2) only reads into memory the message section associated with the user's declared language (environment variable **LANG**).
- There is a translation tool that helps the programmer (or a professional translator) associate the program's messages with messages in other languages, to facilitate the translation into multiple languages, without modifying the program source.

There are two distinct ways of integrating messages into the programs :

- dynamic selection of the appropriate messages: The executable image contains messages in many languages; when the program is called, only the messages in that user's language are loaded into memory.
- static selection of messages: A software house might want to have an in-house copy of each of its applications in every language, but wish to sell copies with messages in only one language. A tool allows this sort of manipulation by transforming the binary into a standard (non-extended) binary format.

The following criteria must also be taken into consideration.

- the message presentation mechanism must work properly for applications that are developed using separate compilation.
- library functions (libc, libm, user libraries ...) must also provide multi-lingual messages so that the same library can be used in many different language environments. Thus :
  — the .o files must be translated and be able to contain several different message sections, one for each language.

  — the link editor (linker) must be able to link these multi-lingual .o files, correctly combining the proper language sections to each other.
- the recognition of a particular language by the linker is done using an eight-letter (maximum) string.
- the message presentation mechanism collects both printable and non-printable strings (which should NOT be translated) in the source. We plan on having such strings marked by the programmer (in a .o or library) so that they will thus not

be translated. This could be proposed as an extension to the C language.

- this implies changes in the tools that manipulate the .o files, particularly : *sdb(1), dump(1), strip(1), nm(1),...*

3.2 Our Solution

The objective is to isolate the strings and put them in a special section of the COFF file called the "message section". The translation will then be made easier; all the messages will thus be in the same a.out, with a separate section for each language. In order to reduce the modifications necessary to the existing program development cycle (cpp, C, as, ld), these tools are complemented by independent pre/post processors:

— a compilation pre-processor : nl_cpp

— an assembler pre-processor : nl_as

These pre-processors are called optionally from the cc command. Thus one can still use the "old" set of development tools. The relationship of the different parts of the international development tools is:

cpp → nl_cpp → ccom → nl_as → as → ld

3.2.1 The nl_cpp pre-processor

In a C program, we want to find all the strings in order to easily print out their translation in any given language during execution. We must be able to change the references to a string without having to change the .text section, only the .data section.

The nl_cpp pre-processor modifies the C program source in order to isolate all the strings in a module and to generate indirection when it does not already exist. The pre-processor **nl_as** modifies the assembler source in order to generate a structure in the message section. Il transformes all the "section 15" instructions into "section .nl", working from the structure declaration.

The assembler was modified to take into account the message section ".nl".

The .o files made by the assembler will now have four sections: Some of these modules could be translated before the link phase (as would be the case for libraries). This translation would create other message sections in the file, one section for each added language.

3.3 a.out format

1. ld normally knows how to treat a certain number of sections. By default, it places the .test and .data sections at pre-determined addresses (the bss section is added to the end of the .data section) and inserts the other sections in unused addresses in the virtual memory space. The loading mechanism and the execution of an a.out demand that the .data and .bss sections be placed at the end of the virtual address space. The message sections will thus be put between the .text and .data sections. So as to not waste space in the virtual address space, and because only one of the (possibly) many different message sections is loaded by exec, the message sections in a.out all have the same virtual address.

2. The .o from an application can be translated (i.e. several versions of every string, each in a different language) before the link phase. It can thus contain several message sections, and there are three possibilities for the a.out after the link

editing phase (see figure one):

a. ld only keeps the message sections for one (specified) language. the linking is refused if any module does not contain that language's module. (→ rather restrictive).

b. ld keeps the message sections found in the .o files. (→ the a.out will not be executable in every language)

c. ld keeps only the message sections that are defined in all of the modules. (→ some translations will be lost)

For the moment, only solution **a**) has been implemented.

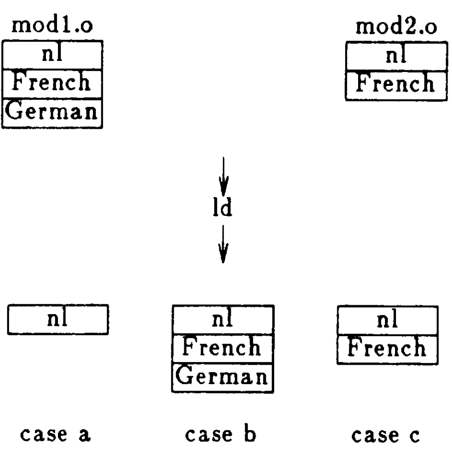


Figure 1

3. In order to satisfy the static message presentation, messages are put at the end of the data section.

3.4 ld modifications

1. Putting message sections between the .text and .data sections can be implemented by giving directives to ld (modification of those given by default). To be able to put all the message sections at the same virtual address, we create a new type of section: the message section.

2. A procedure independent of the link phase is added to ld so that it will only keep complete message sections. (this procedure could constitute a post-processor to ld) The translation tool should, during the creation of a message section in a .o file, add relocation information for this new section (this information is made from those available in the native language). Since the message sections all have the same virtual address, we must modify the relocation calculation.

3. Putting messages at the end of the data section means that we must translate references to the bss section. ld, using a specific option, keeps the relocation table concerning the bss section, so that it will be available when needed.

Normally, sdb only knows how to handle .text and .data and .bss sections. The options -g and INTL of the *cc(1)* command are incompatible. On the other hand, the tools *dump(1), size(1),* and *nm(1)* can handle these new sections without problems.

3.5 Loading an executable file

The solution consists of:

- *exec(2)* only loads into memory the message section corresponding to the LANG environment variable.
  - — the kernel does not usually know anything about the user environment; we have added a field (u_lang) to the user structure inherited from the parent. During loading, if *exec* finds message type sections in the executable file, il uses the u_lang field to decide which message section to load.
  - — char u_lang[8] can be changed by a system call (*setlang*(2) and *getlang*(2)).
  - — the kernel might not know the process causing an error; the error message should be printed on the system console in the system's language (the system administrator only knows one language)
  - — *exec(2)* recognizes each section by the name written at the beginning of each section. char s_name[8];
- memory location of the message section between the .text and .data+bss segments
  - — the sharing of messages between different processes is a possibility.
  - — the different message sections should all have the same virtual address.
  - — memory constraints for message translation : the translation process should not take up more than 256K over the initial message section. (256 K seems a reasonable size for a message section)
- address initialisation for strings contained in the data segment is done by the runtime routine crt0 (updated according to the information contained in the structures in the message section).

3.6 The translation utility

A translation aide works both on .o files and a.out files. The translator is a message validation tool (an a.out file can not be executed if it has not been validated).

Services provided by the translation tool(s) are:

- marking of strings that the programmer does not want translated. This makes the strings invisible to the translator program.
- destruction of the message section associated with a given language.
- insertion of a new message section (for a given language) in the compiled binary file.
- listing the languages of messages in the binary. (similar to dump -h)
- extraction of the message section associated with a given language.
- a screen-based, highly interactive translation tool to help translate catalogues of messages easily and speedily.
- compilation of a message section and its insertion in the binary file.
- transformation of an internationaled a.out into an "normal" a.out. This "normal" a.out is made from the .bss relocation table left by the linker. The transformation tools installs the messages at the end of the data section and resolves the references to the bss. This tool does a lot of the same things as *ld*(1), and is fairly easy to make.

4. References

[XOPEN87] The X/OPEN Portability Guide, Issue 2, January 87.
Elsevier Science Publisher bv, Amsterdam.

[SCH84] Introduction to Compiler Construction, A UNIX tutorial
Axel T. Schreiner, Sektion Informatik, University of Ulm (West Germany) and H.George Friedman Jr., Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana IL 61801, 1984.

[GELD86] Ic : An International Compiler
Wolf G Geldmacher, Communicon AG, Rorscharch, 1986, Unpublished.

UNIX in Manufacturing

By Dr G Kruse, Root Business Systems Limited, London

Computers in Manufacturing

Manufacturing companies have, after a relatively slow start, over the last few years increasingly automated their business environment, using computer assistance wherever applicable.

The work done has predominantly been based on three separate development strands:

i. CNC - Computerised Numerical Control

   This involves the control of machine tools by numerical means, moving machine elements such as work tables and tools in synchronised conjunction to generate a desired component geometry. The work on numerical control goes back to the early 1960's and has increasingly moved away from largely analogue to totally digital controls, making the use of standard computers as control units more common.

ii. CAD/CAE - Computer-aided Design/Computer-aided Engineering

   Computers are today widely used in CAD/CAE applications to develop and design products. Originally based on single-user machines, multi-user workstations are now common, with many systems using UNIX. The need for independent workstations was promoted by:

   - the need for "unusual" peripherals such as graphics terminals, digitising tables and plotters, and

   - the high processor usage of certain scientific processes which would at times load the processor to a degree where it could not be used by other applications within constrainst of reasonable transaction response times.

   Other stand-alone solutions were derived from the above to deal with specific manufacturing tasks such as N.C. tape preparation and computer-aided process planning. The whole area is generally referred to today as CADCAM (Computer-aided Design, Computer-aided

Manufacture).

iii. MIS - Management Information Systems

These systems represent the traditional computing functions in a business, often mainframe-based and even in a mini-computer environment operating in a main-frame mode of individual systems residing on one machine and competing for a common resource.

Integration between CNC, CAD/CAE and MIS sub-systems generally did not exist, and even today integration of computer systems is still a relative rarity.

Until recently, when communications between different computers became easier, two extreme alternatives tended to exist.

i. stand-alone "islands of automation", ie specific localised solutions to problems, but with no data availability to users outside of each area of automation.

ii. the quest for a large monolithic "corporate database" which due to the size of the project and the changing needs of a business rarely achieved a significant level of implementation.

The current view on the resolution of this dilemma is CIM "computer-integrated manufacturing".

CIM

CIM is today understood in its broadest sense as the total integrated computerisation of the technical, administrative and management activities of a manufacturing business, and as such covers all business activities from design and development through to process control, manufacturing resource planning, business planning and financial control.

The ultimate strategic objective is the "automated factory".

Clearly any business system breaks down into many sub-systems, each with its own functional needs, its own boundaries and connectives to other sub-systems.

CIM implies a set of task-specific computer applications, each automating a process or set of processes and each linked through datalinks to one or more other computer applications. Each application may have (normally will have) one or more human interfaces.

Yesterday's islands of automation failed to bring real benefits since:

- suboptimisation at local level was accepted as a goal in its own right

- data had to be entered independently into each sub-system and users had no data access across sub-systems

- management had no ready access to the bulk of company data and could not use it for company-wide decision making

- local changes of plans and local problems were not automatically reflected in the plan execution of other sub-systems

Only integration of sub-systems could bring real business systems benefits from computerisation; thus it is INTEGRATION that is so important in CIM.

The other difficulty was the proliferation of different manufacturers' equipment and software systems in a highly automated system. Different suppliers had (and still have of course) different communications protocols, different operating systems, different high level language computers. True integration was only a reasonable aim within one supplier's range of equipment and solution (and in most cases not even then). This appeared to protect a supplier's user base and was therefore popular with hardware manufacturers, but in practice only benefitted one or two major international suppliers.

International Standards

An increasing effort has taken place over the last few years to establish international standards for computer systems to enable companies to build truly integrated business systems.

Manufacturing companies have probably the greatest need for diverse systems within one company and in manufacturing companies the need for standards is probably most pressing.

The essentials are

- communications standards

- software portability

In both areas great strides have been made; in communications the extensive use of Ethernet-type links and

the General Motors initiative in the promotion of MAP (Manufacturing Automation Protocol) make data communication a relatively routine activity in modern multi-machine computer systems.

General Motors have also done much to further the cause of UNIX in manufacturing by establishing internal policies for the selection of all future computerised manufacturing systems to be based on UNIX. The lead has been followed by many other government and commercial ogranisations.

Software portability has been a more difficult problem and individual efforts such as the initiative by the X-OPEN Group, provided the real breakthrough. UNIX is clearly now emerging as the only universal platform for software portability. SVID and POSIX are moves to formalise that portability and provide a "guarantee" of mobility across the range of UNIX-based operating systems.

Business Applications under UNIX

Business systems can be broadly split into two area:

- Management Information Systems
- Engineering Systems

In engineering systems UNIX is very well established. Some of the early weaknesses of the UNIX environment are not relevant and have not been a barrier to development.

In manufacturing information systems, the situation has been more critical. Record locking and full dynamic recovery from failure are essential in a multi-user transaction processing environment. There is no doubt that UNIX is now emerging as a business systems environment. Three development strands can be observed:

i. upgrading of P.C.-based systems, possibly re-written in 'C', usually cheap, but often with poor security and data protection.

ii. new systems written within the development environment of a 4GL, using the security and resilience provided by that environment, but possibly with a penalty on development flexibility and processing efficiency.

iii. systems from other environments integrated to UNIX, emulating the transaction processing environment of its source and providing within the UNIX implementation the resilience and security normally

available in say an IBM/CICS or DEC/VAX environment.

The latter approach involves initially a high-cost development project since effectively the writing of a UNIX-based operating environment is required which surrounds the application code such that application code and T.P. harness are jointly and totally portable within SVID/X-OPEN standards. It is, however, technically the most rewarding approach, since it provides in a UNIX environment standards of transaction processing efficiency, data security and resilience which are taken for granted in other, transaction-orientated data processing environments. The lack of such facilities in the past was doubtless a key reason for the slow rate of acceptance of UNIX in the business systems world.

Example of a High-Functionality Manufacturing Control System

In 1984 Root Computers made a strategic decision to extend their activities into high quality application software. Such a move was seen as essential for the promotion of UNIX itself as a serious business system environment.

The ROOT objective was to provide high quality software to cover integrated business systems from finance to operations with a strong emphasis on manufacturing. Such software was intended to be of a sufficiently high level of functionality, flexibility and quality to rival existing mainframe application products and to offer

a. UNIX-based solutions with a high appeal to mainframe users

b. distributed systems with a level of functionality which would be acceptable to companies who had experienced the use of sophisticated application packages in the past.

There was in Root's view not one product in the UNIX market place which could reasonably satisfy these requirements. Writing such a system was unacceptable in terms of cost and development timescales, and a commercial collaboration with a current supplier of a state-of-the-art product was deemed to be most appropriate.

Extensive market analysis led to discussions with Hoskyns Group PLC, a subsidiary of U.S. aerospace and defence giant, Martin Marietta Corporation.

As a result of these discussions, ROOT took on the Hoskyns MAS Modular Application Systems and ported them to UNIX to

create a range of products jointly owned by Hoskyns and Root, consisting of:

Manufacturing Control System
Financial Control System

The Hoskyns contribution was the COBOL-based application programs and Root supplied a transaction processing harness and security environment emulating the functionality of CICS within the simpler, more user-friendly operating world of UNIX and written in 'C'.

The result was a product, offering a well-proven, advanced mainframe functionality of established pedigree running efficiently and without any reduction in facilities under UNIX V. For the first time did the UNIX world have an attractive application software option for larger companies, thus opening up for UNIX the large market of sophisticated and complex operations and justifying the claim that UNIX can be a true competitor with existing mainframe systems, enabling large companies to consider seriously a migration to UNIX, not only for its technical systems, but also for its business applications. At the other end of the spectrum, the cost/performance effectivity of the UNIX based MAS system is offering mainframe systems at a price attractive to much smaller users.

A significant by-product of migrating the Hoskyns MAS range of products to UNIX and setting up within UNIX a transaction processing system which emulates the mainframe environment, was the development of a range of programming tools to automatically migrate CICS based applications to UNIX. Where during the early work significant manual effort was involved, the tools have now been refined to such a degree that under 10% of conversion time deals with manual adjustments to the converted source code. The significance of such work will be obvious to all members of the UNIX "fraternity": The tremendous barrier to a migration to UNIX, ie the enormous investment in existing applications, especially in the IBM/CICS world, can now be overcome by means of relatively simple, automatic conversion tools.

This has been a major breakthrough in the future growth potential of UNIX and is so significant that Root have decided to make these tools available to other companies as a packaged software product to enable the UNIX world to take advantage of this opportunity for real UNIX penetration into the traditional mainframe business.

Distributed Business Systems

A manufacturing company or manufacturing "system" is traditionally visualised as a series of sub-systems.

Complex systems problems are generally resolved using the "Systems Engineering" approach. The concept has it birth in advanced engineering environments such as the aerospace industry, where technological problems became so complex, that problem solution was very difficult or even impossible by conventional means.

Systems Engineering views a system as a conglomerate of separate, but interrelated elements. By defining each element and its interface relationship to other systems elements, it is possible to resolve each element in turn, whilst still accounting for its effect on the overall systems behaviour.

A reasonably sized manufacturing business is, of course, a most complex system, requiring the control and co-ordination of a vast number of activities in such a way that the composite objective performance of all elements yields an overall systems optimisation.

Progress to date has largely concentrated on the fundamental aspects of individual sub-systems in isolation and has failed to clearly identify, firstly, the interfaces between functional areas and, secondly, constraints imposed by other sub-systems which must be observed to ensure that sub-systems optimisation does not lead to overall systems sub-optimisation.

Sub-systems in a manufacturing company deal generally with departmental activities such as

- design
- production planning
- inventory control
- process planning
- financial ledgers
- costing
- sales order processing

The concept of the "departmental computer" is gaining wide acceptance. Each department can have its own computer, either real in the form of a separate small machine, or as a "logical" machine within a large mainframe.

In the past "logical" machines were required for financial reasons and economies of scale favoured mainframes. Today independent departmental machines are very cost-effective and score highly in a number of ways, ie

- initial cost of acquisition
- motivational (department "owns" its system and data)
- feasibility
- expandability
- ease of use
- ease of maintenance
- security
- cost of software

A typical departmental system consists of

- a computer
- a database
- a set of maintenance programs
- a set of terminals and other peripherals

The communications needs tend to be simple

i. data from one departmental data base must be accessable from other departmental system, reading data only.

ii. a departmental database can be updated from another department, but only by passing transactions through the programs maintaining that database. Such a "constraint" is invariably desired by users to ensure that any external maintenance of their own database is validated properly by the normal update programs and audited on the audit file of the department whose database is updated.

Thus, the architecture and the communications needs tend to be simple.

A departmental system, which can of course consist of multiple workstations and a file-server will tend to have one single departmental database. This is also in line with most commercial offerings.

Databases can be accessed from any other system via simple communications links such as Ethernet for example.

Database update from external systems is via transaction passing, ie one program outputs a suitably formatted transaction, which is fed to another program which may well be resident on another machine. Again Ethernet is perfectly adequate as the link.

Within UNIX such a process is of course particularly easy. Transactions can be generated by one application program and piped to another program on another machine.

On mainframe applications such a process can be very difficult, requiring often special programs to create such a link, or requiring traditional batch processing to transfer data.

In a UNIX environment the building of database systems is truely simple, particularly in the case of manufacturing systems, where sub-systems are clearly defined.

Root have done some useful work in this area to promote UNIX as a basis for distributed manufacturing systems design. In one instance, Hoskyns Group, Root and a leading UK machine tool builder designed a system to integrate a supervisory production control systems (MAS) with an automated flexible manufacturing systems (FMS). The objective was for the MAS system to provide the FMS cell with a view of available work and let the supervisor "release" such work for manufacture. Work progress was automatically fed back to the MAS system to indicate the work status at all times. Hoskyns chose the UNIX version of the MAS system and Root implemented the system by accepting formatted transactions from the FMS cell through an Ethernet link. In a mainframe application, several man months of work would have been involved. In the UNIX environment the work was trivial.

In a similar exercise, an automatic weighing machine with its own processor was linked to the same MAS system with minimal effort.

Integration at that level is particularly easy in a UNIX environment and opens up a new world in distributed business systems.

Root now confidently offer the range of MAS application products on distributed departmental UNIX machines, requiring merely a standard Ethernet-type of communications link. True integration is assured within a totally flexible implementation. Such a flexible approach, based around UNIX as a common standard has provided a new level of freedom in strategic systems planning and could not easily be achieved in a non-UNIX environment.

Implications for Manufacturing

Returning to the general theme of this paper, it is now fairly clear that UNIX will find rapid expansion within the manufacturing industry.

Technical systems are already to a significant degree UNIX-based. With current standardisation efforts, and true software portability, UNIX will no doubt more rapidly become the accepted standard for business applications.

Business Systems have lagged behind in the past, but there is growing pressure from users to supply UNIX solutions so that UNIX can become truely a company-wide standard.

Root are finding increasingly that major corporations are attempting to build a business systems strategy around one operating environment with open systems capability, and are looking to UNIX as the only currently available vehicle to put that strategy into place.

The lack of business application software has doubtlessly been a serious shortcoming in the past, but after the recent Root initiative, it is likely that a number of other major software vendors will migrate existing well-established quality products to UNIX.

There is no doubt that UNIX can provide very cost effective solutions. Large, traditional suppliers of computers have provided low risk, high quality products and charged a premium; a policy epitomised by the old cliche that "nobody gets fired for buying IBM". Full portability of software reduces that risk dramatically and provides a high level of insurance against

- supply problems

- growth requirements

- technical obsolescence

Portabilitiy allows the user a high degree of freedom over his selection of systems components to progressively build up the optimal business systems for his needs.

Now and in the future UNIX can and must become a true corporate computing standard for manufacturing companies striving towards Computer Integrated Manufacture.

What I want as a manufacturing man is

- a flexible, unconstrained strategy

- full software portability

- a common technical environment

Only UNIX can give me all that to enable me to confidently develop my CIM strategy.

UNIX & Entertainment
Why & How?

Peter S. Langston

Bell Communications Research
Morristown, New Jersey

ABSTRACT

This paper seeks to answer three questions. The first, *"why entertainment?"*, might be asked by a venture capitalist looking for a profitable investment area or by a software engineer looking for an interesting application area. The second question, *"why UNIX?"*, is meant to uncover the advantages inherent in using the UNIX Operating System (and the philosophies expressed in it) in an entertainment project. The third question, *"how unix & entertainment?"*, serves as a jumping off point for describing ways in which the UNIX Operating System has already been used in the entertainment industry.

INTRODUCTION

The UNIX Operating System has found its way into a multitude of application areas ranging from software engineering to accounting to artificial intelligence. Orthogonal to these application disciplines are the industries that use computing in one way or another. Some industries have, for many years, profited from one or two predictable and fairly narrowly defined computing disciplines (e.g. banking & finance). Other industries, such as entertainment, have much less narrowly defined needs and are just beginning to profit from the use of computers and computer science in a broad range of disciplines. The juxtaposition of these observations, while hinting at a connection between UNIX[0] and industries like entertainment, may also raise questions like *"what does UNIX have to offer the entertainment industry?"*, *"why isn't UNIX being used in entertainment now?"*, and *"who cares about entertainment, anyway?"* The following sections develop answers to (paraphrases of) these questions.

WHY ENTERTAINMENT?

Why anything? To put it another way: what criteria should we use to choose one industry over another? Several possibilities spring to mind: *money* (i.e. financial profit), *challenge* (i.e. intellectual profit), and something which we shall call "*quality of life*" (i.e. societal or spiritual profit). An industry like banking meets our first criterion quite well, but requires a fair amount of creative rationalization to score well on the other two criteria. An industry like education scores well on the last two criteria, but fails miserably on the first. How does entertainment score on these three criteria?

[0] "UNIX" is used here to mean "the UNIX Operating System and the philosophies expressed in it". Try to think of it as a shorthand rather than a noun.

Money

In a single year, 1981, video game companies took in over one billion dollars (4,000,000,000 quarters!) in revenues with only minor operating expenses. That qualifies well as a financially profitable industry. While video games companies have experienced a meteoric fall matched only by their meteoric rise, other segments of the entertainment industry have shown consistently high profits for many years. Steven Spielberg and George Lucas have not only made themselves millionaires, but have made quite a few other people rich with movies that make millions of dollars of profit[1]. The Consumer Electronics Show, (a trade show that deals primarily with entertainment hardware), represents hundreds of vendors and draws close to one hundred thousand participants twice a year. Entertainment is a large and profitable industry. With the advent of new technologies (e.g. we can expect to see inexpensive, two-way, broad-band communication in the home soon) the market for entertainment products will grow further.

Challenge

From the point of view of a computer scientist, the entertainment industry is an interesting testing ground. Entertainment products require solutions to problems in many diverse subfields — algorithmic efficiency, human interface design, simulation, and real-time computing, among others. Approaches that are barely adequate in entertainment applications are often orders of magnitude better than those used in other, "real world" applications. Video games, as an example, require solutions to problems in:

- real-time computing — Real-time response must be provided, often on hardware that is only marginally more sophisticated than a Turing machine.
- human interface design — The human interface, even in a bad video game, must be orders of magnitude better than that provided by most data-base management systems.
- human perception — The difference between a believable scenario and an unconvincing one often hinges on an understanding of human perceptual mechanisms; for instance, when a sound has to be louder than the equipment can produce, making it more distorted will give the needed effect.
- "real" world physics — Often the only convincing way to simulate two Atomic Frannistans colliding in outer space is to give them familiar physical characteristics (mass, elasticity, spin, etc.) and treat them as familiar physical objects (e.g. billiard balls). The accurate simulation of the familiar objects lends credibility to the entire fantasy.
- code optimization — As mentioned earlier, the target machine for a video game is often tiny and slow. Well-designed video games are often startlingly good examples of optimization in both space and speed.
- data compression — Video games often depend heavily on data-intensive features - graphics, dictionaries, etc. An arbitrary image to be displayed in color on a 512x480 screen could take a quarter of a million bytes. Extreme data compression is needed to store a dozen such images in the 32,000 or so bytes available in even the largest video games.

---

[1] The ticket sales on George Lucas's films alone amount to more than one billion dollars. Even "Howard the Duck", which is viewed as a financial failure, may end up showing a few percent (of its $32,000,000 production cost) profit.

Quality of life

Quite separate from the questions of monetary profit derived from entertainment industries and the intellectual challenge of an area that is truly sensitive to improvements in the state of the art, is the question of improving the quality of life for the consumer. This may take the form of education, amusement, reduction of stress, strengthening of family ties, or simply appreciation of another's craftsmanship or creative vision.

Entertainment provides an opportunity for all these activities. The dictionary associates entertainment with amusement and pleasure[2], but if we look at individual instances of entertainment, we find further associations. Music is one form of entertainment that can be described as an art whose goal is aesthetic[3]. Games can not only be explicitly educational (spelling games, arithmetic games, etc.) but they also function as one of the mechanisms for teaching societal goals and mores. Further, playing games can provide relaxation and escape from tensions. Finally, almost all forms of entertainment can be used as a social framework in which friendships and family ties can be explored and strengthened.

In a society obsessed with "progress", anything as intangible as the "quality of life" may be viewed with some skepticism. This lofty criterion is certainly the hardest to measure of the three — one can't count dollars earned or papers published to determine the success of an effort — nonetheless, only the most cynical would deny its importance to society.

WHY UNIX?

Although there are many possible reasons for preferring the UNIX Operating System for tasks in the entertainment industry, I will focus on a few that seem particularly important.

Software Development Tools

The UNIX Operating System makes software technology more accessible by reducing the start-up costs for a project. Often a prototype for a system under consideration can be assembled from existing software "tools" in a matter of hours or minutes. Dead ends can be abandoned early in the design cycle and more fruitful approaches found before any "real code" has been written. UNIX systems are not the only ones that provide software development tools; other systems can, and some even do;

Software Development Atmosphere

Entertainment requires innovation. The production of a new movie, for instance, requires a new script, new plot devices, new insights into people's interactions, or whatever. (There are a few series of movies that seem to pride themselves on their penchant for telling a single story over and over in different ways; but at least they tell it in different ways each time.) The more presumptions a tool makes about its use, the more narrowly it defines what can be done with it. UNIX operating systems consciously attempt to stay out of the user's way and make no fixed assumptions about what he/she wants to do, and in so doing establish an environment in which software can be used in ways that the creator could not have foreseen. This potent "open" arrangement has its dangers. Such

[2] "en•ter•tain•ment *n.* 1. The act of entertaining. 2. Something that entertains, esp. a show designed to amuse. 3. The pleasure afforded by being entertained; amusement." American Heritage Dictionary

[3] "mu•sic *n.* 1. The art of organizing sound so as to elicit an aesthetic response in a listener." American Heritage Dictionary

a system is exceedingly vulnerable to user stupidity or malice; by a simple typographic error, such as inserting an extra space into an otherwise innocuous command[4], hundreds of vital files can be destroyed. It's not surprising that operating systems designed for novice or casual users can't afford such openness and consequently make innovation more difficult whereas more open systems tend not to appear outside the computer science research community.

Portability

The advantages of portability are manifold and have been discussed (to death) in technical journals and the popular press. Most of the advantages don't need to be elaborated here; but a few are particularly pertinent to the entertainment industry.

The technique of developing a piece of software on one system for use on another can pay big rewards when the target hardware is small or slow (or both as in the case of home video games). An elaborate development system can provide hundreds of tools for monitoring, debugging, quick prototyping, and documentation, while the system on which the result will actually be used (perhaps owned by the consumer) need only support the display of the result. It is instructive to contrast the simplicity of the home video cassette player with the immense production machinery necessary to produce a movie. In the case of movies, a conversion step is required (the "film chain" which converts 24 frame-a-second film into 30 frame-a-second video). With UNIX operating systems now available on microcomputers, minicomputers, and mainframe computers, very little, if any, conversion is needed to develop programs on big machines to run on little ones.

The ability to use the same or similar development programs on a variety of different computers without having to make special plans ahead of time to do so can be a godsend. When Lucasfilm Ltd.'s graphics group decided to produce a completely computer-generated cartoon to show at Siggraph '85, they used their in-house computing facilities (a D.E.C. Vax 11/780 running BSD 4.2 UNIX) for storyboarding, animation, image rendering, and previewing. Since this project was breaking ground in a number of areas it was difficult to be sure of the film's timetable. As deadlines approached, it became obvious that the 2500 (or so) frames would not be done in time. Fortunately, the graphics software was written to run on a UNIX operating system, using standard compilers and utilities. Not only did this mean that parts of the project could be run on other such systems (about 30 Vax UNIX systems ended up helping out) but, since the new Cray supercomputer system also used a UNIX operating system, it meant that parts of the image rendering could be run on that very fast system also.

Portability will become less and less an advantage for UNIX systems as more and more operating systems and software products are designed with portability in mind. Meanwhile, it's amazing how much "new" software can only run on one specific kind of hardware or can only read files written in one specific format or by programs running under one specific operating system. As an example, a myriad of programs are available to support music synthesizers using MIDI communications.[5] Some programs allow you to compose music, others are used to edit it, others generate or modify sound synthesis parameters, while still others provide database management functions for storing and

[4] E.g. "rm junk*", you can probably guess where the extra space goes.

[5] MIDI is an acronym for "Musical Instrument Digital Interface", and is a hardware and data format standard established by the MIDI Manufacturers Association, a group of synthesizer manufacturers and others. [MIDI85] MIDI allows synthesizers, sequence recorders, home computers, "real" computers, rhythm machines, etc. to communicate through a standard interface, and has spawned an entire subindustry.

retrieving voice parameter data or note sequences. Typically each program provides one or two of the dozen or so operations needed to do any serious work with MIDI equipment. Unfortunately, each program runs on a single specific home computer; to put together a reasonable set of software to support the music demonstration described later in this paper would require an Apple Macintosh, a Commodore C64, an IBM PC (or comparable clone), and an Atari ST. Even then, the file formats for the various programs are different, so exchange of data could only be carried out through MIDI wiring between computers; disks written by each system would need to be kept separate, etc. The only thing portable in a set of software to support the MIDI standard is the MIDI data itself.

HOW UNIX & ENTERTAINMENT?

In the following paragraphs I will describe some ways in which the UNIX Operating system has contributed to efforts in three subfields of the entertainment industry – games, movies, and music. There are many more examples than those given, but these should provide an idea of the range of interactions between "UNIX" and "entertainment".

Games

The first example is Ken Thompson's chess playing machine, "Belle". Ken's parental involvement with UNIX is, hopefully, well known, as is his chess playing machine which has several times been the world's champion, beating much larger programs running on much larger machines. It should come as no surprise then, that the software for the chess playing machine was developed under the UNIX Operating system. It also stands to reason that the principal architect of the UNIX Operating system would make sure that the system provided as much support as possible for the development of games software. As it happens, the first version of the UNIX operating system was written by Ken so he could have a system on which to write a space travel game[6].

The direct predecessor to the global simulation computer game, "Empire"[7] was written on a time-sharing system that provided only a Basic interpreter; there were no software tools to aid in development and the use of a non-structured language made reading the code a Herculean task. As a result, that game was almost impossible to modify or maintain and, despite the thousands of hours of work lavished on it, never gained widespread distribution. The game flowered when it was rewritten in C on a system running the UNIX operating system. It grew tenfold in depth and subtlety while becoming easier to maintain (with maintenance being provided, albeit sporadically, by one person rather than half a dozen). The current version of "Empire" has run on hundreds of systems, has been used in a government course at Harvard University, has been studied by the institute that awards the Einstein Peace Prize, and was used by its creator to get a job as a data base management expert.

The computer game "Oracle" grew out of a simple idea about human interaction. Computer-naive people often ask very subtle or complex questions of computers, not realizing the complexity involved. "Ask the computer why the sky is blue", "can it tell us who's going to win the election?", or even "computer, do you know what time it is?" have been the downfall of many demonstrations. A human would have no trouble hedging on the first question, explaining why the second is unanswerable, or understanding the implicit question in the third, but all three of these are very

[6] "its true about games. most of the stories are well known. the pdp-7 days were dominated by games." [THOMPSON85]

[7] Not to be confused with the space war game "Empire" on Plato.

sophisticated responses. The idea in Oracle is best described by an example:
Oracle: *Hi, I'm the Oracle, ... blah, blah, ... What's your question?*
User Joe: *Why is the sky blue?*
Oracle: *Gee, that's a toughie! I'll think it over and mail you an answer. Bye for now.*

Oracle: *Hi, I'm the Oracle, ... blah, blah, ... What's your question?*
User Amy: *Who's going to win the election?*
Oracle: *Hmmmm ... tricky! I'll have to think it over and mail you an answer.*
Oracle: *Meanwhile, perhaps you could answer this question for me:*
Oracle: *Why is the sky blue?*
User Amy: *The sky is blue because of refraction through water droplets.*
Oracle: *That's right! Nice going. Bye for now.*

The Oracle has calculated the answer by using a human peripheral processor; it now mails the answer to Joe and lies in wait for someone to tell it how the election will come out. This silly idea was very easy to implement under a UNIX operating system because all the necessary communications tools were already present and made no particular assumptions about how they were going to be used (e.g. the mail program was perfectly happy to accept mail from another program).

The paper "The Influence of UNIX on the Development of Two Video Games" [LANGSTON85a] describes, in some detail, how the UNIX operating system and tools made possible the development of the video games "Ballblazer" and "Rescue on Fractalus". That paper concludes:

> We were heavily influenced by the software, philosophies, and concepts associated with the UNIX operating system. The UNIX environment was perfectly suited to our task. We didn't have to fight with programs that *almost* did what we needed but had made some limiting assumption. Nor did we have to trick the operating system into letting us do something that it thought we shouldn't do.

In short, these innovative games profited greatly from the tools and philosophies manifested by the UNIX operating system.[8]

Movies

Lucasfilm Ltd., the company responsible for the "Star Wars" movies, the "Indiana Jones" movies and others, has been a leader in finding high-tech solutions to film industry problems. The Lucasfilm Graphics Group (now a company known as "Pixar") introduced the UNIX operating system to the company and, using it, has produced film graphics never before feasible.

The Genesis Demo from the movie "Star Trek II – The Wrath of Khan" was completely computer generated and pioneered the use of several new image rendering techniques. The images produced were an extremely realistic portrayal of a fantastic event involving the terraforming of an entire planet.

The mini-movie "The Adventures of Andre and Wally B", also completely computer generated, used many computers running the UNIX operating system to produce a cartoon that retains the feel of high-quality human animation. The cartoon makes no attempt at realism, but rather strives to evoke the fantastic cartoon atmosphere in which a bee chasing you with mayhem on its mind becomes as ominous as a P-38 on a strafing

[8] At the time, we considered using a Symbolics "Lisp machine" whose strengths were similar to those for which we chose a UNIX system. The UNIX system won out largely because we could program in both C and Lisp on it.

run.

The single frame (micro-movie?) from the proposed movie "1984" that appeared on the cover of "Science 84" magazine was indistinguishable from a photograph of billiard balls in motion, even to the motion-blur, soft shadows (motion-blurred), and reflections in the balls (with pool hall beer signs, soft shadows, motion-blur, etc.) The images in this frame are entirely realistic.

The short movie "Luxo Jr.", while combining the realism of the 1984 frame and the whimsey of Andre & Wally B, adds an emotional aspect in which the characters (spring armed drafting lamps) take on human personas, displaying joy, sadness, and even emotions as complex as parental sympathy.

All four of these movies were produced on computers running the UNIX operating system and embody many of the philosophical underpinnings of that system. Although the single 1984 frame took many hours of computer time and many, many programs to produce, it was assembled out of individual "tool" programs each of which did a single task well. It's a testimonial to the acceptance of this tool concept that a subject of heated debate around the graphics group at one time was whether or not one of the programs, "reyes"[9], wasn't really doing more than a single task and shouldn't be broken up into smaller, simpler pieces.

Music

Some of the techniques used by the Lucasfilm Graphics Group to produce complex computer imagery from very simple programs fall into a class of algorithms known as "graftals"[10]. One technique in particular, "L-Systems", was originally conceived to imitate the form of growth systems, e.g. plants. The Graphics Group used L-Systems to produce images of plants, grasses, and trees. It occurred to me that the forms produced were very much like the structural form of musical compositions. Using the new generation of inexpensive consumer music synthesizers and a few additional programs I was able to listen to the plants produced instead of looking at them. Again, the modularity of the graftal concept made the reinterpretation of these supposedly graphic entities as musical entities a simple (and relatively graceful) exercise even though the designers of the original L-Systems had no idea that they would be used to produce sound.

Similarly,

In two papers, [LANGSTON86a] and [LANGSTON86b], I describe the music demonstration that grew out of the L-Systems experiment in conjunction with some other algorithmic composition experiments and a telephony project that provides a computer-controlled telephone switch attached to a machine running Eighth Edition UNIX [REDMAN87]. The music demonstration uses software from multiple independent projects, running under three different versions of the UNIX operating system[11]. In assembling the demonstration only two special purpose programs had to be written (one to interconnect the two principal computer systems and one to provide a running commentary on the music being played); the rest of the software was written for other projects and is used intact. The ease with which these elements were integrated into the

[9] This name is not a reference to Pt. Reyes in California (nor to the well-known computer graphic "The Road to Pt. Reyes"). It's an acronym for "renders everything you ever saw".

[10] For more information on graftals consult "Plants, Fractals, and Formal Languages" [SMITH84].

[11] To hear the demo being described, call +1 201 644-2332. If you have U.S. touch-tone you can select which of three demos you will hear.

music demonstration owes much to the flexibility of the operating system under which they were all being developed (the UNIX operating system) and to the networking tools already available in UNIX operating systems. Had these projects been developed under some other operating system then that operating system would have been favored for the music demonstration; as it is, UNIX operating systems are common in software R & D facilities.

Experimentation with sound synthesis equipment connected as peripherals to computers running the UNIX Operating System is currently going on at numerous locations including AT&T Bell Labs [5620/MIDI paper reference], Bell Communications Research, Lucasfilm Ltd., the MIT Media Lab [HAWLEY86], Northwestern University, SDCARL at UCLA San Diego [LOY87], Sun Microsystems Inc., the University of Washington, and others. While no hit records have yet been produced on these systems,[12] thousands of people have listened to the telephone demo[13], and many thousands have played a videogame whose music is produced by "riffology" (Ballblazer).

WHY NOT UNIX?

So far the picture is pretty rosy, what are the disadvantages?

UNIX, as provided by AT&T, is likely to become less "open" and more "user-friendly"[14] as time goes on. The transformation from research tool to commercial product will, almost inevitably, include an increase in the presumptiveness and protectiveness of the system, all in the name of "bullet-proofing."

The UNIX operating system is a time-shared system that gains many efficiencies through optimization for humans working at terminals. It can't provide "real-time" response without giving up these efficiencies or other advantages. Many entertainment uses do not require real-time response, but for those that do, it is not a good choice. My $100 Atari home computer can give better motion to animated graphics than can my $50,000 Sun workstation running UNIX[15].

Other operating systems have much to recommend them and are getting better all the time. The UNIX operating system is just one implementation of a set of ideas and philosophies that are gaining wider and wider acceptance. With few exceptions, those ideas and philosophies are not proprietary, and will doubtless appear in other operating systems if they haven't done so already.

SUMMARY

Entertainment is an area worthy of attention, whether for profit, intellectual challenge, or improving the quality of life. The UNIX operating system owes a debt to entertainment (games, in particular) and appears well able to pay that debt. UNIX & Entertainment not only can work well together, but they already have.

---

[12] Yet.

[13] As of February, 1987 the telephone demo has received just under 5,000 calls.

[14] Unfortunately, software marketers seem to think that the "user" in "user-friendly" is an acronym for "untrainable stooge eating raisins." ("untrainable sap, easily rattled?")

[15] Not entirely a fair comparison, since the Atari has five or ten dollars worth of special purpose graphics hardware built in.

REFERENCES

HAWLEY86 Michael Hawley, "MIDI Music Software for UNIX," Usenix Summer '86 Conference Procedings, (1986)

LANGSTON85 P. S. Langston, "The Influence of UNIX on the Development of Two Video Games", EUUG Spring '85 Conference Procedings, (1985)

LANGSTON86aP. S. Langston, "(201) 644-2332 • Eedie & Eddie on the Wire, An Experiment in Music Generation," Usenix Summer '86 Conference Procedings, (1986)

LANGSTON86bP. S. Langston, "+1 201 644-2332 • Eedie & Eddie Make Beautiful Music in a Distributed Computing environment," EUUG Fall '86 Conference Procedings, (1986)

MIDI85 "MIDI 1.0 Detailed Specification." The International MIDI Association, 11857 Hartsook St., N. Hollywood, CA 91607, (1985)

LOY87 Gareth Loy, "Compositional Algorithms and Music Programming Languages", Manuscript in preparation, (1987)

REDMAN87 B. E. Redman, "A User Programmable Telephone Switch", EUUG Spring '87 Conference Procedings, (1987)

SMITH84 Alvy Ray Smith, "Plants, Fractals, and Formal Languages" *Computer Graphics* Proceedings of the Siggraph '84 Conference, vol. 18, no. 3, pp. 1—10 (July 1984)

THOMPSON85 Ken Thompson, private communication, (January 1985)

Distributed UNIX in Large-Scale Systems

Dale Shipley
Vice-President, Technology.

Tolerant Systems, Inc.
81 East Daggett Drive
San Jose
California 95134
U.S.A.

Introduction

The need for computers that support features such as dynamic expansion and continuous operation is increasing rapidly, as more on-line applications are implemented. According to International Data Corporation (IDC) in its 1985 report, "The Transaction Processing Market", shipments of on-line transaction processing (OLTP) systems will reach $8.5 billion in 1990 - up from $3.7 billion in 1984. Typical OLTP applications included automated teller machines in banking, on-line order entry and inventory control in manufacturing, and reservation systems.

Tolerant Systems chose to address the dynamic expansion requirement by implementing a transparently distributed system based on the UNIX standard. Tolerant's loosely coupled multicomputer architecture provides the user with a single computer image, regardless of the actual number of computers included in a given configuration. With Tolerant's Transaction Executive (TX®) operating system, the distributed system has no effect on the application interface. Standard system calls function as though the application is operating in a single processor environment. No additional naming convention is required to access resources connected to other computers in the network. The UNIX-compatible TX file system also supports consistency and integrity by providing file and block level locking across the network, plus distributed BEGIN/ABORT/COMMIT transaction control. For those applications requiring high availability, continous processing facilities are provided.

This paper describes the system architecture, and facilities of a transparently distributed UNIX based system.

System Architecture

The Eternity® Series architecture is based on a loosely coupled, symmetrical transparent network model. The network may include from two to 40 or more System Building Block (SBB) processing modules. This limit has been set for practical reasons rather than being imposed by the architecture. It is entirely dependent on the processing requirements of a particular application.

TX® is a registered trademark of Tolerant Systems, Inc.
Eternity® is a registered trademark of Tolerant Systems, Inc.

A high-speed, dual bus is used to interconnect the SBBs. Regardless of the number of SBBs, the configuration appears to the user and application programs as one large machine.

The SBB itself contains two 32-bit VLSI microprocessors, one or two 3 megabyte per second I/O channels, the high speed inter-SBB bus controllers, and a main memory subsystem. Disk, tape and communications interfaces are supported by intelligent controllers connected to one I/O channel. Peripherals and controllers are dual-ported to provide interconnections with two SBBs. In this way, the architecture eliminates all single points of failure. In addition, some form of error detection and/or error correction is implemented on all data paths and storage to provide fault detection and isolation.

Each board contains circuitry that identifies the board by type, serial number, ECO (Engineering Change Order) revision level, and firmware revision level. In addition to using this data for maintenance purposes, the TX operating system scans the hardware during boot and automatically configures itself. This eliminates the need to generate the operating system to match a given hardware configuration.

The TX operating system is functionally distributed across the two 32 bit processors in the SBB. One processor - The User Processing Unit (UPU) - implements a UNIX-compatible, time-sliced, demand-paged virtual memory environment for executing application programs. The second processor - the Real-Time Processing Unit (RPU) - is dedicated to the TX operating system and supports preemptive, event-driven, real-time scheduling. The UPU operating system component implements the system call interface and all operating system functions which execute synchronously. The RPU operating system component implements all asynchronous functions (I/O) and all distributed operations. Functions that may be either synchronous or asynchronous, depending on the system state at the time the system call is executed, are tightly coupled across the two processors. For example, the buffer cache is tightly coupled because the data requested by a READ system call may be in the buffer cache or a physical I/O may be required. In the first case, the call is synchronous and therefore is completely processed in the UPU. If a physical I/O is required, the function will be processed by the RPU because the operation is asynchronous. Following the rule for functional distribution, the buffer cache is an example of tight coupling across the two processors.

Communications and network support have been off loaded from the SBB to an intelligent Communications Interface Processor (CIP). This provides an event driven environment ideally suited for real-time terminal handling and network protocol support. For example, the UNIX teletype (TTY) driver and the first three layers of X.25 execute on the CIP. A standard interface is provided for CIP based protocols to communicate with the SBB. Protocols appear to the application program as standard devices in order to retain the inherent simplicity of the UNIX programming interface. Each CIP can handle up to 12 ports, operating to speeds up to 56 kilobaud per second. Each port can be dynamically configured to support a given protocol, so that different protocols can operate on a CIP simultaneously.

The Eternity Series loosely coupled system architecture provides a foundation for fault tolerant continuous operation and on-line expansion of processing power, communications power, and storage capacity. As SBBs are added to a system, the users and application programs continue to view the configuration as though it were one large machine. This single-computer image is achieved by Tolerant's UNIX-compatible TX operating system, which implements the transparently distributed system. In this system, standard system calls function as though the configuration consisted of a single computer. No new naming conventions have been added to implement access to the resources of the

distributed system.

Transparently Distributed Operation

The Eternity Series supports transparent access to all resources by providing a uniform global name space. The name space consists of all addressable objects. These include file systems, directories, files, devices, interprocess communications nodes, and data communications protocols. Application programs gain access to objects via the OPEN system call. Distributed NAME resolution is implemented to locate and open the requested object.

Distributed Name Resolution

Name resolution has been distributed to provide transparent location of any object. When an OPEN system call is executed by an application, the local name resolution function resolves the name to the extent possible. When further resolution is impossible because the necessary portions of the file system hierarchy are stored remotely, a message is passed to a peer level name server running on the computer that can proceed with name resolution. This process is repeated as necessary until the requested object is located and OPENed. A direct linkage is then established between the computer running the application and the kernel level I/O servers on the computer managing the object. All subsequent operations (e.g. READ/WRITE/CLOSE/etc) are handled by passing messages between the requesting computer and the I/O servers on the managing computer. This results in a completely transparent, distributed file system.

The distributed file system name space is composed of a global root file system and all other mounted file systems. The global root is managed by one of the computers in the network. All other computers locate the global root when they are initialised (cold boot) by creating an inode for the global root in a local file system. Because the computer that manages the global root, or any other resource, can change dynamically during system operation, a second level of internal naming is used to logically identify the location of the global root computer. This name is a 64 bit unique address (UID). There is one UID for each resource. The MKNOD (make node) function uses the visable name of the global root to obtain the UID for the global root and stores this in the inode. During the boot process, the operating system switches from a local root to the global root by requesting the network embedded in TX to resolve the UID for the global root to a physical network address. The global root, having been located, makes all objects visible through the distributed name resolution mechanism described earlier.

Distributed name resolution has been optimised by implementing a cache containing path names and their associated UIDs. Entries are made in the cache during name resolution and whenever a file system is mounted. This allows name resolution to skip intermediate nodes during resolution if the path has been seen before.

The Embedded Network

The embedded network or connection system provides linkage between major functional components of the TX operating system by directly linking to the requested function or by sending a message to a server queue on another node. The servers likewise use the connection system to return the result.

As mentioned previously, all objected in the system are identified by UIDs. Of course, file systems and devices are objects. When a file system is mounted, name resolution passes through a file system object or when a device comes on-line, the connection system's routing tables are updated with the UID and network address of the

node that is currently managing that file system or device. Internally, the OPEN function returns the UID to the file system that contains the file being OPENed. TX functions then use this UID through the connection system for all subsequent operations on the OPENed object. This mechanism provides for completely transparent access to all system resources, regardless of the network configuration.

Other distributed System Services

To complete the transparently distributed system, TX provides distributed process management and distributed signal delivery. Process management supports distributed process creation via a new RUN system call which is a combined FORK/EXEC. When used, the new process is created on the node that currently has the lowest load average by default. The most sophisticated user may specify a variety of other site selection criteria to aid load balancing. Signals are also transparently distributed in order to maintain the single computer image for the user.

Summary

Tolerant's Eternity Series provides a fully transparent distributed system. From the user's point of view, the configuration appears as a single computer, regardless of the actual number of computers in the network. The multicomputer environment has no effect on the application interface. Standard system calls function as though the application is operating in a single computer environment. No additional naming convention has been added to support access to resources on other computers in the network.

The system supports dynamic expansion without affecting the application programs. Fault tolerant continuous processing is supported through on-line reconfiguration and process recovery facilities. Transaction processing constructs have been embedded in the file system to provide data consistency and data integrity. All of these features combined make the Eternity a very powerful UNIX-compatible base for on-line transaction processing and data communications applications.

*UNIX for Real Time

Suzanne M. Doughty
Sol F. Kavy
Steven R. Kusmer
Douglas V. Larson

Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino, CA 95014
hplabs!hpda!{sol, kusmer, dvl, suzanne}

ABSTRACT

Adapting UNIX operating systems to real-time markets is a lucrative challenge. By adding a little new functionality and a lot of performance tuning, UNIX systems can support more demanding real-time applications such as those found on the factory floor, tapping into a multi-billion dollar market demanding a portable software environment such as System V. Most of the needed real-time functionality is already found in System V and 4.2BSD. Performance tuning is needed in the area of response time, especially process dispatch latency, which on typical UNIX systems is measured in seconds rather than milliseconds. This paper presents what functionality is needed to adapt UNIX systems to real-time markets, how to acquire the needed performance, and how this combination satisfies real customer needs.

Customer Requirements for Real-Time Systems

The UNIX operating system is found in many marketplaces. It is the operating system for scientific supercomputers and PCs, and it is on desks of software professionals and CEOs. However, one of the final frontiers for the acceptance of UNIX systems is in the real-time marketplace, and for good reason: the real-time customer is the most demanding customer there is. The real-time customer's demands fall within three categories:

Performance

Real-time applications are primarily measured by their performance. Therefore, real-time customers will expect to squeeze the last ounce of performance out of a real-time system to meet their needs, and they will sometimes take measurements their computer vendor never expected. The performance characteristics they measure are typically in terms of response time or throughput. An example of a response time measure is "How long after the receipt of an interrupt from my parallel I/O card can the system run my process which was waiting for that interrupt?". An example of a real-time throughput measure is "How long will it take for me to push my two gigabytes of data from my device to the file system?". The real challenge is that both questions will often be asked by the same customer!

Determinism

Customers expect that a real-time system will react in a deterministic manner. For example, it is not enough to have good response most of the time — you must provide good response *all* of the time. Real-time customers often build a computer into a system that has unforgiving constraints, which is usually because the system is controlling or monitoring other devices or machinery. As an example, a real-time computer built into a steel mill whose steel travels at 30 mph will be expected to respond quickly to an alarm condition. If the computer unexpectedly becomes busy for a whole second, the steel in the steel mill will have traveled 44 feet, and could possibly be strewn over the steel mill floor.

* UNIX is a trademark of AT&T.

Flexibility

In the end, it is real-time customers who truly know best how a real-time computer can solve their applications' needs. Customers must be provided with tools for writing their own drivers and for measuring system performance. They must be provided with source code, because they choose to understand in-depth how a system performs and they might want to tune it for their application. On the other hand, vendors of real-time computers must be humble, because real-time customers are glad to tell them how to build their systems!

The remainder of this paper presents a definition of a real-time system and then explains the real-time features implemented on the HP 9000 Series 800 Model 840. The HP 9000 is HP's computer family for engineering and manufacturing, and it runs HP-UX, a superset of AT&T SVID Issue 1. The Model 840 is the first of HP's new Precision Architecture computer line.

This paper focuses on meeting the above performance and determinism requirements of real-time customers.

What Is a Real-Time System?

A real-time system is a system that can respond in a deterministic and timely manner to events in the real world. *Events in the real world* could mean either large amounts of data that must be processed fast enough to prevent losing the data (data throughput), or discrete events that must be recognized and responded to within certain time constraints (response time).

Specific time requirements depend on the real-time application. For example, in an airlines reservations system, a customer calls an airlines representative and requests a seat reservation on a certain flight. The customer waits on the phone while the real-time system processes the customer's transaction and then responds to the request either with a confirmation or a "flight-is-already-booked" message. The response time requirement for this application is on the order of a second or so, while the customer waits on the phone. If the system fails to meet this requirement, we are left with frustrated customers and perhaps loss of business.

An example of a more demanding real-time application is process monitoring and control in a steam-powered electric plant. Sensors are used to measure variables such as pressure and level of water in the boilers, and speeds of turbines and generators. If, for example, the boiler pressure gets too high, a real-time system must respond immediately with the appropriate action (either reducing the heat source or initiating some cooling action). The response time requirement for this application is on the order of tens of milliseconds. If the system fails to meet this requirement, we could be left with incorrect electrical output or perhaps extensive damage caused by an exploded boiler.

The above applications are just two of the many and varied examples of real-time applications in the world today. Table 1 presents and categorizes additional real-time applications. It is important to note that this list is by no means comprehensive; its purpose is to show the variety and pervasiveness of real-time applications.

Table 1. General Real-Time Applications and Some Examples

General Real-Time Applications	Examples
process monitoring and control	petroleum refinery paper mill chocolate factory
data acquisition	pipe-line sampling data inputs from a chemical reaction
communications	monitoring and controlling satellites telephone switching systems
transaction-oriented processing	airlines reservation systems on-line banking (automatic tellers) stock quotations systems
flight simulation and control	autopilot shuttle mission simulator
factory automation, factory floor control	material tracking, parts production electronic assembly machine or instrument control
transportation	traffic light systems air traffic control
interactive graphics	image processing video games solids modeling
detection systems	radar systems burglar alarm systems

Adding Real-Time Capability to the UNIX Operating System

Given a definition of real time and some sample real-time applications, the next question is "How can the UNIX operating system be augmented to meet the requirements of real-time applications?". While using System V as a base, HP-UX answers this question in two parts: 1) by incorporating functionality from 4.2BSD and adding new functionality from HP, and 2) by doing performance tuning on the kernel and file system. To better understand this approach, it is helpful to be familiar with HP's goals for adding real-time capability to the UNIX operating system:

- Any real-time features implemented must not prevent SVID compatibility.
- Wherever possible, real-time features should be adopted from either System V or 4.2BSD. Only where a needed real-time feature does not exist should HP add a new feature.
- Real-time features must be portable.
- Performance tuning must be transparent to user processes.
- Real-time response must be comparable to real-time response on the HP 1000 A900 (HP's top-of-the-line real-time A-Series computer).

HP-UX on the Model 840 has met these goals. In addition, HP is lobbying through standards-setting bodies to encourage their adoption of HP-UX's real-time features as part of an existing or evolving standard such as SVID or IEEE P1003.

Real-Time Features in HP-UX

This section introduces the real-time features of HP-UX on the Model 840, explains their origin (either System V, 4.2BSD or HP) and also explains how each feature addresses certain concerns about the real-time capability of UNIX systems.

The following features provide real-time capability to HP-UX:

Added Functionality

- Priority-based preemptive scheduling
- Process memory locking
- Privilege mechanism to control access to real-time priorities and memory locking
- Fine timer resolution and time-scheduling capabilities
- Interprocess communication and synchronization
- Reliable signals
- Shared memory for high-bandwidth communication
- Asynchronous I/O for increased throughput
- Synchronous I/O for increased reliability
- Preallocation of disk space
- Powerfail recovery for increased reliability

Performance Tuning

- Kernel preemption for fast, deterministic response time
- Fast file system I/O
- Miscellaneous performance improvements

The functionality additions are discussed first and the performance improvements are discussed later.

Priority-Based Preemptive Scheduling

Priority-based preemptive scheduling lets the most important process execute first, so that it can respond to events as soon as possible. The most important process executes until it sleeps voluntarily or finishes executing, or until a more important process preempts it. **Priority-based** means that a more important process can be assigned a priority higher than other processes, so that the important (high priority) process will be executed before other processes. **Preemptive** means that the high priority process can interrupt or preempt the execution of a lower priority process, instead of waiting for it to be preempted by the operating system when its time slice is completed or it needs to block.

The scheduling policy of traditional UNIX systems strives for fairness to all users and acceptable response time for terminal users. The kernel dynamically adjusts process priorities, favoring interactive processes with light CPU usage at the expense of those using the CPU heavily. Users are given some control of priorities with the `nice(2)` system call, but the nice value is only one factor in the scheduling formula. As a result, it is difficult or impossible to guarantee that one process has a priority greater than another process. Therefore, each process in a traditional UNIX system effectively has to wait its turn, no matter how important it might be to the real-time application.

HP-UX presents a solution to this problem by adding a new range of priorities, called **real-time priorities**. Priorities in the real-time range do not fluctuate like priorities in the normal range, and any process with a priority in the real-time range is favored over any process with a priority in the normal range, including those making system calls and even system processes. Important as real-time processes are, interrupt processing is given priority over them. If several real-time processes have the same priority, they are time-sliced.

Processes with real-time priorities are favored not only for receiving CPU time, but also are favored for swapping and for file system accesses. Real-time processes are the last to be swapped out (except for locked processes — see below), and the first to be swapped in. File system requests for real-time processes go to the head of the disk request queue. All of this preferential treatment gives real-time processes very good response, but at the expense of the rest of the system. (There is no free lunch!)

Real-time priorities are set by the user either programmatically with HP's new `rtprio(2)` system call, or interactively with the `rtprio(1)` command. By default, processes are time-shared and continue to be executed according to the normal scheduling policy. Aside from setting a process to a real-time priority, the `rtprio(2)` system call and `rtprio(1)` command can be used to read the priority of a real-time process and change the priority of a real-time process to be time-shared.

Process Memory Locking

A second important feature in a real-time system is the ability to lock a process in memory so that it can execute without waiting to be paged in or swapped in from disk. In the UNIX and HP-UX operating systems, processes are not normally locked in memory; they are swapped and/or paged in from disk as needed. The time required to swap in a process or page in one or more pages of a process can range from several milliseconds to several seconds, which violates the response time requirements of many real-time applications.

HP-UX has adopted a solution to this problem from System V. The `plock(2)` system call allows a process to lock its executable code and/or its data in memory to avoid unexpected swapping and paging. Also, a process can lock additional data or stack space with the `datalock(3C)` subroutine, and lock shared memory segments as needed with the `shmctl(2)` system call.

Privilege Mechanism to Control Access to Real-Time Capabilities

Because the priority scheduling and memory locking features of HP-UX are quite powerful, it is desirable to allow only certain users to access them. If, for example, all users had access to these capabilities, they could potentially set all of their processes to a high real-time priority and try locking them in memory, which would defeat the purpose of the real-time system. Or, a novice user could assign a real-time priority of 0 to a process that happens to execute in an infinite loop, thus locking up the entire system.

To prevent scenarios such as these, HP-UX created a feature called **privilege groups**. Privilege groups enable certain users (other than just the superuser) to access the powerful real-time priority and memory locking features of HP-UX. A privilege group is a group to which the superuser assigns privileges. Existing privileges are real-time priority assignment (RTPRIO), memory locking (MLOCK) and a third privilege which is not related to real-time functionality. The superuser assigns one or more of these privileges to one or more groups with HP's `setprivgrp(2)` system call or `setprivgrp(1)` command, and assigns certain users to become members of these groups with the 4.2BSD-based `setgroups(2)` system call. You can retrieve the groups you belong to with 4.2BSD's `getgroups(2)` system call or `groups(1)` command, and retrieve the privilege groups you belong to with the `getprivgrp(2)` system call or `getprivgrp(1)` command.

Fine Timer Resolution and Time-Scheduling Capabilities

Another important feature in a real-time operating system is fine timer resolution and time-scheduling capabilities. For high-resolution clock-based applications, both repetitive and nonrepetitive, it is important to be able to execute a process or subroutine at a precise time. For example, a real-time application might require various sensor readings at 20 millisecond intervals.

Standard features in System V that deal with time, such as `alarm(2)` which has a resolution of one second, and `crontab(1)` and `at(1)` which have a resolution of a minute, are not precise enough for many real-time applications. Therefore, HP-UX has adopted a solution from 4.2BSD, known as **interval timers**. Each process can enable its own interval timer to interrupt itself once or at repeated intervals, with whatever precision the underlying hardware and operating system can support. The

interval is defined in units of seconds and microseconds to keep the timer interface portable despite the system-dependent resolution. For HP-UX on the Model 840, the system clock resolution is 10 milliseconds.

Interprocess Communication and Synchronization

A real-time operating system must provide interprocess communication and synchronization facilities. Interprocess communication and synchronization is important because real-time applications often involve several asynchronous processes that need to exchange information. For example, in a manufacturing environment there might be several dedicated computers on a production line, where each executes a process that controls the movements of parts and actions on those parts (such as soldering, molding or welding). A supervisory process that runs on a more powerful computer might monitor the activities of the controller processes. If some type of alarm condition occurs on the production line, the supervisory process can initiate a slowdown or shutdown action. A second process that runs on the supervisory computer could keep track of the inventory levels of each part and inform a third process when more parts must be retrieved and sent to a particular dedicated controller. As you can see, the processes in the supervisory computer must communicate with each other and with the dedicated processes on the production line computers. This is just one example of a group of processes involved in a real-time application that must communicate with each other to get the job done.

Pipes and signals are common interprocess communication facilities in the UNIX operating system. A pipe is essentially an I/O channel through which data is passed with the `read(2)` and `write(2)` system calls. An advantage of using a pipe is that it provides synchronization by blocking reader processes when the channel is empty and blocking writer processes when the channel is full. The disadvantages of using pipes are 1) they require the communicating processes to have a common ancestor process that sets up the channel, and 2) they are often slow because the kernel has to copy the data from the writer process to the system buffer cache and then back again to the address space of the reader process. Many UNIX systems including HP-UX support named pipes, which overcome the first problem, but still have the performance penalty of copying the data.

A signal is essentially a software interrupt sent to a process by the kernel or by a user process. A process can install a handler for almost any signal, and the handler will be executed when the signal is received. Signals can be a good event or alarm mechanism because one process can send a signal to inform another process that an event occurred, and then the other process can immediately enter its handler to respond to the event. The disadvantages of using signals are 1) they pass little or no data (not even who the sender process is), and 2) they are traditionally unreliable when sent repeatedly or when a process tries to wait for a signal.

HP-UX has therefore adopted a reliable signal interface from 4.2BSD, in addition to the System V signal interface. The 4.2BSD signal interface solves the reliability problems of the System V interface, but it is more complicated to use. HP-UX has modified its signal interface to completely emulate both the standard System V interface and the 4.2BSD interface.

HP-UX has also adopted three IPC facilities from System V: shared memory, semaphores and messages. These facilities allow communication and synchronization among arbitrary unrelated processes. An elaborate semaphore facility allows solutions to both simple and complex synchronization problems. A message-passing facility allows transfer of data, along with the ability to prioritize messages. The most important IPC facility for real-time applications is the shared memory facility. Two or more processes can attach the same segment of memory to their data space and then write to and read from it. Shared memory allows the highest communication bandwidth, since data does not have to be copied to be communicated. Recall that a shared memory segment can be locked in memory to provide optimal performance for the communicating processes.

Asynchronous I/O for Increased Throughput

Asynchronous I/O is I/O that overlaps with process execution or other I/O, typically resulting in increased throughput. Both the UNIX and HP-UX operating systems implement system asynchronous I/O to certain drivers, but HP-UX allows you to communicate with some drivers that do system asynchronous I/O, so you can take advantage of their asynchronous abilities.

System asynchronous I/O occurs when the system does asynchronous I/O for a process while the process continues to execute. Two examples where the UNIX and HP-UX operating systems do system asynchronous I/O for user processes are writing to the file system and reading from a terminal when there are enough characters already in the terminal buffer to satisfy the `read(2)`.

HP-UX implements system asynchronous facilities for terminals, pipes, named pipes and sockets. The system asynchronous I/O facilities that HP-UX provides for terminals are:

1. The nonblocking I/O facility: Before launching an I/O request, a user process can set a flag to inform the driver that the driver should cause the I/O request to return immediately if the request cannot be performed without blocking the user process.
2. The SIGIO facility: Before launching an I/O request, a user process can set a flag to enable the driver to send the SIGIO signal to the process when data has arrived in the driver's input buffer.
3. The `select(2)` facility: A user process can call `select(2)` to check if an I/O request should be issued to one or more devices. The driver sets a bit in a user-supplied bit mask for each file descriptor that the user asked about and on which I/O can be performed.
4. The FIONREAD facility: Before launching a read(2) request, a user process can ask the driver to tell it how many characters in the driver's input buffer are available for reading.

These facilities can be used individually or together. For example, suppose you want to read from several terminals and you are not sure which terminal will send you data or when to expect this data, if any. You do not want to launch a series of `read(2)` requests to each terminal, because you might end up missing data from one or more terminals as you try to read from some terminal that will never send you data. Instead, you could enable the SIGIO facility for each terminal so that each can inform you when data has arrived in its input buffer. When SIGIO is sent, you could call `select(2)` to find out which terminal(s) are ready for reading.

Synchronous I/O for Greater Reliability

A real-time application sometimes prefers to do synchronous I/O operations to make sure that its I/O request actually completed. In synchronous I/O, a process initiates an I/O request and then suspends until the I/O request completes. As mentioned above, the file system normally does asynchronous writes, which means that a `write(2)` returns when the data has been written only to the buffer cache, not to the disk. The data is written from the buffer cache to disk later, while the process continues to execute. Although this asynchronous disk write increases your process's throughput, the disadvantage is that you cannot be sure that your data has actually been written to disk. Therefore, HP-UX provides a flag called O_SYNCIO that lets you perform a synchronous disk write. This ensures that your data actually was written to disk.

Powerfail Recovery

Recovery from a power failure is important to real-time applications that cannot afford to lose current data or miss I/O transactions. When power fails, HP-UX saves the CPU state and flushes the data cache to battery-backed up memory. When power is restored, all I/O devices are reset, the CPU state is restored, the cache is reinitialized, I/O transactions in progress at the time of the power failure are restarted if possible, and a signal (SIGPWR) is sent to each process informing it of the power failure. Each process can then take appropriate recovery actions.

Summary of Real-Time Functionality Added to HP-UX

The following table summarizes the functionality that was added to HP-UX. It presents the system call associated with the particular feature and the origin of the system call (either System V, 4.2BSD or HP).

Table 2. Real-Time Functionality in HP-UX

Real-Time Function	Associated system call	Origin
Priority-based preemptive scheduling	rtprio(2)	HP
Process memory locking	plock(2)	System V
Privilege groups	getprivgrp(2) setprivgrp(2)	HP and 4.2BSD
Fine timer resolution and time-scheduling capabilities	setitimer(2) gettimeofday(2)	4.2BSD
Reliable signals	sigvector(2) other calls	4.2BSD
Shared memory for high-bandwidth communication	shmget(2) shmctl(2) shmop(2)	System V
Other interprocess communication and synchronization facilities	pipe(2),msgop(2) msgget(2),msgctl(2) semget(2),semctl(2) semop(2)	System V
Asynchronous I/O for increased throughput	ioctl(2) flags, select(2)	HP and 4.2BSD
Synchronous I/O for increased reliability	fcntl(2) with O_SYNCIO	HP
Preallocating disk space	prealloc(2)	HP
Powerfail recovery	signal(2) with SIGPWR	HP and System V

The performance tuning that HP has implemented in HP-UX on the Model 840 is as important as the added real-time functionality. The following features, kernel preemption, fast file system I/O, and miscellaneous performance improvements comprise the main part of HP's performance tuning efforts.

Kernel Preemption for Faster Response Time

An important requirement for a real-time system is quick and deterministic response time. One of the main concerns about the real-time capability of UNIX systems is that a process can execute in kernel mode for long periods of time (more than 1 second) without allowing a higher priority process to preempt it. Instead, the process keeps executing in kernel mode until it blocks or finishes, while the high priority process must wait. (A process executing in user mode gets preempted much more quickly.)

HP-UX on the Model 840 solves this problem by implementing a **preemptable kernel**. At certain safe places in the kernel called "preemption points" or "preemption regions", HP-UX keeps kernel data structures at a consistent state, so that a higher priority real-time process can get control of the CPU at that point. Kernel preemption is "on" at all times and is invisible to user processes, but it only affects other processes when real-time processes are executing.

The goal of implementing kernel preemption was to significantly decrease the amount of time the kernel executes before it gives up the processor to a waiting higher priority real-time process [1]. This time is known as **process dispatch latency**. Process dispatch latency was measured with a set of software tools that captured stack traces at every preemptable point, along with the time since the last preemptable point. The data was used to add preemption points and regions until the typical measured time was less than a millisecond, and the maximum measured time was a few tens of milliseconds. In other words, the improvement in process dispatch latency was 20 to 100 times better than without kernel preemption. (These measures depend on workload.)

Fast File System I/O

Fast file system performance is important to real-time applications that log data to disk files or read the data logged by other processes. For many of these applications that need quick file access, the traditional file system of the UNIX operating system is not acceptable for the following reasons:

- Data blocks are often scattered randomly throughout the disk, resulting in large disk seek times for sequential reads.
- The data block size is 512 or 1024, which can be inefficient for large read and write requests.
- There is only one superblock, which, if damaged, can make recovery of the file system very difficult or impossible.

The HP-UX file system has adopted its solution from the McKusick or 4.2BSD file system. Two important features in the HP-UX file system are the implementation of cylinder groups, which reduce file seek time and add reliability, and the addition of *two* block sizes which allow increased speed without wasting space on small files.

The HP-UX cylinder group organization reduces seek time because many or all of the data blocks of a given file are on the same cylinder. The file system is composed of one or more cylinder groups. Each is similar to a self-contained file system, as each contains a superblock, a contiguous area of inodes and a contiguous area of data blocks. The data block allocation policies attempt to allocate space from a given file on the same cylinder group, while placing unrelated files in different cylinder groups. Thus, many or all of the data blocks of the file are on the same cylinder, which reduces disk seek time.

A second advantage of using cylinder groups is that having a superblock in each group means redundant copies of the superblock are maintained in case a disk head crash occurs. Also, each superblock on a particular cylinder group is allocated in such a way that destruction of all the copies of the superblocks will not occur if a single disk platter or cylinder is damaged.

The HP-UX file system uses a **hybrid** block size to deal with the time and space tradeoff of big versus small size blocks. There is a block size which is 4K or 8K bytes, and a **fragment** size which is 1/8, 1/4, 1/2 or the same size as this block size. Large file I/O requests are allocated and accessed a block at a time, while smaller requests are allocated and accessed a fragment at a time. The block and fragment size, along with other file system parameters, can be set by the superuser at file system creation time.

The file system of the UNIX operating system normally allocates file space only as data is written. However, HP-UX allows users to **preallocate** file space before ever writing, either programmatically with HP's `prealloc(2)` system call or interactively with the `prealloc(1)` command. Although this preallocated space is not necessarily contiguous, it is allocated in the best possible manner for sequential reading (without moving other data blocks). Also, the time required to do the initial write operations will be reduced, because the space has already been allocated.

Miscellaneous Performance Improvements

HP-UX on the Model 840 is tuned for both real-time response and system throughput. System throughput was increased in two ways. Benchmarks such as the Aim II Benchmark (1) were run, along with HP-produced benchmarks, to track and improve the performance of specific paths in the operating system. Workloads were developed by HP to simulate real-time and other environments. GPROF dynamic call graph measurements were made to justify actions to increase overall system throughput. Other measures, such at time from interrupt to driver, were measured with a logic analyzer interface to the hardware.

This systematic approach to performance tuning led to very significant results, with many performance measures improving by a factor of two or more during product development. Also, this approach led to justifiable returns, including support for two-hand clock replacement algorithm and conversion of various kernel data structures from linear lists to hashed lists. Along the way HP also found a performance bug or two in the ported code, including a bug in the read-ahead mechanism of the buffer cache, which when fixed increased maximum file system throughput by approximately 20%.

Conclusions

The functionality additions and performance improvements described in this paper form the foundation by which HP is enabling its version of the UNIX operating system to successfully enter the real-time marketplace. The features described are rather simple to implement, and in fact, most of them are already in System V or 4.2BSD. HP is working with the IEEE P1003 committee and the real-time subcommittee to help form a common standard by which any vendor can gain the needed functionality.

Acknowledgements

The following people from Hewlett-Packard contributed to this paper: Gary Ho, Bob Lenk and Dave Lennert. Many other individuals at HP and elsewhere have contributed their time, ideas and efforts to the design and implementation of real-time features in HP-UX.

References

[1] David C. Lennert, "Decreasing Realtime Process Dispatch Latency Through Kernel Preemption", *USENIX Conference Proceedings*, Summer 1986, pages 405-413

[2] Robert M. Lenk, "Real-Time Functionality in HP-UX", *TC Interface*, January/February 1986, pages 36-41

(1) Aim II Benchmark is a trademark of Aim Technology.

MINIX: A UNIX clone with source code

Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

MINIX is a complete rewrite of UNIX. Neither the kernel nor the utility programs contains any AT&T code, so the source code is free of the AT&T licensing restrictions and may be studied by individuals or in a course. The system runs on the IBM PC, XT, or AT, and does not require a hard disk, thus making it possible for individuals to acquire a UNIX-like system for home use at a very low cost.

Internally, MINIX is structured completely differently from UNIX. It is a message passing system on top of which are memory and file servers. User processes can send messages to these servers to have system calls carried out. The paper describes the motivation and intended use of the system, what the distribution contains, and discusses the system architecture in some detail.

1. INTRODUCTION

When AT&T first licensed UNIX outside of Bell Laboratories, it was widely studied in operating systems courses at universities (and in industry). Prof. John Lions of the University of New South Wales in Australia even wrote a little booklet providing a commentary on the source code, which for the most part was comment-free. Lions' booklet plus the UNIX source code made it possible for students to get hands-on experience working with, and modifying the code of a real operating system.

With the advent of Version 7, AT&T decided to put an end to the teaching of UNIX to students, and added a clause to the standard university contract prohibiting use of the source code in the classroom. Since that time, professors and students have largely had to be content with operating systems theory because no system that was small enough to be understandable yet large enough to be realistic has been available in source form.

To remedy this situation, several years ago I decided to write a new operating system from scratch that would be system-call compatible with UNIX but completely new inside. In addition to eliminating the licensing problems, this system would be written using modern software concepts such as structured programming, modularity, and file servers. UNIX itself was begun in the early 1970s when the main design issue was squeezing it onto a PDP-11, rather than making the code easy for others to read.

That work is now complete and has resulted in a system called MINIX (mini UNIX) because it leaves out some of the more esoteric system calls in an attempt to make the system smaller and easier to understand. MINIX was originally written for the IBM PC, XT, and AT, but work is currently under way to port it to the 68000 and other

computers. The system is written in C and some care has been taken in the design to make the port to small computers without memory management hardware as straightforward as possible. This point will be discussed in detail later in the paper.

To avoid confusion, it is worthwhile stating explicitly who MINIX is aimed at. There are two potential groups of users.

1. Professors, students, and others who are interesting in legally obtaining and studying the source code of a UNIX-like operating system.
2. People who would like to run a UNIX-like system (especially at home), but have not been able to afford it. Since MINIX does not require a hard disk and the complete system, including both the binaries and the sources costs under $80, the set of potential users is much larger than for UNIX.

Thus MINIX does not really compete with UNIX. Rather, it fills a niche that is currently unoccupied.

For the first category (i.e. educational) users, several options have been provided. The system can be modified and maintained on an IBM PC with or without a hard disk, using itself, a version of UNIX, or even MS-DOS. It can also be cross compiled on a VAX or other time-shared computer running UNIX. Furthermore, the software distribution contains an interpreter for the IBM PC (including I/O) so that the resulting system can be run on a VAX or other computer, preferably a fast one, in case no real IBM PCs are available for students. The MINIX file system can also be modified and run on almost any computer, since it is structured as a free-standing file server. The file server can also be used in a network of diskless workstations.

For the second category (i.e. impoverished) users, several versions of the system have been configured. The normal ones run on 640K PCs with two floppy disks or 512K ATs with one floppy disk, but a special version has also been configured for 256K PCs with only one floppy disk. This version does not contain the C compiler, but is otherwise complete.

MINIX is system-call compatible with Version 7 UNIX for both practical and ideological reasons. On the practical side, I was unable to figure out how to make either 4.3 BSD or System V run on a 256K IBM PC with only 1 floppy disk. On the ideological front, many people (myself included) strongly believe that Version 7 was not only an improvement on all of its predecessors, but also on all of its successors, certainly in terms of simplicity, coherence and elegance. Users who prefer features to elegance should program in Ada† on a large IBM mainframe running MVS.

MINIX implements all the V7 system calls, except ACCT, LOCK, MPX, NICE, PHYS, PKON, PKOFF, PROFIL, and PTRACE. The other system calls are all implemented in full, and are exactly compatible with V7. In particular, FORK and EXEC are fully implemented, so MINIX can be configured as a normal multiprogramming system, with several background jobs running at the same time (memory permitting), and even multiple users.

The MINIX shell is compatible with the V7 (Bourne) shell, so to the user at the terminal, running MINIX looks and feels like running UNIX. Over 70 utility programs are part of the software distribution, including ar, basename, cat, cc, chmem, chmod, chown,

† Ada is a Registered Trademark of the U.S. Dept. of Defense

cmp, comm, cp, date, dd, df, echo, grep, head, kill, ln, login, lpr, ls, make, mkdir, mkfs, mknod, mount, mv, od, passwd, pr, pwd, rev, rm, rmdir, roff, sh, size, sleep, sort, split, stty, su, sum, sync, tail, tar, tee, time, touch, tr, umount, uniq, update, and wc. A full-screen editor inspired by Emacs (think of it as nano-emacs), a full K&R compatible C compiler, and programs to read and write MS-DOS diskettes are also included. All of the sources of the operating system and these utilities, except the C compiler source (which is quite large and is available separately), are included in the software package.

In addition to the above utilities, over 100 library procedures, including stdio, are provided, again with the full source code.

To reiterate what was said above, all of this software is completely new. Not a single line of it is taken from, or even based on the AT&T code. I personally wrote from scratch the entire operating system and some of the utilities. This took about 3 years. My students and some other generous people wrote the rest. The C compiler is derived from the Amsterdam Compiler Kit (Tanenbaum et al., 1983), and was written at the Vrije Universiteit. It is a top-down, recursive descent compiler written in a compiler writing language called LLGEN and is not related to the AT&T portable C compiler, which is a bottom-up, LALR compiler written in YACC.

2. OVERVIEW OF THE MINIX SYSTEM ARCHITECTURE

UNIX is organized as a single executable program that is loaded into memory at system boot time and then run. MINIX is structured in a much more modular way, as a collection of processes that communicate with each other and with user processes by sending and receiving messages. There are separate processes for the memory manager, the file system, for each device driver, and for certain other system functions. This structure enforces a better interface between the pieces. The file system cannot, for example, accidentally change the memory manager's tables because the file system and memory manager each have their own private address spaces.

These system processes are each full-fledged processes, with their own memory allocation, process table entry and state. They can be run, blocked, and send messages, just as the user processes. In fact, the memory manager and file system each run in user space as ordinary processes. The device drivers are all linked together with the kernel into the same binary program, but they communicate with each other and with the other processes by message passing.

When the system is compiled, four binary programs are independently created: the kernel (including the driver processes), the memory manager, the file system, and *init* (which reads */etc/ttys* and forks off the login processes). In other words, compiling the system results in four distinct *a.out* files. When the system is booted, all four of these are read into memory from the boot diskette.

It is possible, and in fact, normal, to modify, recompile, and relink, say, the file system, without having to relink the other three pieces. This design provides a high degree of modularity by dividing the system up into independent pieces, each with a well-defined function and interface to the other pieces. The pieces communicate by sending and receiving messages.

The various processes are structured in four layers:

4. The user processes (top layer).
3. The server processes (memory manager and file system).
2. The device drivers, one process per device.

1. Process and message handling (bottom layer).

Let us now briefly summarize the function of each layer.

Layer 1 is concerned with doing process management including CPU scheduling and interprocess communication. When a process does a SEND or RECEIVE, it traps to the kernel, which then tries to execute the command. If the command cannot be executed (e.g., a process does a RECEIVE and there are no messages waiting for it), the caller is blocked until the command can be executed, at which time the process is reactivated. When an interrupt occurs, layer 1 converts it into a message to the appropriate device driver, which will normally be blocked waiting for it. The decision about which process to run when is also made in layer 1. A priority algorithm is used, giving device drivers higher priority over ordinary user processes, for example.

Layer 2 contains the device drivers, one process per major device. These processes are part of the kernel's address space because they must run in kernel mode to access I/O device registers and execute I/O instructions. Although the IBM PC does not have user mode/kernel mode, most other machines do, so this decision has been made with an eye toward the future. To distinguish the processes within the kernel from those in user space, the kernel processes are called **tasks**.

Layer 3 contains only two processes, the memory manager and the file system. They are both structured as **servers**, with the user processes as **clients**. When a user process (i.e. a client) wants to execute a system call, it calls, for example, the library procedure *read* with the file descriptor, buffer, and count. The library procedure builds a message containing the system call number and the parameters and sends it to the file system. The client then blocks waiting for a reply. When the file system receives the message, it carries it out and sends back a reply containing the number of bytes read or the error code. The library procedure gets the reply and returns the result to the caller in the usual way. The user is completely unaware of what is going on here, making it easy to replace the local file system with a remote one.

Layer 4 contains the user programs. When the system comes up, *init* forks off login processes, which then wait for input. On a successful login, the shell is executed. Processes can fork, resulting in a tree of processes, with *init* at the root. When CTRL-D is typed to a shell, it exits, and *init* replaces the shell with another login process.

3. LAYER 1 - PROCESSES AND MESSAGES

The two basic concepts on which MINIX is built are processes and messages. A process is an independently schedulable entity with its own process table entry. A message is a structure containing the sender's process number, a message type field, and a variable part (a union) containing the parameters or reply codes of the message. Message size is fixed, depending on how big the union happens to be on the machine in question. On the IBM PC it is 24 bytes.

Three kernel calls are provided:

- RECEIVE(source, &message)
- SEND(destination, &message)
- SENDREC(process, &message)

These are the only true system calls (i.e. traps to the kernel). RECEIVE announces the willingness of the caller to accept a message from a specified process, or ANY, if the RECEIVER will accept any message. (From here on, "process" also includes the tasks.)

If no message is available, the receiving process is blocked. SEND attempts to transmit a message to the destination process. If the destination process is currently blocked trying to receive from the sender, the kernel copies the message from the sender's buffer to the receiver's buffer, and then marks them both as runnable. If the receiver is not waiting for a message from the sender, the sender is blocked.

The SENDREC primitive combines the functions of the other two. It sends a message to the indicated process, and then blocks until a reply has been received. The reply overwrites the original message. User processes use SENDREC to execute system calls by sending messages to the servers and then blocking until the reply arrives.

There are two ways to enter the kernel. One way is by the trap resulting from a process' attempt to send or receive a message. The other way is by an interrupt. When an interrupt occurs, the registers and machine state of the currently running process are saved in its process table entry. Then a general interrupt handler is called with the interrupt number as parameter. This procedure builds a message of type INTERRUPT, copies it to the buffer of the waiting task, marks that task as runnable (unblocked), and then calls the scheduler to see who to run next.

The scheduler maintains three queues, corresponding to layers 2, 3, and 4, respectively. The driver queue has the highest priority, the server queue has middle priority, and the user queue has lowest priority. The scheduling algorithm is simple: find the highest priority queue that has at least one process on it, and run the first process on that queue. In this way, a clock interrupt will cause a process switch if the file system was running, but not if the disk driver was running. If the disk driver was running, the clock task will be put at the end of the highest priority queue, and run when its turn comes.

In addition to this rule, once every 100 msec, the clock task checks to see if the current process is a user process that has been running for at least 100 msec. If so, that user is removed from the front of the user queue and put on the back. In effect, compute bound user processes are run using a round robin scheduler. Once started, a user process runs until either it blocks trying to send or receive a message, or it has had 100 msec of CPU time. This algorithm is simple, fair, and easy to implement.

4. LAYER 2 - DEVICE DRIVERS

Like all versions of UNIX for the IBM PC, MINIX does not use the ROM BIOS for input or output because the BIOS does not support interrupts. Suppose a user forks off a compilation in the background and then calls the editor. If the editor tried to read from the terminal using the BIOS, the compilation (and any other background jobs such as printing) would be stopped dead in their tracks waiting for the the next character to be typed. Such behavior may be acceptable in the MS-DOS world, but it certainly is not in the UNIX world. As a result, MINIX contains a complete set of drivers that duplicate the functions of the BIOS. Like the rest of MINIX, these drivers are written in C, not assembly language.

This design has important implications for running MINIX on PC clones. A clone whose hardware is not compatible with the PC down to the chip level, but which tries to hide the differences by making the BIOS calls functionally identical to IBM's will not run an unmodified MINIX because MINIX does not use the BIOS.

Each device driver is a separate process in MINIX. At present, the drivers include the clock driver, terminal driver, various disk drivers (e.g., RAM disk, floppy disk), and printer driver. Each driver has a main loop consisting of three actions:

1. Wait for an incoming message.
2. Perform the request contained in the message.
3. Send a reply message.

Request messages have a standard format, containing the opcode (e.g., READ, WRITE, or IOCTL), the minor device number, the position (e.g., disk block number), the buffer address, the byte count, and the number of the process on whose behalf the work is being done.

As an example of where device drivers fit in, consider what happens when a user wants to read from a file. The user sends a message to the file system. If the file system has the needed data in its buffer cache, they are copied back to the user. Otherwise, the file system sends a message to the disk task requesting that the block be read into a buffer within the file system's address space (in its cache). Users may not send messages to the tasks directly. Only the servers may do this.

MINIX supports a RAM disk. In fact, the RAM disk is always used to hold the root device. When the system is booted, after the operating system has been loaded, the user is instructed to insert the root file system diskette. The file system then sees how big it is, allocates the necessary memory, and copies the diskette to the RAM disk. Other file systems can then be mounted on the root device.

This organization puts important directories such as */bin* and */tmp* on the fastest device, and also makes it easy to work with either floppy disks or hard disks or a mixture of the two by mounting them on */usr* or */user* or elsewhere. In any event, the root device is always in the same place.

In the standard distribution, the RAM disk is about 240K, most of which is full of parts of the C compiler. In the 256K system, a much smaller RAM disk has to be used, which explains why this version has no C compiler: there is no place to put it. (The */usr* diskette is completely full with the other utility programs and one of the design goals was to make the system run on a 256K PC with 1 floppy disk.) Users with an unusual configuration such as 256K and three hard disks are free to juggle things around as they see fit.

The terminal driver is compatible with the standard V7 terminal driver. It supports cooked mode, raw mode, and cbreak mode. It also supports several escape sequences, such as cursor positioning and reverse scrolling because the screen editor needs them.

The printer driver copies its input to the printer character for character without modification. It does not even convert line feed to carriage return + line feed. This makes it possible to send escape sequences to graphics printers without the driver messing things up. MINIX does not spool output because floppy disk systems rarely have enough spare disk space for the spooling directory. Instead one normally would print a file *f* by saying

```
lpr <f &
```

to do the printing in the background. The *lpr* program insert carriage returns, expands tabs, and so on, so it should only be used for straight ASCII files. On hard disk systems, a spooler would not be difficult to write.

5. LAYER 3 - SERVERS

Layer 3 contains two server processes: the memory manager and the file system. They are both structured in the same way as the device drivers, that is a main loop that accepts requests, performs them, and then replies. We will now look at each of these in turn.

The memory manager's job is to handle those system calls that affect memory allocation, as well as a few others. These include FORK, EXEC, WAIT, KILL, and BRK. The memory model used by MINIX is exceptionally simple in order to accommodate computers without any memory management hardware. When the shell forks off a process, a copy of the shell is made in memory. When the child does an EXEC, the new core image is placed in memory. Thereafter it is never moved. MINIX does not swap or page.

The amount of memory allocated to the process is determined by a field in the header of the executable file. A program, *chmem*, has been provided to manipulate this field. When a process is started, the text segment is set at the very bottom of the allocated memory area, followed by the data and bss. The stack starts at the top of the allocated memory and grows downward. The space between the bottom of the stack and the top of the data segment is available for both segments to grow into as needed. If the two segments meet, the process is killed.

In the past, before paging was invented, all memory allocation schemes worked like this. In the future, when even small microcomputers will use 32-bit CPUs and 1M x 1 bit memory chips, the minimum feasible memory will be 4 megabytes and this allocation scheme will probably become popular again due to its inherent simplicity. Thus the MINIX scheme can be regarded as either hopelessly outdated or amazingly futuristic, as you prefer.

The memory manager keeps track of memory using a list of holes. When new memory is needed, either for FORK or for EXEC, it searches the hole list and takes the first hole that is big enough (first fit). When a process terminates, if it is adjacent to a hole on either side, the process' memory and the hole are merged into a bigger hole.

The file system is really a remote file server that happens to be running on the user's machine. However it is straightforward to convert it into a true network file server. All that needs to be done is change the message interface and provide some way of authenticating requests. (In MINIX, the source field in the incoming message is trustworthy because it is filled in by the kernel.) When running remote, the MINIX file server maintains state information, like RFS and unlike NFS.

The MINIX file system is similar to that of V7 UNIX. The i-node is slightly different, containing only 9 disk addresses instead of 13, and only 1 time instead of 3. These changes reduce the i-node from 64 bytes to 32 bytes, to store more i-nodes per disk block and reduce the size of the in-core i-node table.

Free disk blocks and free inodes are kept track of using bit maps rather than free lists. The bit maps for the root device and all mounted file systems are kept in memory. When a file grows, the system makes a definite effort to allocate the new block as close as possible to the old ones, to minimize arm motion. Disk storage is not necessarily allocated one block at a time. A minor device can be configured to allocate 2, 4 (or more) contiguous blocks whenever a block is allocated. Although this wastes disk space, these multiblock **zones** improve disk performance by keeping file blocks close together. The standard parameters for MINIX as distributed are 1K blocks and 1K zones (i.e. just 1 block per zone).

MINIX maintains a buffer cache of recently used blocks. A hashing algorithm is used to look up blocks in the cache. When an i-node block, directory block, or other critical block is modified, it is written back to disk immediately. Data blocks are only written back at the next SYNC or when the buffer is needed for something else.

The MINIX directory system and format is identical to that of V7 UNIX. File names are strings of up to 14 characters, and directories can be arbitrarily long.

6. LAYER 4 - USER PROCESSES

This layer contains *init*, the shell, the editor, the compiler, the utilities, and all the user processes. These processes may only send messages to the memory manager and the file system, and these servers only accept valid system call requests. Thus the user processes do not perceive MINIX to be a general-purpose message passing system. However, removing the one line of code that checks if the message destination is valid would convert it into a much more general system (but less UNIX-like).

7. DOCUMENTATION

For a system one of whose purposes is teaching about operating systems, ample documentation is essential. For this reason I have written an ample textbook (more than 700 pages) treating both the theory and the practice of operating system design (Tanenbaum, 1987). The table of contents is as follows:

CHAPTERS

1. Introduction
2. Processes
3. Input/Output
4. Memory Management
5. File Systems
6. Bibliography and Suggested Readings

APPENDICES

A. Introduction to C
B. Introduction to the IBM PC
C. MINIX Users Guide
D. MINIX Implementers Guide
E. MINIX Source Code Listing
F. MINIX Cross Reference Map

The heart of the book is chapters 2-5. Each chapter deals with the indicated topic in the following way. First comes a thorough treatment of the relevant principles (thorough enough to be usable as a university textbook on operating systems). Next comes a general discussion of how the principles have been applied in MINIX. Finally there is a procedure by procedure description of how the relevant part of MINIX works in detail. The source code listing of appendix E contains line numbers, and these line numbers are used throughout the book to pinpoint the code under discussion. The source code itself contains more than 3000 comments, some more than a page long. Studying the principles and seeing how they are applied in a real system gives the reader a better understanding of the subject than either the principles or the code alone would.

Appendices A and B are quickie introductions to C and the IBM PC for readers not familiar with these subjects. Appendix C tells how to boot MINIX, how to use it, and how

to shut it down. It also contains all the manual pages for the utility programs. Most important of all, it gives the super-user password.

Appendix D is for people who wish to modify and recompile MINIX. It contains a wealth of nutsy-boltsy information about everything from how to use MS-DOS as a development system, to what to do when your newly made system refuses to boot.

Appendix E is a full listing of the operating system, all 260 pages of it. The utilities (mercifully) are not listed.

8. DISTRIBUTION OF THE SOFTWARE

The software distribution is being done by Prentice-Hall. Four packages are available. All four contain the full source code; they differ only in the configuration of the binary supplied. The four packages are:

- 640K IBM PC version
- 256K IBM PC (no C compiler)
- IBM PC-AT (512K minimum)
- Industry standard 9-track tape

The 640K version will also run on 512K systems, but it may be necessary to *chmem* parts of the C compiler to make it fit. The tape version is the only one containing the IBM PC simulator and other software needed for classroom use on a VAX or other time sharing machine. The software packages do not include the book.

If there is sufficient interest, a newsgroup net.minix will be set up. This channel can be used by people wishing to contribute new programs, point out and correct bugs, discuss the problems of porting MINIX to new systems, etc.

9. ACKNOWLEDGEMENTS

I would like to thank the following people for contributing utility programs and advice to the MINIX effort: Martin Atkins, Erik Baalbergen, Charles Forsyth, Richard Gregg, Michiel Huisjes, Patrick van Kleef, Adri Koppes, Paul Ogilvie, Paul Polderman, and Robbert van Renesse. Without their help, the system would have been far less useful than it now is.

10. REFERENCES

Tanenbaum, A.S., van Staveren, H., Keizer, E.G., and Stevenson, J.W.: "A Practical Toolkit for Making Portable Compilers," *Communications of the ACM*, vol. 26, pp. 654-660, Sept. 1983.

Tanenbaum, A.S.: *Operating Systems: Design and Implementation*, Englewood Cliffs, N.J.: Prentice-Hall, 1987.

A User Programmable Telephone Switch

Brian E. Redman

Bell Communications Research
Morristown, New Jersey 07960, USA

ABSTRACT

The basic function of a telephone switch is to allow subscribers to place calls among one another. The basic service provided is Plain Old Telephone Service (POTS). There were relatively few changes in POTS since telephone switching was introduced in 1880. In 1919 dial service became available alleviating the need for operator assistance on many calls. Direct Distance Dialing (DDD) in 1951 expanded this to long distance calls. In 1964 touch-tone service provided faster and easier dialing. It wasn't until 1972 that Vertical Services were offered to residence costumers. These were services such as Speed Calling, Call Waiting and Call Forwarding.

When you rented a pair of telephones in 1887 there was only one option available. For an additional $5 installation charge they were equipped with "thumpers", the predecessor of the bell. Otherwise you could simply yell into your telephone and hope the other party was close enough to theirs to hear you. When you subscribe to telephone service today you are offered a number of enhancements to POTS. Unfortunately the precise behavior and control of these options is quite limited.

Nowadays telephone switching systems are controlled by computers. There is the capacity to do more than switch calls among subscribers. It is both practical and attractive to have telephone services controlled dynamically by the subscriber, either via direct input to the telephone switching system or by exercising customer designed control algorithms.

The telephone switching system which will be described provides several interfaces to the subscriber. At the highest levels, the user can activate or deactivate preprogrammed algorithms and modify their behavior to the extent that such modification has been allowed for in their design. This is achieved with touch-tone input or by issuing commands from a computer terminal. At the intermediate level the user can incorporate program library functions and implement control algorithms with their own computer programs. At the lowest level users can claim total control of their assigned circuits.

This system has been in use for over a year, providing essential telephone service to twenty subscribers. Although the basic design has remained intact, the emphasis on utilization of the different layers is shifting markedly.

1. Introduction

The BerBell user programmable telephone switch places into the hands of its customers the responsibility of determining how their telephone should behave beyond a basic standard service. There are so many options available in modern systems that it is not reasonable for their designers or installers to predict or restrict the desires of each individual. By providing a powerful set of primitives and a clean interface, the subscribers themselves or their agents can dictate the functionality of their service. Thus service definition is open-ended, evolving with the needs and imagination of the system's users. The exploitation of BerBell's capabilities has resulted in a continually growing library of user programs and services. These include placing calls from an on-line directory and scheduling calls from an on-line calendar. Users with computer access can take somewhat greater advantage of BerBell. Whenever possible, the telephone touch-tone interface provides similar capabilities to the computer terminal interface. However more complex features are more easily manipulated with the use of textual input and output. Although rotary dialing cannot be fully supported, BerBell will function well with all types of touch-tone telephones and computer terminals. Putting the data bases and features within the system or within reach of the system through computer networks obviates the need for expensive special purpose accessories.

The work described involves the use of a general purpose operating system, UNIX, and its associated program resources to support the development and application of a telephone switch. The system as a whole is referred to as BerBell. Its software consists of a core program, *bellerophon*, a number of programs varying in their degree of independence from *bellerophon*, the UNIX operating system and its tools. The hardware which realizes the system is composed of a host minicomputer, a Redcom Modular Switching Peripheral providing the basic electrical interfaces and switching capabilities required for telephony applications, speech synthesizers and an assortment of ancillary audio sources and recorders. These components together provide flexible and comprehensive telephone service.

2. User Level

How is BerBell different from other telephone offerings from the end-user's point of view? What new capabilities are there? There are no significant differences between the basic service provided by BerBell and that provided by other vendors. The default BerBell dialing plan is designed to look like CENTREX since most subscribers use it at work. However, dialing plans are associated with the telephone line, so home subscribers can use the standard residence dialing plan. Basic service implies that when the subscriber lifts the receiver, they hear dialtone. They then dial a valid number and a call is placed to their party. On the receiving side, if the telephone is in use callers get a busy tone. Otherwise the telephone is rung. If the receiving party answers, a talking connection is established.

BerBell and most other vendors provide more interesting services upon request. BerBell subscribers can activate and disconnect these features using a touch-tone telephone or by issuing shell level commands from a computer served by the BerBell host. Although the fundamental concepts of these features are similar, BerBell advances their applications. First, we present a discussion of those features that are typically provided by most vendors.

2.1. Call Forwarding

Subscribers can arrange to have their calls transferred to another number. Typical residence service only allows unconditional call forwarding. Most PBX and CENTREX vendors provide call forwarding when the called line is busy as well as after the line has rung some number of times (no answer). These functions are available to BerBell subscribers with some improvements. Most importantly, the parameters of each of the call forwarding operations are conveniently changeable by the subscriber. From a touch-tone telephone, the user enters '*' (non-call dialing), then '2' indicating a feature setting, followed by '1' (call forwarding) then various codes to set parameters. The feature dialing syntax is designed to be consistent and hierarchical. It is simpler to use the computer interface to describe these parameters. In all cases the command is

```
setforward <extension> <option>.
```

The basic call forwarding options are:

`rings <n> -`
: the number of rings after which no-answer forwarding will be effected. If `<n>` is zero unconditional forwarding is activated.

`busy/nobusy -`
: activate/deactivate forwarding when the line is busy.

`noanswernumber <number> -`
: the number to transfer the call to if no-answer forwarding is active and the call is not answered after the specified number of rings. The name of a program can be substituted for <number>.

`unconditionalnumber <number> -`
: the number or program to transfer the call to if `rings` is set to zero.

`busynumber <number> -`
: the number or program to transfer the call to if the line is busy and busy forwarding is activated.

Some new call forwarding operations are implemented to allow callers and recipients of forwarded calls to be made aware that call forwarding has been invoked. The terms 'inform' and 'announce' are used to indicate messages to the caller and ultimate recipient respectively. The following options involve speaking a text message synthetically or playing a recorded message.

`inform/noinform -`
: enable/disable calling party notification that the call is being forwarded.

`announce/noannounce -`
: enable/disable recipient party notification that the call has been forwarded to them.

For each of the forwarding conditions described a different message can be specified with `<file>` which contains text to be recited or binary audio data.

```
busyannounce <file>
noansweranno unce <file>
```

```
unconditionalinform <file>
```

A 'continueringing' option allows the originally called line to continue ringing even after it has been forwarded. It can then be answered at any time. This is disabled with 'stopringing'.

Finally, the 'on/off' option will activate/deactivate all forwarding without modifying the parameter settings described above.

2.2. Call Waiting

A call to a party whose line is busy causes audible ringing to be heard by the caller and a short tone by the recipient. The recipient may then talk to the calling party by hookflashing, placing the current conversation on hold. In the BerBell implementation call waiting can be enabled or disabled from the telephone by pressing '*20' then '1' or '0' respectively. From a terminal the command

```
setcw [on | off] <program name>
```

is used to specify a program which is executed when the caller encounters a busy line with call waiting enabled. As in call forwarding and most other services the program can be supplied by the user. A popular program in use for this purpose informs the caller that their party will be responding shortly, then connects them to silence, music or an answering program, at their option. The recipient, having heard a high-pitched tone when the new call arrived will hear a low-pitched tone if the new caller should hang up. In fact the new call is simply a held call and the subscriber will hear a low-pitched tone any time a held caller hangs up. Many calls can be on hold simultaneously.

2.3. Call Transfer

A call in progress can be forwarded to another number. This is a fairly common feature but BerBell provides a slight twist. In the normal case a call is transferred using the telephone by first placing it on hold, then dialing '*12', then the slot number or '#' for the oldest held call, then the destination number. From the terminal the user issues

```
xfer <extension> <destination number>.
```

In this case the call in progress, not a held call, is transferred. The twist mentioned is that calls can be temporarily transferred. That is to say that the call is transferred but still remains on hold. This means that the user can still pick up the call, etc. This feature conveniently implements services on hold such as music, advertising, games, or information. Each of these services is implemented as a program which can be designed by the subscriber. Programs exist which give the caller the option to select a preference (including silence). In any event the options are dictated by the subscriber and the choice is made by the user, not the system. To invoke the temporary transfer from a telephone, place the call on hold then dial '*16#', then the number to call. The command to issue from a terminal is

```
txfer <extension> <number>.
```

Like 'xfer' the current call is transferred, not a held call.

The more common features found in most telephone systems have been covered. The discussion now turns to some less common services.

2.4. Editing

BerBell incorporates some editing facilities similar to those used for computer terminal interfaces. These are '*##', erase the last keypress; '**#' erase all dialed digits; and '***' recite the digits dialed so far.

2.5. Call Announcement

A program can be executed when a subscriber receives a call. The command

```
setcallfor [on | off] <program>
```

is used to control the option. From the telephone, '*221' and '*220' are used to activate or deactivate the feature. The default program utilizes a switched public address system to announce to subscribers that their telephones are ringing. The user can provide a text or data file to be spoken by a speech synthesizer or played by a digital sound device. Another file specifies at which locations the announcement is to made. Users can answer their calls from any BerBell extension.

2.6. Additional Manipulations of Calls

It is possible to pick up a call that is ringing on another telephone, or to redirect it to another number. Other functions can be performed on held calls. By dialing '*14#' the user invokes 'hold-on-hold'. This program, dubbed 'revenge on hold' and designed on a whim, causes the held call to repetitively receive a message advising the party to dial '*' when they return to the telephone. The user's line is freed for incoming or outgoing calls. When the held party does dial '*' the user's telephone is rung just as if they were receiving a call, and upon answering they are connected to their party.

Two additional features which apply to held calls are held retrieval and held transfer. Held transfer allows a user to transfer a held call to another extension while in the held state. Thus the call appears, still on hold, on another extension. The code is '*15#', then the destination extension. Held retrieval permits a user to pick up a call which is on hold on another extension. The code for this is '*18#', then the extension from which the held call will be retrieved.

2.7. Programs

User programs provide a large portion of BerBell's functionality. The system is designed specifically to give the programmer flexibility and control over the telephone switch. Users issue commands from the BerBell host or remotely from any machine the communicates with the host over the Internet. In the 4.3 BSD implementation, which uses Internet Datagrams to communicate among processes within BerBell, the programs that are used locally function identically over the network. What follows is a brief description of some of the programs in common use now. The list is not exhaustive and continues to grow. One should also note that many of the functions described above will be recoded as stand alone programs.

2.7.1. Accessed from the telephone

The following list of programs are services invoked by *bellerophon* as a subscriber option or a general service.

`bebap`
: The default answering program. It allows callers to leave a number or a message, or connect to a message taking persons. Callers can also receive messages that are left for them.

`ttda`
: Touch-tone directory assistance[1] permits users to look up telephone numbers in various telephone books. The user can be transferred to the matched number. Current directories include Bellcore, Seattle, and many New Jersey books.

`ttweather`
: Using touch-tone input, the user gets the National Weather Service forecast for any desired city in the continental U.S.

`ttsh`
: Using a touch-tone mapping scheme where two keypresses represent one ASCII character, a user can log in to the BerBell host and execute commands.

`demo2332`
: The audio research laboratory real-time music generation demonstration.

`wakeup`
: The wakeup service calls the user at a desired time, and delivers a message.

`ccstatus`
: Computer Center Status provides information to the caller and permits them to enter their number to receive updates. When staff personnel change the status file, users are called back.

`wrong`
: The wrong-number server randomly executes one of many BERPS scripts to entertain and confuse callers who dial invalid BerBell extensions. BERPS is described in a subsequent section. These scripts include simulations of telephone interfaces to various institutions such as banks, the Federal Government, the Phone Company, the Defense Department and the Underworld.

`robop`
: The robot operator recites the list of dial-up services.

`help`
: An interactive program for feature dialing assistance.

`htime`
: Recites the local time, date, and weather.

`musak`
: Connects the caller to an arbitrary audio source.

`ph/silence`
: These services are primarily for testing, connecting the caller to a permanent high-pitched tone or to silence.

`hanna`
: Simulates the utterances of a three year old child.

`newsgen`
: Assembles an inane scandal sheet news story from a table of phrases and personalities.

`rosary`
: Recites same.

`winner`
: Announces to the caller that they have won something and asks them to hold on (indefinitely).

`marge`
: Simulates the gratuitous utterances of a cashier.

`suicide`
: An irreverent suicide hot-line.

The following programs are issued from the user's terminal.

`call <from> <to>`
: The `<from>` number is called. When it is answered, a call is placed from it to `<to>` number.

`nway <number> <number> <number> [number] ...`
: Conference calls are established with this command.

`gm [options] <from> <to> [message file]`
: Getme dials the `<to>` number and may deliver the optional message contained in *message file*. *options* may be used to elicit a response from the answerer at `<to>` number. If a satisfactory response is received or if no response is required, the call is connected to `<from>` number.

`pa <number> <message file>`
: `<number>` is dialed and the contents of `<message file>` is delivered.

3. The Programmers View

There are several programming interfaces to BerBell. For implementing user level functions such as the programs described above, there are C language library routines, the BERPS interpretive language and the UNIX shell. Programs may be invoked implicitly by a user or by the system or explicitly by typing at a computer terminal. The system will invoke a program when an inbound call to a number associated with that program is received (e.g., directory assistance), when a feature implicates a program (e.g., recite speed codes) or when a user's feature settings dictate (e.g., call waiting). In each of these cases the arguments to the program will include the name of the circuit holding the call and the number dialed. Programs can then connect auxiliary devices such as tones, recorded audio, speech synthesizers, answering machines and audio components to the call, reroute the call, hold or release it.

3.1. C Programming

There are C libraries for manipulating BerBell objects, speech synthesizers, touch-tone receivers and digital audio input and output channels. The BerBell library contains six 'system calls' that allow the programmer to easily connect objects such as trunks and lines* among one another and to place telephone calls using these objects. It is straightforward to write C programs using this library. No significant knowledge of telephony is required to take substantial control over your communications channels.

3.2. BERPS interpreter

BERPS stands for BER Phone Script. It is a simple language that interprets scripts which manipulate telephones. It is designed to alleviate the need for users to program in C in order to write application programs. The wrong-number servers are written in BERPS as well as an answering, call waiting, and call announcement program.

* Trunks connect telephone switches together, lines connect telephone instruments to switches. Within BerBell, audio equipment is connected to the switch by trunk interfaces.

The interpreter itself was written using the library functions. The language provides for flow control, arithmetic calculations, BerBell operations and operating system services. The following example implements one of the wrong-number services.

```
# Trans Galactic Phone Network      [a comment.]
%h z      [jump to label z (:z) when the caller hangs up.]
You have connected to the intergalactic communications network.
Please enter galaxy code, use sharp or pound to terminate.
%r g      [read a number input by the user into the variable 'g'.]
Now enter planet destination.
%r p
%h 1
%= p 10      [execute commands until the '%}' if the variable 'p' has the value '10'.]
Planet Ten is in the Eighth Dimension.
Please dial the interdimension operator for assistance.
%! p 10      [the 'not equal' case.]
We are sorry, planet $Np in galaxy $Ng has been obliterated.
Please check the code and call again.
%}
:1
%F /logs/wrong $T: galaxy $ng planet $np ($P)      [append to file.]
%e 0      [exit.]
:z
%F /logs/wrong $T: caller hung up ($P)
```

Text is simply spoken through a speech synthesizer. **$Np** expands the value of **p** to text as digits separated by spaces. **$n** expands a variable as a single number. **$T** is the current time, **$P** is the process identifier.

3.3. Shell Programming

The interface between user processes and the core BerBell process uses pseudo-ttys on the Eighth Edition version and Internet Datagrams on the 4.3 BSD implementation. Thus one may simply echo commands to a named pseudo-tty or use a special echo which deals with Datagrams as required. As usual, examples of Shell code look awful.

4. Operators

At the dawn of the telephone business, boys were employed as switchboard operators. But,

> Boys did not last very long as operators. At the time they were often impatient, rude, and foulmouthed to the subscribers.[2]

Another set of C library functions exists for programming BerBell operations at a lower level. Programs written at this 'supervisory' level are termed 'operators' and are responsible for complete control of individual circuits in cooperation with the BerBell kernel program. While user level programs are unaware of system details, operators require some knowledge of the switching conventions and the hardware functionality. A program using these functions would receive hardware state changes from and issue commands directly to the circuit or circuits under its control over IPC channels.

It is likely that all call processing will be done by autonomous operator programs distributed over several machines. The advantages are clear in terms of processor utilization, flexibility, customization and prototyping. The operator concept has proven useful and rewarding, allowing a quick, permanent and elegant solution to several

unforeseen problems.

5. Software Descriptions

The BerBell software is made up of a number of permanently executing programs and processes that communicate through pipes and pseudo-ttys or Internet sockets. During the course of operation new short lived programs are invoked that implement specific features and services as described previously.

5.1. MSP Protocol Program

The switch used is called a Modular Switching Peripheral (MSP). It is described in detail in the Hardware section. The communications between the software system and the hardware switch is done over an RS232 serial port using the ANSI X3.28 protocol. This program was written at Bellcore in Navesink and trivial changes were made locally to convert it from UNIX System V to the Eighth Edition and 4.3 BSD. The program consist of four concurrently executed modules. The first is a parent program which opens the serial line and sets the appropriate options such as baud rate, parity, etc. It then sets up communication pipes and invokes the 'control', 'host' and 'msp' processes. Paraphrasing from the commented C program:

> The *control* process controls both the *host* and *msp* processes. It reads from its input and changes state and takes actions depending on the present state and the input. This process takes care of all procedural messages for granting master/slave status.
>
> The *host* process reads a string from its standard input to be sent to the device using the ANSI protocol. This process notifies the *control* process for permission to transmit. The *control* process notifies the *host* process when it is ready, and the *host* process transmits the message. The *host* process informs the *control* process regarding the success of the transfer.
>
> The *msp* process reads from the MSP and forwards any characters to the *control* process. It waits to receive a response from the *control* process before continuing.

The MSP protocol program is invoked from the core switching process.

5.2. The Core Program

The main, once monolithic, program is named 'bellerophon'. It sets up communication pipes and invokes a filtering program that processes all the textual output, a status program to which hardware state changes are sent, the protocol program as described above, and a DECtalk server program. These will be described in succeeding sections. Communication with these processes is implemented using operating system dependent macros. In the case of Eighth Edition a pseudo-tty is opened by *bellerophon*, the device name is linked to a name known to other programs thus implementing a 'named pipe'. Under 4.3 BSD a named socket is opened which receives Datagrams. These provide the input channel to *bellerophon.*

Bellerophon then initializes the hardware and data structures. There is a structure for each port and for each telephone number. The status process mentioned previously maintains the state of each port as represented in an internal data structure in a file. This file is read by an initialization routine to determine if the port ought to be initialized. It will not be initialized if it is not idle, thus calls in progress during a software reboot are not affected.

Bellerophon then enters its main loop gathering data from the MSP or from its input channel. Input from the MSP is parsed to identify the port involved. The current state is used to determine the function to be called. This function receives the port's data structure and any other data provided by the MSP (e.g., digits received) as arguments. In the case of data from the input channel, a function is called which acts on the input, a command and its arguments. A status reply is sent by writing to the device (or file) named in the arguments. Under Eighth Edition it is typically a pseudo-tty. In the 4.3 BSD implementation the recipient's address is part of the received message.

5.3. State Information Program

The program *statproc* maintains a file with the current state of each circuit. Each state change sends the circuit name and new state through a pipe to *statproc*. Other reported data includes dialed digits and circuit connection. As well as maintaining the state file, *statproc* also disseminates state information to other programs. By reading commands on its input stream in a similar manner to that in which *bellerophon* accepts its commands, *statproc* is informed of the appearance of clients who wish to receive status updates. Clients include programs that display system activity on bitmapped terminals, monitor the system's heartbeat (reported every three seconds), and maintain usage statistics in real time.

6. Hardware

The following sections describe the current hardware compliment, not the minimal or ideal configuration. Figure 1. is a schematic representation of the major hardware components.

6.1. Redcom Switches

The Modular Switching Peripheral (MSP) from Redcom Labs has proved a satisfactory piece of hardware for experimental applications. It provides the electrical interfaces required for telephone switching and does not interfere with the programmer's need to control its functionality.

6.2. Host Computers

For the sake of experimental diversity, two different system configurations are implemented.

6.2.1. VAX 11/750

The first system was implemented on the VAX 11/750 running UNIX Eighth Edition. This system supports approximately five full-time users with a general time sharing environment as well as the experimental telephone switch application. The VAX is configured with 4 megabytes of memory and 1.4 gigabytes of disk storage on 3 ra81s and 1 ra60. There are four DZ11s providing 32 serial lines. Thirteen of these lines are dedicated to BerBell, interfacing DECtalks, the Cytek switch, and the MSPs. The VAX is also equipped with the DSC-200 connected to the UNIBUS, an Interlan Ethernet controller, and miscellaneous other peripherals.

6.2.2. MicroVax II

The MicroVax is a relatively new addition to the experimental telephone laboratory. The operating system is 4.3 BSD. It is equipped with four megabytes of main memory and 140 megabytes of disk storage. There are 8 serial lines and an Ethernet controller. It is

connected to the single shelf experimental switch which is dependent upon the VAX BerBell system for access the DDD network.

6.3. Audio Devices

There are 11 DECtalks in use, 8 DTC03s and 3 DTC01s. The DTC03 is rack mounted and functionally superior to the DTC01 stand-alone models. The main functional improvement is the ability of the DTC03 to automatically terminate speech when a user presses a button on the telephone keypad. Providing this needed function in the host's software on the DTC01s was a somewhat exasperating exercise. DTC01s are still in use because they alone are equipped with terminal interfaces and audio output separate from the telephone interface. They are also more appropriate for touch-tone signaling of other devices such as the Watson.

Recording voice messages is an absolute necessity. Users expect at least the capabilities of a conventional answering machine in their sophisticated telephone environment. The system utilizes such answering machines principally as backup devices as well as the Watson, a sort of programmable multiuser answering machine and the DSC 200, a truly programmable audio record and playback unit.

6.4. Audio

There is an experimental audio lab accessible by BerBell. BerBell serves to provide access from the DDD network to this lab in order to demonstrate ongoing audio/music research.[3] It consists of a MIDI controlled studio of synthesizers and percussion machines. The MIDI host is a SUN 3/160.

There are also a number of other audio program sources attached to BerBell to give users a choice of entertainment on hold or otherwise.

7. Evolution

In April of 1985 an RS232-controlled telephone switch was obtained. The switch, manufactured by Redcom Laboratories, is described in detail in the Hardware section of this document. With the switch came two pieces of software: An ANSI X3.28 protocol handling program to support the low level communications between the switch and the host was written by colleagues at the Navesink Research and Engineering Center of Bellcore and remains in use today. A rudimentary call processing program, provided by the Network Architecture Research Division at Morristown, was useful in bootstrapping the system.

The initial application for the switch met two requirements. The first was to provide new and interesting services such as 'a better answering machine'.[4] The second was to reimplement services found in modern switches, but with enhancements making them more useful.

BerBell grew naturally and easily from the desire to provide better telephone service for an individual user to its current power and generality through circumstance and experience as well as vision. The requirement to switch telephone calls was initially fulfilled with a homemade switch consisting of a matrix of relays and a UART. The author originally built this device to permit several computer terminals to share half as many computer ports. When port selectors became available the switch was shelved, later to be resurrected as an ad hoc telephone switch, and subsequently to switch high voltages to control coin telephone functions. Its use as a telephone switch demonstrated the desirability of greater capacity for more interesting services. Serendipity provided a temporarily unused Redcom switch from another laboratory with which to experiment.

Its greater capacity suggested providing services to other people. Its wealth of functionality, the capability to act as a central office, suggested greater service possibilities. Along with the acquisition of a true telephone switch came the desire to design truly innovative switching features and services, not just a better answering machine.

A crude system was put together in a matter of weeks, providing basic enhanced services. The immediate desire was to correct the problems experienced with currently available services, such as the lack of flexibility and feedback from call forwarding functions. As such improvements were put in place, new offerings were invented and the software was designed so that new features and services could be implemented quickly. The feature challenge was popular for a period of time, demonstrating how easily a new idea could be implemented, tried and perhaps discarded.

Obtaining desired services and interfaces from other telephone companies was not a speedy process measured by do-it-yourself standards. Answer supervision, an arrangement which reports back to the originating switch when the called party answers, took over 18 months to be installed. Mechanisms to report the originating number to the terminating switch will not be universal for years to come. Interim techniques were employed pending installation or propagation of such services. Although these techniques provided less satisfactory results, they demonstrated the objective. For instance, when people were called without human intervention, a keypress was solicited in place of answer supervision. This allowed development of automated services to progress even under restricted conditions.

The initial system was somewhat unreliable. Service was frequently interrupted in order to install changes or demonstrate bugs. As more users came to depend on the telephone service, steps were taken to reduce outages. The first of these was the dynamic loading of structures that mapped telephone numbers to program names. This straightforward yet subtle tactic reduced downtime considerably because it obviated the need to recompile and reboot *bellerophon* each time a new service was introduced. Next, an independent program checked for the existence of the *bellerophon* process, reinvoking it if it was missing. This fell short of the desired goals, since it was not uncommon for *bellerophon* to hang. A more robust solution was implemented involving an audit primitive which caused *bellerophon* to issue a benign command to the MSP and receive a reply. The independent verifier issues this primitive and knows with relative reliability if the system is functioning. The next enhancement was to place a call from BerBell through the DDD network back to BerBell to verify the external interfaces. (One valuable lesson learned about the design of automatic auditors is not to sweep away evidence that is necessary for debugging. This mistake was made with the program described which killed the hung *bellerophon* process and invoked another. It was a mystery why the system occasionally rebooted itself, until the auditor was changed to produce a core dump of the hung program.)

Greater system reliability attracted more users. At first the mechanism for forwarding a subscriber's phone to a service involved the assignment of an additional number which terminated on the desired answering service program. This was costly in terms of dedicated telephone numbers. The solution, changing the forwarding algorithm to accept program names as well as telephone numbers was a simple change which supported the notion of flexibility as well as eliminating the extra forwarding step and the dedicated numbers.

BerBell began as a monolithic system. Services were implemented within *bellerophon* using a myriad of special structures. The command interface originally supported a

different command for each different service. The original idea was to generate interesting applications. The motivation to provide a clean and general interface came when other programmers wanted to write applications. The interface was defined empirically. A new application (call screening) was implemented with primitives that were defined as needed. Then one by one each application was rewritten using these primitives and defining new ones if necessary. It was rewarding to find that all but one of the primitives were defined by writing the call screening program. When all the services were changed to use the general interface and the old-style commands removed, the size of the command processing module was cut in half. Service application development proceeded independently of the switching program development. This started a trend of decentralization which was enhanced by the advent of *operator* primitives. New call processing algorithms are implemented by programs external to *bellerophon*.

8. Future Work

Decentralization will be pursued with the goal that all call processing be done by individual processes per port. *Bellerophon* will simply distribute messages received from the hardware to the appropriate processes and maintain a database to map extensions to processes. Processing modules will be distributed among processors communicating over networks.

There is interest in the artificial intelligence group to design parsers to process log files and create individual scenarios in English so that event paths can be easily identified and understood for the purpose of debugging or illustration. It will further be attempted to learn from these log scripts the events which lead to system errors and to predict and circumvent such conditions.

Another concrete goal is to provide administrative documentation and to package the system for distribution to interested parties.

9. Conclusion

The system described provides unconventional as well as conventional telephone service to a growing community of users. There is no limit in the potential for user programmable utilities. The next generation of consumers will require and demand to customize their products and services and they will have the educational background to do so. We expect that producers will abandon the current fad of featurism and provide value in terms of generality and well documented interface specifications. Service industries will provide end-user software as well as tools for do-it-yourselfers. BerBell has been, is, and will continue for the foreseeable future to be an extremely fun, captivating and rewarding project. To quote Louis Fyne, ***Like the song says, it's a scientific lifestyle.***

Acknowledgements

Credit goes to all the users of BerBell. Their bug reports and design input helped mold it. I am particularly indebted to Peg Schafer who lived with it, Peter Langston who wrote a number of fun programs that use it, Don Ford who wrote *ttweather* and *statproc*, Adam Buchsbaum who wrote BERPS, and Stu Feldman for his moral and technical support and the patient editing of this document.

References

2. *An Introduction to the Bell System,* pp. 2-4, AT&T Marketing Sales Administration, 1975.

3. Peter S. Langston, "(201) 644-2332 or Eedie & Eddie on the Wire: An Experiment in Music Generation," in *USENIX Association Summer Conference Proceedings*, pp. 19-27, Atlanta, 1986.

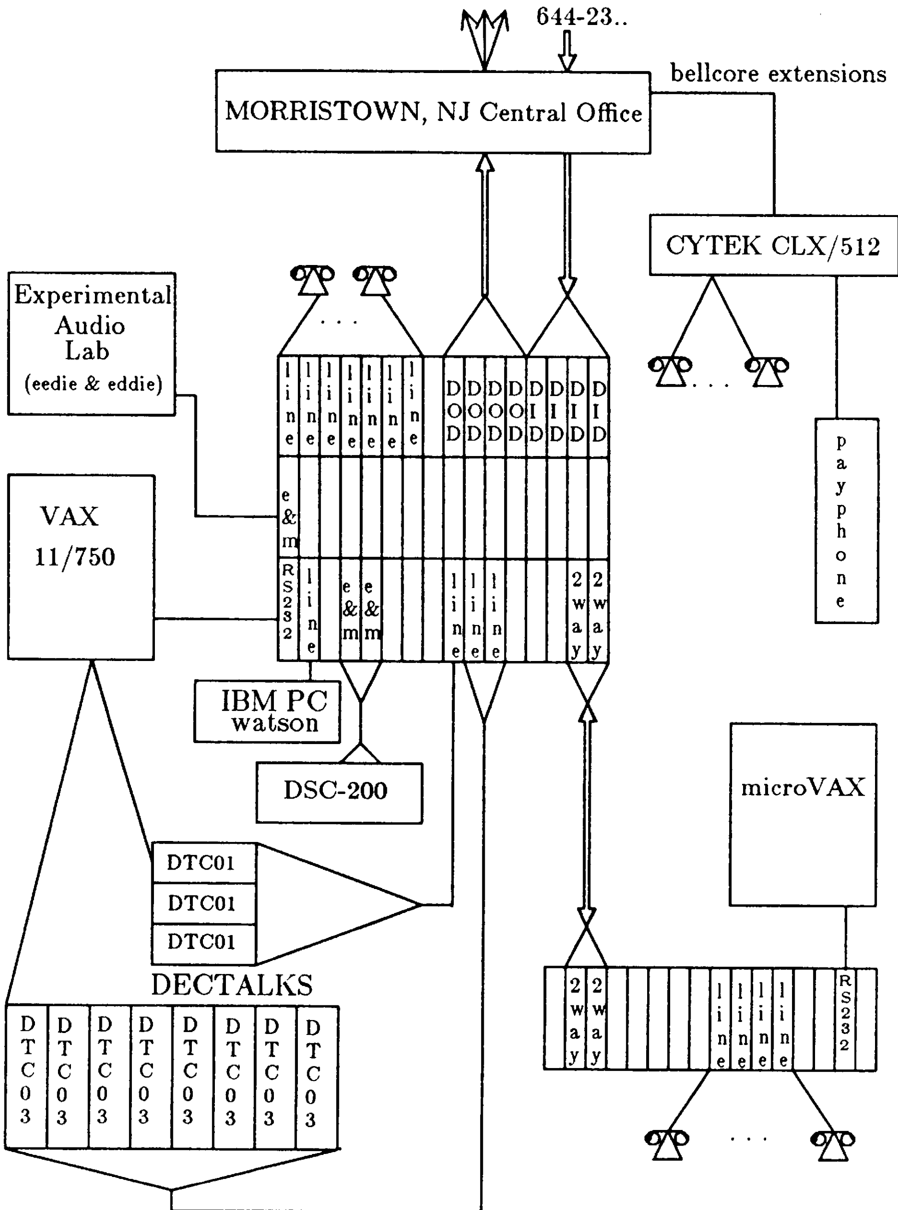


Figure 1. Skematic Diagram of Major Hardware Components.

UNIX in banking

R Grafendorfer

ABSTRACT

I want to present two ways of using UNIX for solving specific problems in banking automation.

The first one is about a very small UNIX lookalike for low cost 8 bit processors controlling a self service cash dispenser.

The second topic shows one way how UNIX machines can be used for solving the communication puzzle in a banking enviroment.

1. The Company

KEBA - Elektronikbau is an Austrian systems house and specialized on industrial automation and on banking automation. We have never solved any commercial EDP problems like transaction processing. The only things we produce for banks are self service machines like cash dispensers and self service safes.

About three years ago KEBA decided that UNIX and C are the only ways to a happy future.

2. UNIX for 8085

Three years ago KEBA developed a cash dispenser based on the very cheap 8085 microprocessor from INTEL. All of the application had been written in assembler. Everything was fine until the first customer came to buy one. When he wanted some minor changes the efforts for adapting the existing software soon exceeded the capabilities of the programmers and, of course, the available memory. When the second customer asked for changes the estimated efforts for the new adaption exeeded man years and the available hardware resources. So the project manager decided that something went wrong:

- The only change to sell the dispenser was to adapt it to the wishes of the customers. Otherwise the customer buys from the firm who sold the mainframe to him.
- The only change to earn money was to adapt it fast, otherwise the costs for changing the software would exeed the difference between our costs and the price the customer was willing to pay.
- The efforts for changing the software exeeded that difference several times.

Considering this facts we decided to analyze the problem anew. This time we fixed several requirements:

- A High level Language (C, of course)
- Command interfaces on meta level (yacc,lex)
- The capability to simulate the dispenser enviroment on UNIX

When we analyzed the application part of the problem we recognized that we had some very restricted requirements for a system interface:

- Only four tasks, running all the time.
- Very limited communication between them.
- Only raw I/O.
- No maths, no graphics, no specials.
- No mass storage.

So we decided to code a UNIX compatible layer in 8085 Assembler. All we needed was I/O (open, close, read, write, ioctl) System V msg control, strings and stdio. The drivers for the used interface chips existed already from an old UNIX port. This layer was finished in less than one man month. Next we implemented a dispenser simulation on a HP9000/550 mini running HP UX, a system V derivat. We also purchased a 8085 C cross compiler for this machine. Now we were able to split the team in two parts:

- The first part for analyzing the special wishes of all the customers and for prototyping on the 9000/550 together with the customer.
- The second part for cleaning the prototype of the first team and for porting it to the target.

By now we have already sold more than 300 cash dispensers and about every one of them has an own software release. The turnaround time for a new release was reduced to less than one week.

As we started to sell a lot of dispensers, soon our customers came and wanted them connected to the existing machines and networks, most of them to SNA. So we have reached the second part.

3. UNIX as translator

I guess I don't have to write about the strategies of some well known EDP equipement vendors. The effect is that most of the bankers are addicts to some more or less weird systems and networks and that the cannot get rid of them without collapsing. Today most of them are aware of that fact. So they don't want to have another strange system thats just compatible to itself. They try to use standards wherever they can. So, when we started discussions about interfacing different machines from different vendors together with the biggest banks from Switzerland, Germany and Austria one of the main dictions was portability of software. So we started discussing UNIX.

3.1. The facts

- Most of the nets in European banking are based on SNA.
- Because of the hierarichal structure of a SNA net each connected device must be known to the host and must have a corresponding session on a cluster controller or something equivalent.
- All of this software exists on old fashioned machines in old fashioned languages and is definitely not portable, but must be used.
- Most of this software is programmed for one type of machine (for example just for the NIXDORF Teller Machine) and cannot be changed without the vendor.

3.2. The requirements

- Portability.
- Different Machines from different vendors must be connected to an existing net via one device.
- A lot of interfaces: RS232, RS449, IEEE 488, IEEE 802.3, Manchester Encoding, X.25, SDLC,...
- The bank must be able to use this software on different machines.

Its obvious that the requested machine must be a UNIX machine. We decided for a VME based UNIX System. So we had connected the software standard with a hardware standard.

On this machine we are currently developing software which is able to compound a couple of machines from the same type (for example printers, cash dispensers, teller machines, etc...) on a couple of different interfaces to one logical machine for the existing net and make this one look like a machine for an existing application. So the bank has no need for changing software.

All this software is strictly based on the SVID and will be ported to p1003.1 as soon as the standard is fixed. All protocol dependent stuff is placed into the application.

4. Conclusion

You make bankers very happy if you sell them UNIX based applications. They have been addicted long enough.

A Digital Selective Calling System for Use in the Maritime Mobile Service (DSC)

Osmo Hamalainen

Posti- ja Telelaitos
Radio-osasto (Finnish PTT)
Helsinki, Finland

Markus Rosenstrom

Oy Penetron Ab
Espoo, Finland

ABSTRACT

This paper describes the finnish implementation of the maritime digital selective calling system specified by the CCIR and the IMO. The first phase network already offers all services specified, including automatic calls from ships to the public switched telephone network. The system is based on a central host computer running UNIX System V. The use of UNIX as both a developement and runtime environment resulted in a remarkably short developement time. The use of UNIX in a real-time environment has not caused major problems, due to careful system design.

1. Introduction

Morse code and voice are still the main calling mechanisms in maritime radio communication. The continuously increasing radio traffic and, consequently, an increasing demand for calling channels in all radio frequency bands (HF, MF and VHF) have made the disadvantages of these conventional calling methods even more evident. Aural watchkeeping has always been a difficult task, but today it is nearly impossible with simultaneous monitoring of 5 to 10 different radio channels. As a result of the increasing problems associated with these costly manual routines, an automated Digital Selective Calling system (DSC) has been specified by the CCIR (Recommendation 493-2) together with the International Maritime Organization (IMO).

The use of digital selective calling improves the technical performance of maritime radio traffic, one of the most important improvements being a decrease in radio frequency loading. In addition to these direct benefits, a DSC system also offers many possibilities to rationalize the call handling methods at the coast stations. The use of electronic call handling means considerable savings in employee costs, resulting in an increased profitability of the maritime mobile services. This is one of the main reasons why the Nordic countries intend to automate the handling of digital selective calls as far as possible.

2. DSC System Description

The DSC system is based on the use of separate calling channels and working channels. This implies that special calling channels are reserved for digital selective calls. A DSC call contains all necessary information about the subsequent working frequency or channel and the service wanted.

2.1. Transmission Method

The transmission principles of DSC calls is similar to the principles of radiotelex. The modulation method used is FSK (Frequency Shift Keying) modulation with a frequency shift of 170 Hz on HF and MF channels and 800 Hz on VHF channels. The data modulation rates are 100 baud and 1200 baud, respectively.

The data is made up of 10-bit symbols, consisting of 7 information bits and 3 parity bits, which indicate the horizontal parity of the information bits. Time diversity is provided, so that each symbol is transmitted twice, the time diversity interval being 400 ms in the HF/MF bands and 33 1/3 ms in the VHF band.

Every call starts with a phasing sequence, in most call types preceded by a synchronizing bit pattern. The vertical parity of the call is also computed and transmitted as the last symbol of the call. The data contents of the DSC call is of variable length and consists of several different fields, some of which determine the type of the call.

2.2. Call Types

The different DSC call types can be roughly divided into the following categories:

- Distress and distress related calls.
- Manual calls (ship or coast station).
- Automatic calls directly to the public switched telephone network.
- Special calls.

The calls in each category can be further divided into subcategories, which consist of related calls with one or more fields indicating the reason of the call or the desired type of communication over the working channel. As an example, a call of the basic distress subcategory includes the following fields:

- Self-identification (maritime mobile identity number).
- Nature of distress (fire, explosion, flooding etc).
- Distress coordinates.
- Time when the coordinates were valid.
- Type of subsequent communication (voice, morse, telex etc).

Other subcategories of distress calls are: distress acknowledgement calls, distress relay calls and manual calls related to a distress situation.

2.3. Special Processing of Calls

Because of the special nature of distress type calls, they are treated separately from all other types of calls. The national organisations responsible for handling distress situations may also be different from those taking care of normal maritime mobile radio- and telecommunication. Distress call information transfer has thus to be arranged between the DSC host computer and the national Search and Rescue Centre.

Another call type to be treated in a special way is the (semi)automatic VHF call. Calls of this type have a telephone number field, which makes it possible to automatically connect a calling ship to a public switched telephone network number. The DSC specification only supports automatic calls in this direction (ship to shore), although certain call types in category d (poll call, location poll call) can be used to support future automation also in the shore to ship direction.

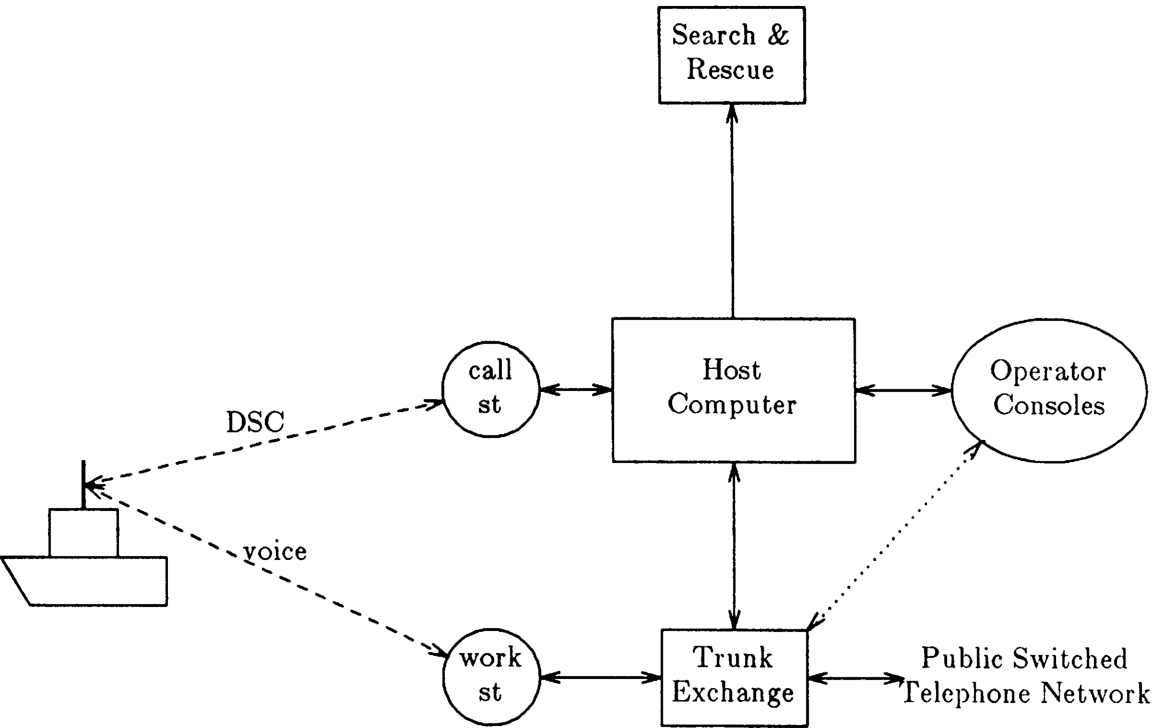


Figure 1.

3. Finnish DSC Network, First Phase Implementation

The architecture of the preoperational DSC system in the MF and VHF bands, partially covering the finnish coastal waters, is described in Figure 1. The system will consist of 4 VHF and 1 MF DSC base stations (call channels), using the same work channel base stations as the existing manual call network.

Figure 2 describes the main hardware and software components involved in the automatic processing of DSC calls. The following brief descriptions of the functions performed by the external components give a picture of the tasks left to the host computer.

3.1. Base Station Controller

The DSC base station controllers are the links between the fixed base station transceivers and the host computer. The controller hardware is based on an 8-bit microprocessor and the software is written in assembler. Each controller performs the following functions:

- Conversion of the incoming DSC call from an analogue FSK signal to a digital serial CCITT V.24 signal for the host computer.
- Conversion of the outgoing DSC call from a digital CCITT V.24 signal to an analogue FSK signal for transmission.
- Conversion of the relative field strength value of the incoming DSC carrier to a digital format. This information is sent to the host computer together with the base

station identification at the beginning of each call sequence.

- Buffering of outgoing calls if the base station is occupied (receiving or transmitting).
- Transmitter keying.

The controller and the host computer communicate over a 1200 bps asynchronous, full duplex serial channel using a specialised message type protocol.

3.2. Trunk Exchange Controller

The trunk exchange controller is the interface between the host computer, the trunk exchange and the work channel (voice) base stations (tranceivers). The controller is based on an 8-bit microprocessor and runs a program written in assembler.

From the host computer's view, the controller works as a digital parallel I/O device with some special capabilities such as number dialing and engaged tone generation. The actual functions performed in order to establish a connection between the telephone network subscriber (B-subscriber) and the ship station's voice channel (A subscriber) are:

- Transmitter keying.
- Engaged signal generation.
- Line connection.
- Number dialing.
- Work channel and trunk monitoring.

The trunk exchange controller and the host computer communicate over a full duplex, 9600 baud asynchronous serial line. The communication protocol used is based on messages with ACK/NAK handshaking.

3.3. Operator Consoles

Manual and distress DSC calls which need human interaction, e.g. call acknowledgement editing, are queued for display on the operator consoles. The operators can retrieve relevant information, such as the name and owner of the ship or available telecommunication facilities on board ship, from the local host computer database. This database is also used in the manual call radio traffic.

An operator console is either a normal asynchronous terminal or a Personal Computer running a special console application, allowing more sophisticated and individually tailored traffic functions to be performed locally. When properly designed, this distributed solution naturally reduces the load on the host computer.

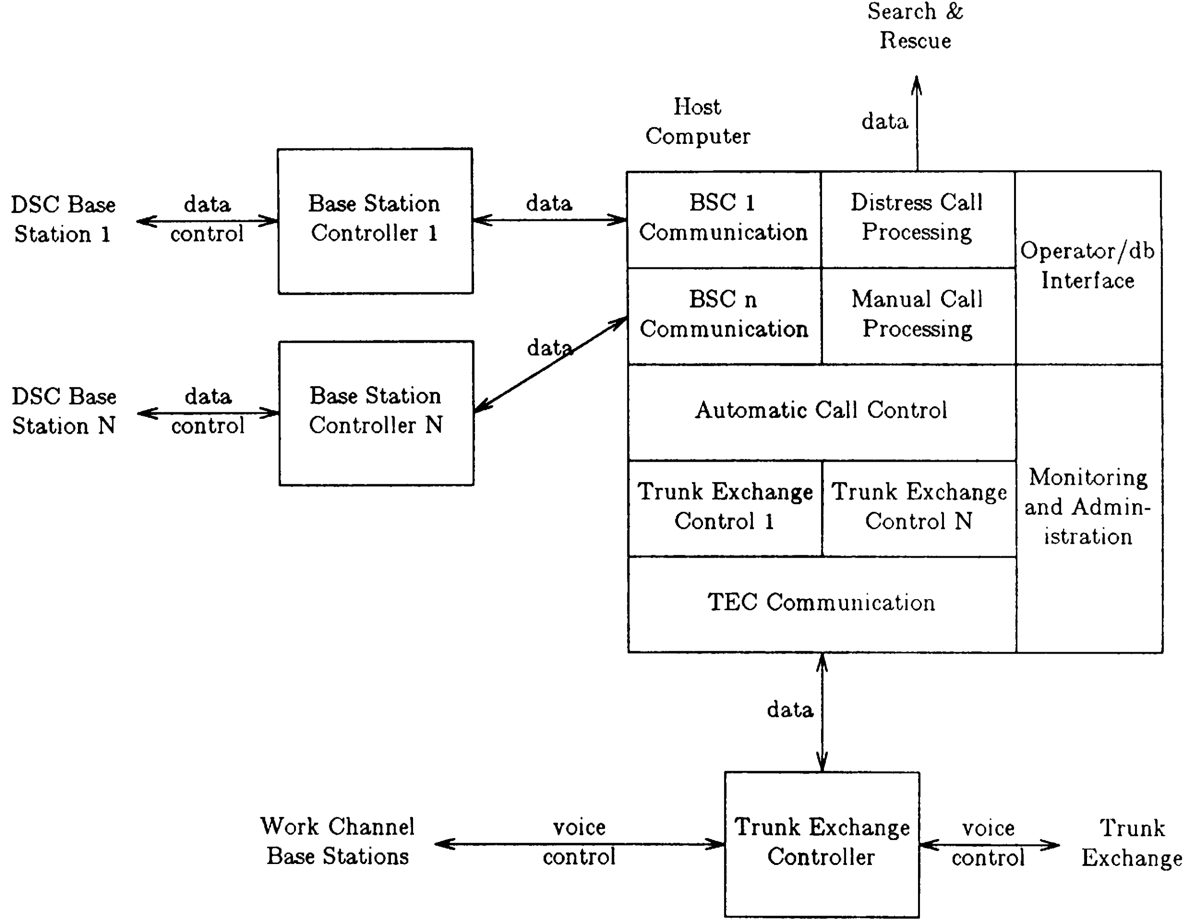


Figure 2.

4. The Host Computer

The DSC software running in the host computer performs the major share of the tasks in the system. This centralised approach was considered the appropriate choice for a first phase implementation, which, undoubtedly, would undergo a series of modifications before reaching its final state. Also the uniform programming/execution environment and the extensive set of programming tools offered by the UNIX system combined with the speed and versatility of the C language contributed to this solution.

A standard supermicro running UNIX System V.2 was chosen as the development and runtime environment. It was also specified that any further development would be done in a V.2 compatible environment, which made it possible to use the System V Interprocess Communication Utilities in the software design. This choice was important because of the timing requirements in the DSC system specification.

4.1. Software Components

The DSC software consists of the following main components (see also Figure 2):

- Base station controller (BSC) communication processes.
- Processes for preprocessing manual and distress calls.
- A set of operator interface mechanisms, e.g. a database server process.

- A main automatic call controlling process.
- Control processes for calls through the trunk exchange.
- Trunk exchange controller (TEC) communication processes.
- System monitoring and administration processes.

All processes, except some used for system monitoring and administration, are running continuously.

The interfaces between the different processes are mostly implemented with *message queues*, resulting in a fairly loosely coupled system. Some of the interfaces have been specified to enable future distribution of the system among several computers in a network (the message queues can easily be substituted with some other mechanism). In a few cases, global semaphores are used, e.g. for resource status indication.

As the processes are, in general, fairly simple and straightforward, only the more interesting parts will be described in greater detail.

4.1.1. BSC Communication

The communication with the base station controllers is based on a quite conventional asynchronous protocol, that consists of messages (framed DSC calls) and acknowledgements (a single character). The communication is full duplex and because of buffer space limitations in the controller, the required response time is about 2 to 3 seconds. If the other station doesn't answer in a specified time (adjustable), the message is retransmitted. The maximum number of retransmissions is also adjustable.

The host computer end is implemented completely at user level (no special device drivers or kernel modifications), in the form of two processes for each serial line. One of the processes reads the incoming characters, packages them into messages and routes the resulting DSC calls to the appropriate process, while the other process handles the outgoing messages and their acknowledgements. Both processes are implemented as *state machines* (infinite automata) with only one blocking system call. The two processes communicate with each other through the output message queue and by using signals. Signals are also used to control reinitialisation and shutdown of the whole DSC system.

4.1.2. Automatic Call Control

All incoming automatic calls and their acknowledgement messages are routed by the BSC communication processes to a message queue read by a single process. One reason for this centralisation is the possibility multipath reception of a call, ie. the same call may be received by several DSC base stations and propagate through the system as different calls, of which all but one must be eliminated at some common point. Another reason for processing all calls in one process is the use common resources (work channel, telephone lines).

The common process controls the flow of each incoming call until a free work channel and a line, with a controlling process, has been allocated or the call has been rejected. The process also controls the *optional ringback procedure*, which allows the host computer to enqueue incoming calls if no suitable work channel is available. The host computer then calls the ship back when a channel is available.

The somewhat complicated task of simultaneously controlling several incoming calls was solved by reserving a specific state machine (a data structure) for each active call. The input to this state machine is any event that matches certain characteristics of the particular call, e.g. the maritime mobile identity number (self id). When the control of

the call is transferred from the common process, the state machine of the call is released (its state set to 0).

4.1.3. Trunk Exchange Control

After allocating a suitable work channel and a telephone line, the common process transfers the control of the call to one of the trunk exchange control processes (one for each line). These processes perform the following functions:

- Transmitter control.
- Work channel carrier control.
- Exchange line allocation and control.
- Number dialling.
- Call accounting.

The control process is implemented as a state machine, which is fed with all events associated with the work channel and line used by the process. Most of these events are changes in the logic state of one of the 5 digital input signals connected to each base station and exchange line. As many of these signals must be monitored simultaneously, the input to the state machine is preprocessed by a *programmable preprocessor*, which automatically handles specified events. A timing feature is also included, which makes it possible to perform simple digital filtering of input signals. The major advantage of the preprocessor mechanism is a considerable reduction in the size of the state machine itself (from about 30 to 12 states).

5. Development Work Experiences

The DSC system was written in C using normal UNIX tools (make, SCCS, yacc, lex etc.). Part of the administration software is written as shell scripts. The total amount of time spent on the host computer software, not including the operator console interface, was about 7 man-months. This includes about 2 months of system planning. Most of the the programming work was done by a single programmer.

5.1. Performance

The first system was installed in a quite heavily loaded supermicro (Olivetti/AT&T 3B2/400), which made it possible to study the system's performance under heavy load. One of the more interesting experiences was the swapping out of one of the communication processes. This resulted in a series of (harmless) retransmissions and a longer response time than usual, but no data was lost.

It was also proved that the required response time for automatic calls (3 seconds) can be maintained even with fairly high system loads and normal process priorities. When the traffic density increases, it may, however, be necessary to increase the priority of some of the processes.

6. Future Directions

Currently, the whole DSC system runs on a single host computer, but when the system evolves, it could be distributed among several computers in a network. Some additional hardware redundancy will probably be used in the future, particularly for distress call handling.

More services will probably be added to the system in the future. One natural extension would be support for automatic calls in the shore to ship direction. This would, however,

require up-to-date information about the ships' locations in the host computer's database. This information could be collected at every contact with the ship and could be kept up-to-date by automatic position polling calls (already specified) at regular intervals. The use of polling calls should be very carefully planned, as it could easily overload the available call channels.

Using The UNIX Operating System to Market Selected Information

Alan Chantler BSc(Eng) FBCS

Department of Computer Science
Coventry Lanchester Polytechnic
Priory Street, Coventry, CV1 5FB
United Kingdom

Mail: alan@covpoly.uucp

ABSTRACT

This paper describes how a small business in the heart of England has developed a service based upon the use of UNIX and associated software. The service involves the collection, collation and publication of specialist information, which is vital to the building and allied trades. Subscribers to the service are able to retain a competitive edge over their rivals in the timely provision of all aspects of building services, ranging from site planning and management to final equipment and decoration of the building. A specially selected service is also available, which enables subscribers to home-in on relevant projects without difficulty.

The service is entirely based upon a number of computer systems, which are used to collect, collate, sort and select the information. The paper describes how the process of selection, purchasing and installation of the computer systems was completed. Some detail of the service supplied and the software which provides it is also given, including the use of laser technology printers, specailist document preparation and remote data collection facilities using PCs.

The paper includes a justification for the selection of a UNIX-based solution, and some warnings which should be heeded by anyone who intends to choose a computer system for a small business application.

Introduction

Within the United Kingdom the construction industry and allied trades market place has become very competitive. Large new developments are infrequent, especially glamourous (and therefore lucrative) projects, and many suppliers of materials and services find it difficult to locate projects which are suitable for the attentions of their sales staff. Much valuable time and money can be wasted in searching out new business.

As a consequence of these factors marketing and selling as a professional activity in the construction industry have received greater prominence. It is no longer enough to have a well known name, with a long history of fine quality workmanship and successfully completed projects, and rely upon people with work available to bring it to your attention.

Eighteen years ago a small group of astute people spotted a business opportunity which would enable them to provide an unique service to the building and allied trades. Since then their new business has grown until its annual turnover is well in excess of one million pounds.

The Business Opportunity

The methods of seeking business used by the construction industry and suppliers of building materials and services are many and various, including personal contacts, information gained from published sources, a well-established *Old Pals Network* and even spotting obvious signs of development on the ground. A significant problem highlighted by this approach is due to the fact that, by the time a development is obvious, most of the contracts will have already been placed. What was needed was an independent method of gaining advanced knowledge of proposed developments, so that effort could be placed in approaching likely prospects.

In the United Kingdom no building development can take place without the necessary planning permission having been granted by an appropriate planning authority. Since members of the general public are allowed to raise objections to proposed developments, on the grounds of unsuitability for example, all planning applications are placed into the public domain before being considered. In order to find out about any proposed developments then it is only necessary to pay regular visits to the planning office to study the applications. In Great Britain there are at least five hundred and thirty such offices!

From the foregoing it is clear that a market should exist for a service which can provide information about planning applications to the potential suppliers of building services and materials. It is of course essential that the information provided should be accurate, up to date and, if possible, available on a regional basis.

The Initial Solution

Once the idea was established a small business was set up, based in Basingstoke, Hampshire and in Elstree, North of London. The Head Office has since been relocated in Rugby, in the heart of England. Information about planning applications was gathered by a team of staff, who visited local planning offices on a regular basis. They soon developed a method of quickly noting down the salient details of planning applications. This information was then typed onto special paper using electric typewriters with a very small typeface. The typists were expected to decipher the information as they typed it!

Once the pages of information had been typed according to the special layout required they were proof read before being despatched to the printing works in Basingstoke. The pages were then printed and collated, a printed cover added and the whole lot stapled together. The resulting document was sold as a Bulletin of contract lead information. Recipients of the Bulletin were all subscribers who therefore knew that the information was not generally available. Thus subscribers to the Bulletin had a definite commercial advantage over their less aware competitors.

As demand for this service grew the business expanded into other regions of the country. In each region a number of collectors visited the planning offices on a regular basis. The information was typed up by local outworkers, usually housewives working in their own homes. The collectors would visit the outworkers regularly, to submit new information and to check work already typed. Once typed the information was sent to Basingstoke by mail, ready for printing.

In order to simplify the process of error correction a computer system was installed at Rugby for use, among other things, as a word processor. Once information was entered into the computer it could be updated if any changes were made either to the planning application or to the status of the development (eg. moving from being an application to permission being granted). The computer, which was running the CP/M operating system, was also used to maintain the list of subscribers and a list of collecting agents.

Due to the need to sell information to national companies, whose business representatives cover the whole of England, Scotland and Wales, it was necessary to expand. Owing to the success of the sales and marketing team of the new company (who also organise and present marketing seminars) the Bulletin business grew to the extent of covering thirteen regions of the United Kingdom. Each region had its own Bulletin printed once every three weeks. This has meant processing the Bulletins at a rate of one per day, leaving two days for contingencies. In order to save some time, and because not all of the information changed within the three week cycle time, new editions of a Bulletin were produced by *cut-and-stick* operations being performed upon previous editions.

Clearly if the business was to even survive, let alone grow, something had to be done to simplify the production of the Bulletins.

Planning the New Approach

The Managing Director of the company was already convinced that the answer lay in the proper application of information technology. Having started his business with a *shoe-box* filing system he had already taken on board the use of a computer based system, both for accounting purposes and for word processing. Members of his staff, whose knowledge and interest in computing had grown alongside his, were asked to investigate the possibility of using a computer to manage more of the Bulletin production processes. Not knowing of any better way of going about this they contacted a number of potential suppliers to ask for assistance.

In common with many small businesses finding themselves in a similar position, the company did not have direct access to sufficient expertise to specify suitable systems, either software or hardware, but had a very good idea of their functional requirements. The lack of technical detail was, however, tending to limit their view of what might be possible in the way of computerisation or even automation of their business practices. A few of the equipment salesmen had given them general statements of availability of suitable systems but many of them were unable to adopt a sufficiently detached view of the business requirements, relying instead on their own understanding of the capabilities of their wares. In particular it was noticeable that none of the potential suppliers contacted had much idea about the capabilities of or even the availability of suitable software. This apparent lack of detailed knowledge was stifling the proper development of a strategic plan by means of which the business opportunity could be exploited.

At this point a chance encounter led to the involvement of an independent consultant, whose widespread knowledge and expertise was complemented by an ability rapidly to assimilate the details of the proposed business development. An outline proposal was drawn up and discussed in detail with the management and staff of the company.

The main points of the proposed new system were as follows:

(a) collectors would work as before, visiting the Planning Offices on a regular basis.

(b) information collected would be entered into a micro-computer system by the outworker, using some suitable software which would enable local editing and printing.

(c) local printing and checking of the data would be undertaken by the outworkers.

(d) the data would be transmitted to the Rugby office where it would be entered to a central database. Here the data would be sorted, edited if necessary and finally printed in the form of masters for reproduction, for onward processing at the Basingstoke printing works.

(e) the customer database would be maintained at the Rugby office; the outworkers would each keep a record of contacts (known as agents).

In addition to this computerisation of the existing business, it was proposed to develop and offer a selective service whereby the subscriber would specify particular classes of information which he required. The use of a suitable database system at Rugby should enable the automatic selection of such information, based on any of a number of selection criteria.

Obviously, the key to such a system would be the availability of suitable software. Ideally a fully integrated system was required, which would enable the processing of large amounts of information. At the time (summer 1985) there was nothing suitable available which could be expected to operate on both small micro-computers used by the outworkers and a much larger central system. It was also noticeable that there were no truly portable systems available, and that the purchase of an *off-the-peg* system would render any local adaptation virtually impossible. Thus the company could either redesign its business to match available technical solutions, enter into a long lead time associated with the specification, design and implementation of a fully bespoke system or rethink its business plan altogether.

Fortunately the consultant had extensive experience with the UNIX operating system, having held a licence since 1978. He was able to produce a prototype (mock-up) system, using available software utilities such as *grep, awk, sort* etc., which was sufficient to demonstrate to the company management two main points: firstly that their plans were feasible and secondly that a solution could be formulated based upon the integration of existing and new software provided that this software was being run within a sensibly designed operating system; a system which had been designed to support the ideas of cooperating software processes. Clearly the only sensible system was going to be UNIX but, at this time, there were not many commercial applications being run under UNIX, especially not in small businesses. It would be necessary, therefore, to persuade the management of the company not only that the selection of UNIX was the right thing for

their business plan but also that the fact that not many other companies of their size had so far made such a decision would not mitigate against success.

As stated earlier, the managing director was already aware of the potential benefits of information technology. During the summer of 1985 he had read a one page series of articles in the Financial Times extolling the potential virtues of the UNIX operating system based upon the emerging 16/32 bit microtechnology. This meant that he was already receptive to the consultant's suggestion that a UNIX based solution was the most feasible. Once that decision had been made it was comparatively simple to put together an outline specification of software and hardware requirements†, and eventually a plan was evolved which would involve the installation of a UNIX system at Rugby to support the database and to produce the print masters, another similar system at Basingstoke (the printing works) which would be used to support management functions, including job costing and scheduling, stock control and sales order processing, and the purchase of a number of micro computers for use by the outworkers in their own homes.

Selection of Software

Although there is a growing number of applications programs available for running within the UNIX environment the choice is still not very wide. Clearly any local requirements could be met by the production of bespoke programs, often preceded by the rapid development of a prototype system as before. The key problem appeared to be the database system for storing the details of the planning applications. What was required was a free text storage system which would enable searches to be made on keywords which would not necessarily be held in predictable fields within a record. Such a system, called *<<STRIX>>*, had been developed a year or two previously and was about to be launched on the market. Early contact was made with the author of this system who, it transpired, was in the throes of adapting his software to run in a general UNIX environment. From the advanced promotional material it was apparent that this software would meet the requirements of both the current proposals and the foreseeable future developments. A particularly attractive feature of the software was its ability to optimise the usage of disc space without any compromise of accessibility of information. In particular it was noted that a record within the free text database could contain any number of bytes (characters) from one to the available disc size, that records did not all have to be the same size and that the database could be searched extremely quickly using any text as a key. It was also noted that data could be added from a batch file as well as via the keyboard in interactive mode. This latter point would turn out to be very important when deciding how to collect the data from the outworkers.

The original specification of this database system included only simple report generation capabilities of a fairly general nature. The business proposal required that the information, once selected from the database, could be printed in the same format as had been used by the manual system. This requirement was due to the perceived conservatism of existing customers, together with the obvious fact that the current format was what customers wanted and would therefore sell. Because the software was being developed under UNIX, the capabilities of which were well understood by all parties concerned, it was possible to specify and produce a suitable post-processor program which would take the output from the *<<STRIX>>* database selection process and format it for the printing stage. Experience has shown that this kind of *do-it-yourself* integration is much easier within UNIX based systems than elsewhere, owing to the regular nature of input

† particular care had to be taken in choosing disc configurations

output and file handling.

The choice of software for use by the outworkers was not quite such an easy decision. The only sensible choice of a micro-computer operating system lay between CP/M and MSDOS. Since the company already had some experience of CP/M this was their initial choice. However, subsequent experience with a prototype system written in BASIC showed up both the inadequacy of the operating system and the unsuitability of BASIC for a system of this size. At the time there was no BASIC compiler available for a CP/M based system. The choice was further complicated by the fact that the equipment used by the outworkers had to be of a compact and portable design, yet capable of supporting quite large data files (3 to 4 Mbytes each in some cases).

Eventually it was decided that suitable software would have to be written to provide the editing and storage capabilities required by the outworkers. Since speed of operation and efficient use of any disc space were of paramount importance, it was decided that this software should be written in the C programming language. It has to be admitted that this decision was influenced by the choice of UNIX for the main operating system, as well as by the preferences of the chosen consultant. It should be noted, however, that the choice of C for the micro-computer software meant that much of the development could be carried out on the main UNIX system, making full use of the excellent tools available.

The only software problem not yet addressed was concerned with the method of transfer of data between the micro-computers and the main UNIX system. The communications software available for non UNIX systems at the time was very limited. In particular there was very little available for the sensible transfer of files between systems which were running different operating systems. Then along came KERMIT! The availability of KERMIT, which has two† very significant advantages over all of its rivals, meant that the problem of file transfer had been solved.

And so the development of the software could go ahead, secure in the knowledge that, because the chosen system would be based on UNIX, the software would be transportable to whatever hardware was eventually installed.

Selection of Equipment

It soon became apparent that, of the number of potential suppliers of UNIX based micro-computer systems, very few had any detailed knowledge of the capabilities of either hardware or software. Many companies appeared to be jumping on the UNIX/MC68000 bandwagon and the only serious offer of applications software was an integrated office automation system. There were no offers of free text databases or specialised report formatting programs.

The managing Director of the company had already spent some time visiting exhibitions and talking to sales staff. He was becoming overwhelmed with promises of what might be possible. Some suppliers were even able to offer demonstrations of UNIX based systems, but these demonstrations were often less than satisfactory, owing to the inexperience of the staff directing them. Detailed technical queries, concerning such things as disc capacities and speeds, communications capabilities, etc., were often met with blank looks. There were many offers of systems onto which UNIX was apparently being ported, either *now* or *soon*. Particular difficulty was experienced when trying to determine the support capabilities of proffered systems, especially in terms of the number of simultaneous active users.

† firstly it works and secondly it is free

Eventually a short list was drawn up from which a final choice was made. The chosen supplier was marketing a British made system, which was important. The system ran a version of XENIX and should have supported a maximum of eight simultaneous users. A decision was made to order two systems, one each for Rugby and Basingstoke, and steps were made to enable the <<*STRIX*>> text database system to be installed on the system before and during delivery.

At about the same time the company was introduced to the desk top laser printer. Initial trials of this device were so impressive that it was decided to base all further development of the Bulletin system on the use of a number of such printers. The author of the report formatting program was also becoming familiar with this technology and so was able to adapt his software to make full use of its capabilities. Hence it was possible to plan for the production of camera ready copy directly from the computer system, thus saving at least one operation later on.

Installation Problems

When the equipment was installed at Rugby a number of problems became apparent. The performance of the system was far below that expected and was, in fact, totally inadequate for the purpose.

The system used some non-standard interfacing and so proper connexion of terminals was not supported in all cases. In particular the system console could not be used as a log-in terminal! The reason for this appeared to be that the system had not been designed with UNIX in mind but the manufacturer had been forced by the market to arrange for a UNIX port , without really understanding the implications. Unfortunately these difficulties were exacerbated by the totally inadequate support staff available from the supplier.

In order to evaluate the system an attempt was made to begin the build up of a database of potential subscriber names and addresses. Owing to the almost total lack of any software support for this activity a prototype system was developed, based on a number of shell scripts. This proved to be perfectly adequate for the purpose of familiarisation and a large number of records were created. Some difficulties were experienced with the disc drives and these were made much more serious by the fact that the cartridge based backup system did not work at all; it went through the motions but there was no data available on the backup cartridge. At about the same time a standard half-inch magnetic tape was purchased, containing the names and addresses of approximately 25,000 potential customers. It then transpired that the supplier of the computer system was unable to read such a tape since he could not attach a suitable magnetic tape drive to his hardware!

At this point it was decided to give up with this particular system and return it to the supplier. It was then discovered that the version of *tar* provided was totally non-standard and that there was therefore no means available of transferring the existing data from the rejected system onto any new system. The only exchangeable magnetic medium available was floppy disc, which used a totally non-standard format the details of which were unknown by the equipment supplier.

Fortunately it was possible to return the equipment and to recover most of the money. It is interesting to note that the supplier is no longer in business selling UNIX systems.

A new system had now to be obtained. This was chosen with the aid of the consultant who, together with the developer of the text database software, was able to select an adequate, well supported (and British made) system without much difficulty. The manufacturer of this system was so interested in the application that he entered into various agreements with the company and the supplier of the <<*STRIX*>> text

database, to ensure that all went well and that he would be able to offer the text database to his other customers.

The newly chosen system provided a version of UNIX called UNIPLUS+, which is based upon AT&T System V. The <<*STRIX*>> text database software was readily transferred to this system, as was the software for formatting the reports for the laser printer. The availability of a standard magnetic tape unit, for use as backup and transfer of remote data, was an added advantage.

Remote Data Collection

It had already been decided to provide each of the outworkers with a micro-computer for the purpose of inputting, editing and printing the data. They would also use the system to maintain a database of agents names and addresses. Hitherto the outworkers had been using typewriters provided by the company. They were therefore familiar with some keyboard skills but had no experience of information technology. Clearly the chosen equipment must satisfy a number of criteria:

(a) it must be adequate for the purpose, including the capability of storing large data files in a secure fashion.

(b) it must be easy to set up and operate.

(c) it must be compact and if possible capable of being packed away when not in use.

(d) it must be supported by a national network of support staff since down-time would prove very disabling to the production of the Bulletins.

(e) it must be capable of running the chosen software.

(f) it must be easy to learn to use.

After a couple of abortive attempts to make progress with CP/M based portables (KAYPRO and WREN), and realising the potential for a growth in the functionality of the outworkers, a decision was made to purchase a number of Olivetti M21 portable computers. This is a very compact system which was available with a choice of either two floppy drives or a single floppy and a hard disc of some 20 Mbytes capacity. It runs MSDOS and is completely compatible with the IBM PC. Unfortunately the model has since been discontinued so a number of the outworkers have been equipped with the M24 instead. This is functionally identical to the M21 but is much larger and not portable.

Software was developed by the consultant to provide a user-friendly system for the creation, maintenance and printing of a free text database. As the software was being developed so the users recognised the possibilities of further enhancements. Consequently the software has displayed a *topsy*-like characteristic and plans are already afoot to completely rewrite it†. The software is written in C and the compiler used was the

† It may, in fact, be replaced by <<*MINISTRIX*>>, a PC based version of <<*STRIX*>>.

Lattice C Compiler. Currently the software runs to about 4,000 lines of code in 50 source files. It was developed on a PC, with a copy being maintained on the UNIX system for printing and checking with lint. The availability of KERMIT for both the PC's and the UNIX system has made the transfer of files extremely reliable.

The software provides the capability of specifying the record structure of a text database, creating and modifying data using a purpose built screen editor, printing selected records for checking and creating files on floppy disc in a format suitable for transferring to the UNIX system for use as batch input files for the <<*STRIX*>> free text database system. The software is entirely menu driven and is designed for use by non-computerate people. The outworkers have taken readily to its introduction. It is used both for collecting the planning information and for the maintenance of the agents database.

Once a set of newly collected data has been successfully entered into the PC a copy is sent, on floppy disk, to Rugby by mail. There it is transferred to the UNIX system via another PC running KERMIT. The free text database is then updated.

Updates to the software, arising out of requests for new facilities, can be distributed via the mail. It is not yet thought to be economic to equip outworkers with modem equipment to enable them to access the UNIX system directly but this would obviously be possible in future. All outworkers have a copy of KERMIT installed on their machines.

The recent arrival on the scene of Tannenbaum's MINIX operating system has introduced the possibility of running all of the computers in a UNIX-like environment. No decision has yet been taken on this but it must be seen as a definite possibility for the future.

Producing the Bulletin

Once the data for a given region has been updated a copy of the regional Bulletin is produced. This is done in two main stages. Firstly the selection facilities of the free text database system are used to produce a list of the items to be included in the particular issue under production. This is done by using the issue number field as selector. Then, by running a specially written program, this list is formatted for printing on a laser printer. A variety of fonts is used, together with special layout features, to produce an acceptable document. Information is presented by planning authority within county within region. The output from the laser printer is mailed to Basingstoke† (or sent by courier) where it is used to create the necessary printing masters.

Once printed the Bulletins are collated and bound, before being dispatched direct to the customer. The necessary mailing labels are produced at Rugby and sent electronically to Basingstoke.

Using the computer system to prepare, sort and select the information has not meant that the time taken to produce each Bulletin has been reduced. In fact the process takes slightly longer but this apparent disadvantage is more than offset by the advantages of being able to select information before printing.

Selected Information

Many of the existing subscribers to the service have expressed the need for selected information. Rather than spend their time searching through several regional Bulletins looking for, say, contracts involving the installation of lifts they would be happy to pay for the provision of a list of just those contracts which matched their interests. To meet this demand it was decided to offer a tailored leads service.

† In fact *email* is already used for this purpose

Subscribers to this new service would be able to specify their own selection criteria, and would expect to receive regular updates of appropriate information. In addition they might expect to be able to request such selective information on an *ad hoc* basis.

In order to gain experience in this new tailored leads service Bulletins were manually searched for interesting items, which were then cut out and photocopied onto special Action Sheets. This process was very long winded and labour intensive, but did serve to prove that a market for such a service existed and could be sustained. What was needed was a machine based system to provide this service.

The system already in use for the production of the regional Bulletins could be readily adapted to meet this need. The <<*STRIX*>> text database system enables a searching strategy file to be created and run against a database, producing a list of just those records which match the search criteria. The resulting list can then be printed onto special stationery, and creates a good impression of the information having been specially prepared for that customer.

This new service has taken off remarkably quickly. It would not have been possible without the initial development of the system within the UNIX environment, which has enabled new software to be integrated within the system with consummate ease. The possibility of providing direct access to this selected information via dial-up lines is currently under active consideration. It is likely that such a development would involve marketing a complete turnkey system, consisting of all of the necessary hardware and software to enable a customer to access a suitable computer system. Again it is known that much of the necessary software is already available for running under UNIX.

Communications

The company has for a number of years made use of a privately owned internal telephone system at each of its offices. This was essential to maintain close contact between the sales staff in Basingstoke and the Bulletin production staff in Rugby.

During the saga of equipment selection, installation, removal and installation, some decisions were made concerning communications between Rugby and Basingstoke. These were to have an impact upon further business developments, as outlined earlier. In particular a British Telecom high speed data link was installed, together with some electronic switching equipment at either end. This made it possible for users of the computer at Rugby to make direct contact with the similar system at Basingstoke from their own terminals. It also opened up the possibility of transmitting the finished Bulletins electronically to the printing works.

Electronic mailing is becoming widely accepted within the company. It is particularly valuable for maintaining the necessary contacts within an organisation whose personnel are often out of the office. Questions about the operation of the computer system raised in one office can often be answered by mailing the requisite software fix directly in machine readable form, thus reducing the possibility of error. Larger data transfers are managed by sending magnetic tapes via a courier service or with one of the many inter-site travellers.

Lessons Learned

A number of valuable lessons were learned during the development of this system.

Firstly it is apparent that there are small business applications for which the UNIX operating system is ideal. This is especially true when there are specialised software needs, particularly involving the use of filters. The availability of relatively cheap and

powerful hardware, together with the growing number of people with the appropriate level of expertise, means that no-one should be afraid to choose a UNIX based solution. It also became apparent that the almost total lack of availability of suitable software should not interfere with the development of new business opportunities.

Secondly, and rather sadly, it became apparent that there are many people trying to sell UNIX based systems who do not fully understand the underlying philosophy of the operating system's development. Consequently they tend to overestimate the capabilities of their products and to underestimate the problems of adapting non-UNIX software to the environment. This problem will correct itself as such suppliers will have to *wise up or give up*. It is also worth noting that the general level of awareness of potential customers is rising all the time, particularly now that there is a system which is worth learning about because it is here to stay.

Thirdly it was demonstrated that, by choosing UNIX as the operating system, true hardware vendor independence was possible. Not only was it possible to switch machines in mid stream it will also be possible to add new machines, from any source, as and when needed without having to re-write software. This will enable the small businessman to take proper advantage of the marketplace when selecting new equipment. It should also sharpen up the vendors.

Fourthly it was demonstrated that a number of professionals could cooperate on a joint venture provided that they had a common basis from which to work and through which to communicate. This basis was expertise with UNIX. Not only, then, does software become hardware independent, the same is true of professional personnel. This must lead to a better quality of advice and assistance, provided that the consultants involved continue to maintain their independence.

Final Remarks

The company continues to thrive. It is able to offer its customers the service which they require and can continue to do so for the foreseeable future. Already the printing capacity of three desk top laser printers has been exceeded in one office and a new, much larger unit has been installed. The possibility of offering a direct mailing service to industry-based groups is being actively explored, based upon the large number of names already stored and the capability of selecting and printing at will.

The high quality of the information provided, as well as its presentation, has led many customers to subscribe to several additional regional Bulletins.

It is the considered opinion of all involved in this venture that it would not have been possible without the use of the UNIX operating system.

Acknowledgements and Thanks

Thanks are due to a number of people for the opportunity to produce this paper: to John and Phil for their continued faith in and enthusiasm for UNIX, to Tony for producing some useful software, to Ken and Denis for having thought up the idea of UNIX in the first place and to Margie for things not much connected with computing at all.

Contact Addresses

<<*STRIX*>> and <<*MINISTRIX*>> are available from:

Tony Kent,
Microbel,
The Loft,
Lord Nelson Yard,
Main Street,
Sutton-on-Trent,
Newark,
Nottinghamshire,
NG23 6PF,
United Kingdom,

Telephone (+44) 636 821722

The company referred to othroughut the paper can be contacted via:

John Rogers,
Managing Director,
Contract Leads (Tailored Services) Ltd,
Temple Buildings,
Railway Terrace,
Rugby,
Warwickshire,
CV21 3EJ,
United Kingdom,

Telephone (+44) 788 67816

The consultant was:

QSOFT
The Old Bakehouse,
Yelvertoft,
Northampton,
NN6 7LF
United Kingdom

Telephone: (+44) 788 822964

Is there a Future for UNIX in the World of Commercial Computing ?

Philip H Dorn

Dorn Computer Consultants, Inc
25 East 86th Street
New York
NY 10028
U.S.A.

ABSTRACT

UNIX has reached a crucial stage in its evolution. After many years of drifting through the scientific and engineering communities looking for a home, a snug harbour, UNIX has been agreed upon as the operating system of choice for support of various special purpose systems (CAD/CAM/CAE, engineering workstations, high resolution graphics), software development facilities and some types of small multi user systems.

Where UNIX has been unable to make much headway is in the large scal, high volume, DBMS, transaction processing, heavy communications applications which are the stock in trade of commercial installations.

UNIX, the operating system, and the UNIX community, a very broad term covering training, support, documentation, DBMSs, and applications programs, have failed to impress those who make the decisions in major installtions. In the commercial world, UNIX is thought of as a toy system for amateurs and diletantes, far too insecure to be entrusted with really important data files, and virtually unknown among those who work in these installations. Perhaps even more devastating, installation managers are not convinced the highly touted UNIX portability really works in a world with complex, multi layered applications software systems.

Many of these issues do not really address the questions of UNIX functionality or performance but rather deal with public perceptions and the UNIX infra-structure. This paper will discuss some of the reasons UNIX has been unable to make much headway penetrating major commercial computing markets.

Introduction

Any paper worthy of presentation at a conference must have a point of view and represent the thinking within a reasonably large sized group of computer using installations. The individual speaker cannot help but reflect the views of a constituency. Few individuals are smart enough to honestly represent the entire data processing community.

This paper comes from a background of many years experience with very large scale installations running a variety of main frames. These shops have a number of elements in

common. A list of the most pertinent characteristics of the critical applications running at these centres includes the following:

1. High volume, large database problems.
2. Heavy communications load.
3. Mandated security/privacy requirements.
4. Close relationship to existing applications.
5. Emphasis on accessibility and reliability.
6. Use of generally understood languages and tools.

The problems tend to lend themselves to monolithic solutions with little divisibility. For example, master policy update (insurance), demand deposit accounting (banking), granting of individual or corporate credit, or senior management payroll.

This class of installation tends to spend between 65-70% of its software budget performing maintenance on its large scale application systems. The remaining money generally is spent developing a mix of add ons to existing systems and on rare occasions, a completely new system. While installations with these attributes develop a vast amount of software, when reference is made to "software development" or "software developers" the terms are not intended to reflect this class of work.

This brief paper addresses some critical questions of the present and future status of UNIX with this constituency. It is fair to reveal the biases of the author, admittedly not known to be an enthusiatic supporter of UNIX or any other operating system excepting the one personally written in 1961-1962. In grander scope of things, what really counts is keeping the stability of the applications programs. In the view of at least one veteran observer, hardware and operating system are little more than necessary overhead, budgetary negatives which, were it possible, would be avoided at all costs. Conversely, it is the applications upon which the corporations who are paying the bills for computing (hope to) earn a profit. Therefore, the applications are serious business. Operating systems are merely rather interesting curiosities about which academics argue. A few may wish to quarrel with this point of view.

Major Operating Systems

The world is full of operating systems. In reality, only a small number have achieved sufficiently wide dissemination to have earned prolonged discussion. This says nothing at all about the quality of the individual system, many have high reliability and broad functionality. However, in the marketplace this is less important than the attractiveness of the base machine and the marketing skills of the vendors.

The heavy hitters list is very short. Not surprisingly, IBM leads off with its MVS and VM pair. DEC's VAX/VMS is very widely used. Microsoft's MS-DOS has more licencees than all the others combined.

Why are the others not on this short list ? There are several explanations including: too lightly used, only runs on a single processor, minimum functionality, obsolete, or has been restricted to machines of little influence.

Multics was a first rate piece of work but has been doomed by Honeywell's termination of the hardware on which it resided. What the future holds for GCOS, now under Bull's control, and the Unisys (nee Burroughs) MCP is anybody's guess. DRI's Concurrent DOS looks good but has not been successful in the marketplace. IBM's DOS is heavily used but is of little real interest except to those who still rely upon it continuing to be available.

Where in the spectrum of current systems does UNIX fall ? The marketplace has already clearly established the niches which are open and available to UNIX. whether its supporters or detractors agree, most customers have made their choices.

The territory in which UNIX seems to have proven itself the operating system of choice is bounded by the following:

1. Special purpose single user systems, especially for CAD/CAM/CAE.
2. Small multi-user systems supporting mainly local users.
3. Heavy developmental work load.

Where UNIX does not seem to be making any serious progress is in the following categories:

1. Single user of the conventional "Personal Computer" variety.
2. Large scale, multi-user systems with heavy communications loads.
3. High volume production systems.

Of the major operating systems, MS-DOS continues to dominate low end PC applications, VAX/VMS is supreme in the multi-user world and IBM's MVS/XA leads the way in high volume, heavy production installations. While from time to time each will infringe on the other's "turf" and VM nibbles at the three of them, by and large the boundries hold up.

The question is not so much which categories are dominated by UNIX, MVS/XA or GCOS but why one particular system is able to establish itself in a niche.

Apparently, domination stems directly from fit. DEC's VAX/VMS has a wide range of consumers from the very small VAXmate up to the newer 25/50 MIPS clusters of 4/8 DEC 8700s. In theory, this is a natural arena in which UNIX ought to be successful. UNIX was born on DEC machines. UNIX has the same attribute of not being size dependent which DEC promotes so vigorously. Why is DEC's product so thoroughly in charge ? Measurement of the UNIX-using component is a matter of small disagreements about a percentage point of two. The best guess is no more than 3% of the larger VAX systems are operating UNIX while at the low end of the VAX series the UNIX or Ultrix penetration may be as high as 6%.

Without any attempt to be definitive, the answers to the DEC puzzle are a combination of factors. In no special order, the elements include: quality and performance of VAX/VMS, functionality supported within VAX/VMS, number and broad availability of programmers trained under VAX/VMS, amount of software on the market for various utility functions (DBMS, 4GL, graphics), pressure from DEC marketing force, better responsiveness from DEC to reported software problems when the installation is running VAX/VMS, and tight relationship between VAX/VMS and DEC hardware.

The Current view

UNIX is not a major factor within the defined constituency. Most large scale shops in North America consider UNIX merely one of the many interesting, researchy developments which need to be watched and tracked because in the future they might be important.

Not only do large installations not use use UNIX on their primary production machines, UNIX is not really involved when software systems are being built in house. Neither are the independent software vendors who create packaged software bought by large corporations UNIX users. UNIX is advertised as superior for creating software products, but the software houses show no special bias towards UNIX. At least one,

Cincom, produces some of their packages IBM commercial software using Mantis, their proprietary 4GL, running on a DEC minicomputer under VAX/VMS.

In large installations in house software is usually created with either a 4GL or (sigh) COBOL hooked to any of the widely available database management systems. Although not a very comfortable development environment, because software must be tested under conditions approximating those in which it will be run, and for connecting to other programs and data files which already exist, there seems very little choice.

There is only slight UNIX penetration in the commercial world. No matter what publicists and apologists may claim, few large scale shops are serious UNIX users for their real work load. Why ? There are a whole flock of reasons:

1. Lack of high quality commercial software packages in the UNIX market.
2. Unavailability of commercial programmers trained and productive under UNIX.
3. UNIX only marginally available on some of the major machines.
4. At best, primary vendors continue to be quite reluctant UNIX supporters.
5. Too many existing versions of UNIX for the comfort of installation management.
6. The widely advertised UNIX portability does not seem to work except with low level programs or by paying for carefully (and expensively) orchestrated conversions.
7. Continued controversy over the attempt to create a standardised UNIX and validate the resulting product.

There is one additional factor, a question of personal and professional attitudes, which reluctantly must be added to the list. This point is neither provable nor can it even be quantified. Nevertheless, it is real.

Large users have noted the sneers of disdain with which much of the UNIX community, especially those primarily affiliated with the academic world, talks to others who do not agree with them. Being told if you are not using UNIX, you clearly must be computational moron does not help UNIX to win many supporters in the commercial world. Nobody likes to be told they are stupid. Yet, this seems a regular feature of many confrontations between UNIX adherents and the commercial world. There are times when UNIX would be better served by people keeping silent. (This is apt to prove difficult for those accustomed to the give and take commonly associated with academic life but it is worth trying.)

Today, UNIX is a major factor in the world of the engineering workstation, the software development laboratory, and the university computer science environment. How far UNIX will move from this base in years to come is very conjectural.

There does appear some differences exist between North American and European views of the current status of UNIX. From the Western side of the Atlantic, Europeans seem much too optimistic about the long term future for UNIX. How much does this reflect a thinly disguised European attack upon IBM and DEC ? Hard to say. Even though UNIX comes from North America, observers find the case for an increasingly wide spread use of UNIX in general purpose computing to be quite unconvincing.

The Future for UNIX

Those who expect UNIX to become a major factor with a good deal of impact upon the commercial side of the industry, together with those who find all present operating systems less than satisfactory, have quite a number of major issues to address in the near future. Coming up on the wrong side of any single one of these issues will not cause UNIX to blow away, but too many wrong answers may cumulatively sink the UNIX ship. Taking

the issues in no special order:

Issue: *Does anyone know how to define UNIX ?*

- There clearly is a UNIX V, currently Version 3.0. It is described in a document, SVID (the **S**ystem **V** **I**nterface **D**efinition) and by any legal test is the intellectual property of AT&T. There are also such interesting derivatives as Xenix, Ultrix, IX/370 and a multitude of others which have been built. Most use some or all of System V as a base. There is also POSIX, the Portable Operating System - IX, formally known as IEEE 1003.1. POSIX seems to represent the U.S. government's way to snatch AT&T's property rights away from the developers and transform UNIX into a public domain standard. While POSIX and SVID-defined UNIX will move closer to each other, for the moment it is not clear which is UNIX.

Issue: *Is the rejection of the SVID validation test suite by European vendors merely a ploy to avoid paying licence fees to AT&T or a way to weaken the standard so they can implement whatever they elect to include ?*

- This is a tough question. While vendors whine about the fees AT&T is charging, without formal verification their claims of being System V compatible are only unproven assertions. Languages such as COBOL and Ada now are verifiable by independently developed tests. Why are UNIX implementors so nervous about having their claims tested

Issue: *Why do so few major commercial applications packages such as Payables, Receivables, General/Ledger or Payroll run under UNIX ?*

- This is a question of pure economics. When the major independent software houses sense the presence of a viable market, a critical mass of buyers in the UNIX community, they will begin to migrate their packages. For the moment, most major software houses do not see a feasible market for what will be an expensive conversion. The way out of this "chicken and egg" dilemma may be to implement typical commercial applications software using a 4GL which runs under UNIX as well as one of the existing operating systems.

Issue: *Why doesn't UNIX really understand data structures ?*

- UNIX and the C programming language are not oriented to high level data structures. The origin of UNIX (of and by system programmers whose fundamental concerns were the manipulation of low level objects) is clearly evident in this particular area.

Issue: *UNIX remains weak in certain features essential to high volume, commercial data processing including tape drive support, error handling, security and data file handling. When (if ever) will this be corrected ?*

- This is another area in which the origin of UNIX shows in strengths and weaknesses. an operating system can not be equally strong in every function and feature. Building an operating system involves a series of trade offs. For example, giving the users a high degree of command language flexibility may require the omission of tight security measures.

Issue: *UNIX is weak at transaction processing. Can this be corrected ?*

- Since UNIX is quite strong at dealing with low level objects, one would think it not impossible to add support for communications, mirroring of data files and high speed interrupt processing, necessities for transaction processing. The question is:

would the resulting operating system still be recognisable as UNIX ?

Issue: *Can data integrity in UNIX be strengthened by going to mandatory locking ?*

- Certainly. However, mandatory locking will slow down a system when many users are trying to enter the same (system) file. Besides, it is hard to find any real examples of the need for mandatory locking except within very high security, cryptographic applications. In these instances, the user's best move would be to run on a single user machine.

Issue: *If AT&T is forced out of the picture by POSIX, who will take over the long term maintenance, upgrading and enhancement of UNIX ?*

- Probably a committee which would operate under the overall blanket of either ANSI, the American National Standards Institute, or the IEEE, and eventually, ISO, the International Standards Organisation. While AT&T would likely remain chief developer, their role as "proprietor" of UNIX would be diminished.

Issue: *Will UNIX handle current hierarchical and relational DBMSs ?*

- Many popular relational DBMSs are available for UNIX. The older hierarchical systems, IDMS, Adabase, IMS and Total have never been converted. At this date, it would seem unlikely this will ever be done. Why go backward ? SQL is widely available and the distributed DBMSs have started to appear. That's plenty!

Issue: *Are UNIX applications really portable ?*

- Yes, if you believe the recent EEC demonstration held in Luxembourg by X/Open of a modified (to comply with X/Open's CAE, Common Application Environment, standards) version of Access Technology's 20/20 integrated spreadsheet running on 11 systems. Aside from specially rigged demonstrations of CAE versions of UNIX, portability is not clear. If it was easy, no conversion companies would exist. It is hard to visualise most vendors giving up their existing proprietary versions of UNIX. Also, it is not clear if identical results will be obtained when running a complex program across many systems. At some point, basic architectural differences will surface. This is especially true if mathematical precision is involved.

Issue: *Will UNIX survive if AT&T is forced to get out of the computer business?*

- UNIX survived before AT&T was in the computer business and there is no reason to believe it would not survive AT&T's departure. Could anything happen to AT&T much more disasterous than their 1986 financial results ? Being forced to lay off 27,000 employees and managing to lose $3,200 million is not trivial. The AT&T "UNIX PC" has been something less than a major market triumph.

Issue: *What will be the effect on the UNIX market place of a multi tasking, multi user MS-DOS for the Intel 80386 ?*

- No known impact. All recent gossip out of Microsoft suggests such a development is at least 8-24 months away, so why worry about it ?

Issue: *Can you convert applications which have been running on an IBM System/38 to a UNIX machine ?*

- No more easily than you can convert a System/38 program to any other machine. The IBM System/38 has the most sophisticated operating system ever created including a built in relational DBMS. Conceptually it is far, far ahead of any other operating system.

Issue: *Will any two versions of UNIX ever be identical ?*

- No, but this is not important. An operating system has two sides, one faces the machine and the other is the programmer's interface. So long as the program interface achieves stability, the machine side can (and will) change to keep up with the hardware. When a vendor moves from a Motorola 6800 to a 68000 and then to a 68020, what happens on the machine side is not material so long as the applications continue to run undisturbed. The same process occurs as the hardware goes from a single processor to multiple processors. To the programmer, it is all immaterial. This is one of the questions X/Open's CAE seeks to address.

Issue: *What will it take for UNIX to be successful in the end user market place ?*

- End users do not really make decisions about operating systems, data processing professionals call the shots. At least 99% of the end users who sit happily at their keyboards pounding away on "1-2-3" or "Word Perfect" do not understand MS-DOS. Many do not realise it even exists. Why should anyone expect users ever to grasp the nuances of UNIX ? Nor should they have to learn UNIX or any other operating system to do what they have it do.

Issue: *What will it take for UNIX to displace IBM's MVS or DEC's VAX/VMS ?*

- A total rewrite of UNIX adding features orientated to commercial data processing for which UNIX today lacks support. This cannot really happen. such a task would destroy the inherent strengths in the UNIX architecture. It would be sheer folly even to attempt such an effort and it is hard to think of any real reason it should be done.

Concluding Comments

Contrary to what will be said at this EUUG conference, UNIX is not the long awaited panacea which will instantly solve all accumulated programming problems from the past two decades. A fair number of practioners, especially those with no sense of history, seem determined to impose UNIX as the only solution regardless of the situation or problem to be solved. This is sheer nonsense but the dedication to a cause and single minded devotion of a true missionary should never be under estimated.

Having seen many, many previous pseudo-panaceas, the true historian of the computer industry knows there cannot be any single solution. It was not many years ago when each of the following was being touted as the one and only solution to all data processing problems: COBOL, PL/1, MIS (Management Information Systems), IRM (Information Resources Management), DBMS, POLs (Problem Oriented Languages), APL, and Management by Objectives. Will UNIX join the list ?

UNIX is a nice operating system. It has some solid, positive features which can prove to be very useful in a constrained subset of all problems. It is not a universal solution nor did anyone with any shred of knowledge ever pretend this was the case. The small group of zealots really does not know what it is they are talking about.

For UNIX to achieve greater use, a certain amount of major surgery will be required. Before pulling out the knives, however, it might be wise to think carefully of what will be left of today's UNIX if all the changes which have been suggested are made to it. Wouldn't this be just another case of propping up the name and sliding a whole new operating system under it ?

It is better to have a generally useful; even if slightly flawed system, today's UNIX, than to create another monster in an attempt to be all things to all people ?

A reasonably objective look at UNIX reveals a few warts. The system is good but not great. If the applications to be supported involve high volume data or real time transaction processing or intensive use of mass storage peripherals, UNIX may not be the ideal choice. However, there likely is no single pick (not a pun) for all applications, all the time. This should not be interpreted as disparaging UNIX but rather a slap at those who constantly try to sell UNIX as the cure for all of data processing's ills.

While those within the UNIX community are amused by the ongoing battle between various versions of UNIX deciding which will emerge as standard, management of large installations does not find it funny. They wish to make their commitment to a system which has generally wide spread support.

For too long, the adherents of POSIX and System V have been flailing away at each other in print. Those who believed in Berkeley 4.2 and its successors as the ultimate version swing freely at all of the others.

While these pleasant, amusing side shows provided ammunition for the press to fill up empty columns, to those with bottom line responsibility for multi million dollar budgets, the whole thing sounds like a bunch of spoiled children arguing about who owns which toy in the playroom. There are no doubts about who is in charge of VAX/VMS or MVS/XA, today or in the future. While it has not generally been very widely realised, there are inherent negatives in betting major piles of development money on a system when there is no single control agent. AT&T's current financial struggles in the computer business are not calculated to reassure those who have doubts. Management by ANSI (or any other committee) is equally suspect in the eyes of senior MIS management.

While IBM and DEC both support versions of UNIX, neither of these two giant corporations exhibit much enthusiasm for the system. The Armonk/Maynard attitudes have rubbed off on their major customers who see little reason to move away from the primary products. Firmer support from IBM and DEC would greatly facilitate UNIX being able to strengthen its position in the commercial world.

The lack of commercial software, conventional off the shelf packages continues to be a major UNIX negative. Large shops do not want a brand new Accounts Payables or General Ledger, they want the versions which have already been marketed by MSA and McCormick & Dodge, tested out in 1,000 or more other installations, and approved by their internal and external auditors.

The problem is these packages work only with the standard hierarchical database management systems and have not yet been adapted to run with relational DBMSs. While there are some exceptions to this rule of thumb, in the main it remains true. The conversion work now in progress seems largely to be bounded by IBM's DB2, not likely to become a favorite with UNIX users.

There is no easy answer for the package software dilemma. The critical mass of installations is lacking. Nor has there been very much user pressure for a UNIX conversion. Until the key 8-10 commercial applications packages come over to UNIX, it does not seem likely we will see UNIX a major factor in the large scale commercial world.

Conversion and portability remain an unproven case in the eyes of serious data processing professionals. The simple pragmatic test of observing how long it takes to bring up new versions of UNIX makes many quite skeptical.

On paper, moving from X's version of UNIX to Y's is supposed to be easy. Yet, outsiders see many programmes available only on one manufacturer's version and not on that of their next door competitor. The widely publicised X/Open experiment does little to quiet lingering doubts since by their own admission both UNIX and the application, a

simple one, had to be modified.

The bottom line in the commercial world is UNIX is of some interest for selected applications but only the U.S. federal government would be stupid enough to consider mandating its use for general purpose computing. On the other hand, these are the same people which keep telling us how good the Ada language will be for commercial computing although how they will run Ada on the 15 year old computers (not to mention their remaining 1401s) is a most fascinating question. Their credibility isn't very high.

Automating Administration of UNIX-Systems with Thousands of Users

Ernst Janich

Universitat Ulm
Sektion Informatik
Postfach 4066
D-7900 Ulm
W-Germany

ABSTRACT

On heavily populated UNIX Systems, it is impractical if only one person can be the super user to manage administrative problems or simply to do administrative work. Security requirements prevent that the super user password be available for a even small community. Extensive use of the group ownership and group protection can help to facilitate security-sensitive programming.

The problem - to manage a UNIX system with several thousand users - was once solved at Ulm under UNIX V7 by making extensive changes to essential UNIX commands. Advantages and disadvantages of this are discussed.

The former solution is compared with the new attempt that tries to use a UNIX System V without modifying original sources. The second goal on the new solution is to make life and administration work as easy as possible.

Using all the administrative possibilities in UNIX System V helped to bring up a System automatically after power fail without the need to touch any key. That is the easiest way for unexperienced users. If problems arise, experienced users can still manipulate the automated boot. There was no need to add difficult commands.

Distributing and managing changes on a system over ethernet or terminal lines, only using *uucp* respective *mail*, can be done in a secure way. On heterogeneous UNIX systems todays standard for communication is *uucp*. If some systems are still only V7 compatible or do not have ethernet hardware, distribution over terminal lines is still necessary.

Even when people are using personal computers with UNIX and connect to a central line only sometimes, they can automatically participate in software distribution. It is shown how this can be done, using only standard software.

The use of special hardware from every site poses protection problems. A solution is conceivable - even on heterogeneous (binary) systems. Our solution will be presented and discussed.

I will describe two solutions. The first one was implemented 1984 in Version 7 of UNIX. The new one is implemented in UNIX System V.

1. High user population

The city of Ulm manages a central computer system for all their schools. There are about 1800 users on this system. A similar environment might be found at universities or at sites where people do not need a computer every day. A typical pupil does not work more than twice per week at this system.

The problem for such a system is not heavy machine load, but a big user population, in this special case a rapid and often changing population.

The users of this central computer system are often miles away from the machine. The system manager or super user does not know "his" users on this machine. Neither has he the chance to find out, if a request of a new user to be added to the community is legal, nor the right to refuse such requests.

The teachers of the pupils know their class, the names of their users - but teachers are nearly regular users.

You cannot not give dozens of teachers super user privileges, just for the case to manage their pupils, and hope that you can run a secure UNIX system.

Amazingly often, users forget their password (or they manage to enter a password in such a strange way that they do not remember how to get it again). Another problem is that you must control the amount of space users keep. If most of them use the maximum allowed, you probably will run out of disk space.

Managing a limited amount of users, and only those, is desirable. For pupils this should be done by the teachers. Teachers loose their passwords too. Hence the privileges of a teacher should be limited to his class.

Teachers are managed by the system manager or super user.

2. A hierarchical user environment

2.1. Password hierarchy

Big password files slow down some essential commands. Managing one big password file with often changing entries requires long locking. It is hard to control, who is allowed to change entries or fields of some entries.

The old version was realized with a password directory. The files therein - represented by group names - kept only password entries for one group. Hence user names had only to be unique within one group. That is much easier to handle than unique names over a whole system. Splitting */etc/password* into several files means, that all access problems can be managed by the regular UNIX file protection mechanism.

The **login** sequence required user name and group name. **mail user** did send **mail** to **user** in the same group, or **mail user/group** could send to somebody else. Mailing privilege could be limited via the password file. We controlled, "who" can send to "whom", or "who" can get mail from "whom". **ls -l** reported user and group by default.

One entry in each password file identified the manager of the group. With a password editor, realized with *curses*, he could change passwords, eliminate or add users, manage mail privilege etc. He could "su" to users within his group without knowing their password.

2.2. System manager not super user

The system manager does not have to be identical with the super user. In a similar way (as shown for group manager) the system manager managed the group manager. He could "su" to them, add or remove them and their groups. For the system manager there was a bunch of shell scripts that helped to add or remove groups and to check the integrity of the management.

Even the weekly backup procedure was based on the group hierarchy.

2.3. No super user password

There was no unique super user password. The system manager could bring users into a "su" password file. There, every user had to manage his own "su" password. The number of users who could "su" was severely restricted.

Even at the system console there was no way to login as *root*. Checking the accounting files could show who "su"ed where.

2.4. Modifications for hierarchical user management

As mentioned, **ls** was modified for easier use. **password** and the **getty/login** sequence had to be modified, **su** was severely restricted.

The biggest change was made for **mail**. Keeping **uucp** running with our **mail** lead to an unreasonable amount of changes. Caused by the redesign of the password file, the restriction that user names are only eight characters long, was eliminated.

3. Set uid for shell scripts

Most of the programming for system management can be done easier with shell scripts. The showed hierarchy sometimes poses the need to use the *set uid* or *set gid* feature. Regular shell scripts do not offer this. To facilitate programming, I implemented this feature in a secure way. The shell scripts are kept in a separate well protected directory (unreadable for regular users). A special program that was linked to an appropriate name, cared on executing a shell script with the desired *uid* or *gid*.

4. Vanilla System V

Only some months after all these changes, System III and then System V came up. It showed that it would lead to the same or even more amount of work to port our group hierarchy to System V.

Because changes to **getty/login** had been made, based on the knowledge of the original sources, it was not possible for somebody with a binary license to get our changes.

We wanted to meet the goal to keep System V as it is delivered. All changes are add-ons, based on the original system and commands.

4.1. Administration of population under System V

A hierarchical management of users needs a password editor that takes care on group privileges, adds and removes a directory for users etc. This leads to a nontrivial program. It is less elegant than the old solution, is not a good UNIX tool, but is portable.

The idea of putting people into groups was kept. The password file will grow. We now have more machines, hence this capacity should hide the bigger overhead.

Group managers need to be identified by their *uid* and *gid*. A simple trick helps: if *uid* and *gid* are equal, a user will be a group manager.

Allowing "**su** to arbitrary group members" for group managers, without knowing their password, would need to change the **su** command. The change is simple.

When group managers add new users, a simple algorithm suggests user names by putting initials and family names together. Unfortunately many German names must be cut. Our self made password file with arbitrary long user names was more polite.

4.2. Local environment

Commands are no longer changed to meet local needs. An example: the German standard for paper size is different from the American standard. We neither want a user having to specify the paper length each time he uses a command, nor do we want to change the original commands.

With help of */etc/profile* we bring the **/u/bin** directory in front of the PATH. Correct settings for the European and local environment are placed at **/u/bin**. New and self made commands reside there too.

5. System management

5.1. Automating booting

Booting a system should bring it up in a desired state. You might specify this state as **initdefault** in */etc/inittab*. If problems like unrepairable errors for **fsck** arise, you might wish that the bootstrap procedure should stop.

To achieve this, I specify state 0 as **initdefault**. State 0 is single user mode, and only the *root* file system is checked and repaired. Even this can be prevented, if the interrupt key is pressed while entering state 0. A regular shell will appear for repairs by hand.

As last action in state 0 **init 6** is called to enter state 6 - the desired final state - but only if no problems arose during state 0.

Be very careful when specifying **initdefault**. Unless you plan actions for problems, the default state is entered, and there is no way to prevent this. It is no good idea to let users login, run spooling and system activity reporting on damaged disks, or to mount bad file systems.

5.2. /etc/inittab as finite state automata

Depending on what has to be done, different states are necessary to run a machine. Regular multiuser mode with all network connections active will be the default. This should be started automatically after booting. If hardware problems exist, multiuser mode without network might be useful. Another less active mode disables system activity reporting too.

For backup we want a single user terminal active, but all disks mounted, system activity reporting disabled. Just for the case of severe problems, the lowest mode has everything disabled.

States are described in */etc/inittab*. For state changes stamp files control which activities are enabled. The boot procedure deletes all activity stamps. After each state change **who** informs which state is entered.

5.3. Disk management

Large disks are logically divided into many pieces of equal size. This helps if changes of the layout are necessary or when big parts are copied. Several pieces might form a file

system. Those parts of the file system that grow or change are mounted on separate, probably small, file systems. Hence parts that are constant under normal circumstances are on their own file systems.

Each group of users resides on a separate file system. Exceeding disk limits affects only a part of the system.

For file systems we create linked device names with a name like the directory name they are mounted on. This prevents that after spelling errors files systems are mounted at the wrong place. The better mnemonic makes management much easier and it makes management procedures portable to various systems.

5.4. Low training of system managers

Managing machines by state changes, automated commands, controlled actions and good mnemonics helps a lot to save trouble.

Machines can be managed by a group of system managers. There is no need that all of them get detailed training.

Activities (state changes) like mounting, dismounting or spooling and system activity reporting, are controlled by careful commands. Mounting is done, only after a file system is checked. Mounting and **fsck** are only tried, if not yet mounted. In */etc/fstab* we specify if a file system is used "read only" and where it is mounted. **mountall** mounts all file systems of */etc/fstab* that are not currently mounted.

A command that is called with wrong or no parameters should tell how to use it correctly. A command should do its work for the regular most common jobs with very few parameters.

6. Distributed systems

6.1. Preparing a distribution

make install creates correct entries for */etc/inittab* and other site dependent files. For system dependent files, a common header part, a middle part depending on the system name and a common tail are put together.

File systems are build according to local disk parameters. A manual page for each supported machine is maintained. It helps an unexperienced user to bring up a system from scratch. He must not know disk types, number of cylinders or boot block offsets.

6.2. Managing systems on various machines

All system data for all machines are kept at one place, at the master system. Most of the system files never change. There is no need for weekly backup of such parts. A copy of them is at every site (or might be brought to it). Parts that change are kept in some specific directories.

If commands should be site dependent, they contain either code that checks the system name, or they are made by an installing procedure.

A system manager, who wants to distribute software, must have knowledge of site dependent features of the systems where he is sending new software to. New commands are sent to other systems as source, along with *makefiles*. They are automatically compiled on the remote system.

System parts, e.g. new kernels, source files, commands for remote execution, etc., are sent to remote systems via mail. A demon on the remote system looks at a specific mail file,

controls carefully sender and origin, and checks a crypted time dependent password in the mail. Unfortunately the password has to be kept in readable uncrypted form on each machine. If somebody imitates my remote mail, knows the password and sequence number, he may bring Trojan horses into our system.

Decrypt is not available in new European systems. Hence we cannot use it. If we had it, transmissions would be much saver. Sending new software by mail is an advantage for those systems that are hooked to a central line only sometimes.

uucp and **mail** are available everywhere, so every system can participate in software distribution.

6.3. Restricted use of special hardware

Regular users need only to specify the destination. Access to remote devices is hidden by commands in the **/u/bin** directory.

Before spooling output to special files is done, parameters can be set with a *model* shell script. The access rights for a special device can be checked there too. This is easy for users working on the same machine where the special device is connected.

The best way for heterogeneous UNIX systems for remote spooling is **uux**. Unfortunately when using **uux** the senders identification is no longer secure. The *login id* is **uucp** and the originating user is transmitted as **uux** parameter.

A secure transmission of access rights could be achieved by a similar procedure as in the last section, but the overhead would be equally high.

7. Future

We want to keep every user with his files on several machines. When he arrives in the morning he chooses one system. That will be his system for this day. Later **login** at another system can be prevented. At night his files are sent to the other machines.

If a system fails, or if it is too heavily populated, each user has the choice of machines every day.

One of the bigger problems is changing the password file. If a user logs in simultaneously at two systems, he will disturb the distribution.

8. Conclusions

Automating management not only saves time but also a lot of trouble caused by unexperienced users. Packing typical tasks makes every day's work easier and possible to delegate. On heavily populated systems the establishment of group managers keeps management clear. By automating software distribution, a single system manager (or a small group of them) can maintain a similar environment on several machines.

9. References

F.T.Grampp and R.H.Morris, "UNIX Operating System Security", B.S.T.J., 63, No. 8, Part 2, October 1984, pp. 1649-1672

J.A.Reeds and P.J.Weinberger, "File Security and the UNIX System Crypt Command", B.S.T.J., 63, No. 8, Part 2, October 1984, pp. 1673-1684

D.A.Nowitz, "UUCP Implementation Description", UNIX Programmer's Manual, Section 2, AT&T Bell Laboratories

B.Lang, "UNIX fur viele Benutzer", Sektion Informatik, Universitat Ulm, 1985

The development of a standard UNIX system for Intel®-based microcomputers: A technical perspective

Doug Michels
Vice President

The Santa Cruz Operation, Inc.

ABSTRACT

In just a little more than its first three months, 1987 has already been a year in which more significant developments than ever before have been announced that ensure the future direction of the UNIX System for personal computers.

Particularly significant was the recent announcement that AT&T had joined with Microsoft Corporation - and its XENIX® development partners, The Santa Cruz Operation, Inc. and Interactive Systems Corporation - to establish a standard UNIX System porting base for Intel processor-based microcomputers, based on merging the latest in XENIX and UNIX technologies.

There is a clear need to answer specific questions concerning the roles of the UNIX and XENIX Systems, as well as their prime developers - AT&T and Interactive, Microsoft® and SCO - in addressing the new and emerging needs of the multiuser market: the 80386, internationalisation, UNIX System V Release 3, and beyond.

SCO is proud to be a major contributor to each of the recently announced developments that pave the way towards a single, standard UNIX System porting base for personal computers. We offer this background paper in the interest of clarifying the roles of both the participating developers and their respective products, explaining the key technical issues involved, and providing specific answers to what we have found to be the most frequently asked questions concerning these issues.

Contents

- Primary relationships between the major participants through 1986.
- Recent major announcements (1987).
- The evolution of XENIX System V 386 Release 2.2.
- The critical contribution that XENIX 386 2.2 makes to the forthcoming standard UNIX System porting base.
- What will be on the new standard UNIX System porting base?
- How will the new SCO packaged product differ from the new porting base?

- What does the AT&T/Microsoft agreement mean to developers today?

Primary relationships between major participants through 1986

- AT&T originally developed (1969), licensed and distributed the UNIX Operating System to universities (mid-1970s), and later, to other software developers (1979 to present).
- Envisioning the value of a commercially-enhanced version of the UNIX System for business applications, Microsoft licensed UNIX source code from AT&T for the development of its own XENIX System for PDP™-11s (1980s).
- Realizing the potential of the XENIX System in turning personal computers into multiuser business systems, The Santa Cruz Operation, Inc. licensed XENIX source code from Microsoft, and developed the original port of XENIX to the 8088-based IBM® PC. SCO became Microsoft's "second source" for XENIX, as well as co-developer and sole distributor of unbundled packaged XENIX product (1982 to present).
- Recognizing both the emergence of Intel processor-based PCs and the need to promote the UNIX System on them, AT&T contracted Intel to port the UNIX System to its own processors (1983 to present).
- Intel contracted the actual development work for these ports to third-party companies such as Interactive Systems, chosen most recently to develop UNIX System V Release 3.0 for the 80386 and thereby gaining valuable UNIX V.3 expertise (1986).
- As part of this development, Interactive recognized the potential of the 80386 to run DOS as a process under UNIX, and worked with Phoenix Technologies to develop VP/ix™, an operating system extension which will provide a bridge between UNIX and DOS (1986).
- In order to provide DOS capability to UNIX users running VP/ix, Phoenix licensed DOS from Microsoft to bundle with VP/ix. Microsoft, in turn, worked with Phoenix to provide "hooks" for VP/ix capability in its own XENIX System (1986).

Recent major announcements (1987)

- Microsoft and SCO announce merger of their respective XENIX kernels with introduction of XENIX System V Release 2.2, establishing a standard XENIX System for all future releases. (UniForum™, 1/21/87)
- Microsoft, SCO, and Compaq announce that SCO will assume from Compaq the distribution and support of XENIX System V for COMPAQ 286- and 386-based personal computers, with SCO XENIX 286 System V Release 2.2 replacing XENIX System V/286 by Compaq. (UniForum, 1/21/87)
- Microsoft, SCO, and Interactive announce agreement to merge current XENIX and UNIX technologies in a cooperative effort to develop XENIX as the standard UNIX System product for computers based on Intel 80386 microprocessors and beyond. SCO and Interactive would bring to this development effort their respective areas of expertise - XENIX System V 386 Release 2.2 and UNIX V/386 Release 3.0. (UniForum, 1/21/87)
- Microsoft and AT&T announce agreement to develop a new implementation of UNIX for the 386 that would be upwardly compatible with AT&T UNIX System V/386 Release 3.0 and XENIX System V 386 Release 2.2. This product would be developed

by Microsoft under contract to AT&T. For this development, Microsoft would draw upon the pool of expertise that its partners, SCO and Interactive, had developed. This new implementation would be the sole UNIX System product for the 386 and would be distributed under AT&T's trademarked name, "UNIX". (New York, 2/19/87)

- IBM announces PC DOS 3.3 and Operating System/2 (OS/2) along with its new Personal System/2 Series, which includes four models: the Model 30, based on the Intel 8086 microprocessor; the Model 50 and Model 60, both of which are based on the Intel 80286 microprocessor; and the Model 80, based on the Intel 80386 microprocessor. (Rye Brook, NY, 4/2/87)
- Microsoft concurrently announces the availability of XENIX System V Release 2.2 for the IBM Personal System/2 Series (for which SCO will provide packaged product) to coincide with hardware availability. Microsoft concurrently demonstrates XENIX 386 System V Release 2.2 on the IBM PS/2 Model 80. (Redmond, WA, 4/2/87)

The evolution of XENIX system V 386 release 2.2

XENIX has always been "real" UNIX, only a superset. Conceived as a commercially-enhanced, AT&T-licensed version of the UNIX System, it was only because of prior AT&T trademark restrictions that the name XENIX was created in the first place; otherwise, in every sense XENIX could have legitimately been named "Microsoft UNIX."

XENIX System V 386 Release 2.2 had its origin in the porting base for AT&T's UNIX System V Release 2.0, which AT&T created originally for the VAX. Microsoft and its development partner, SCO, wanted to create a version of XENIX that performed better on 80386 microprocessor-based machines. Microsoft rewrote the memory management code to compensate for the different page size and fault behavior. SCO then added its PC/AT® drivers, drivers which SCO has been refining continuously since the first XENIX system (XENIX System III, circa 1983) was developed for the IBM PC.

The XENIX development team then added full backward binary compatibility from all previous releases of both SCO and IBM XENIX systems, which include IBM XENIX 1.0, IBM XENIX 2.0, Microsoft XENIX 286 System III and System V, and SCO XENIX 86 and 286 Releases 2.0 and 2.1.

To create the binary code for XENIX System V 386 Release 2.2, the development team used the Microsoft C 5.0 Compiler. The Microsoft C 5.0 version of DOS will not be released until later this year, even though XENIX 386 will include it in the first release.

The development team's next step was to add important XENIX extensions, such as shared data, semaphores, mapchan, record locking, a compiler-less link kit, executable data, and rdchk.

Beyond this, SCO provided numerous operating system "add-ons" - such as installation facilities, administration utilities, and windowing - as well as adaptations for international use. All of these will be covered in detail later in this paper.

In summary, this is the XENIX System V 386 Release 2.2 that Microsoft and SCO bring to the UNIX merge effort announced by Microsoft and AT&T.

The critical contribution that XENIX 386 2.2 makes to the forthcoming standard UNIX system porting base.

The result of the Microsoft/AT&T agreement will be a standard porting base that all UNIX System vendors will need to port to the 80386 microprocessor. This standard UNIX

System porting base - in a very real sense, simply the next XENIX upgrade - will enable UNIX vendors to build systems that provide backward and forward UNIX/XENIX binary compatibility and 386 optimization.

The porting base will be created by combining XENIX System V 386 Release 2.2 and UNIX System V 386 Release 3 (the port developed by Intel and Interactive). AT&T, Microsoft, SCO, and Interactive will collaborate on the development of the merged product.

What will be on the new standard UNIX system porting base ?

The new standard UNIX System porting base, or source tape, will contain the AT&T Portable C Compiler, the merged kernel source code, and the standard UNIX utilities, as found on the original UNIX System Version 3 tape with any modification required to support the 386 processor or the new kernel. The tape will contain some new utilities, however, to support the new system calls, such as mapchan, that were derived from XENIX. Installation and system administration will be based on AT&T's standard (3B2) approach rather than the menu based system included in SCO XENIX.

The tape will also include a library interface to provide library support for these new system calls and the I/O controls.

The kernel will contain several important modifications to the "h" files, also to account for the new system call modes, IOCTL defines, data structures, and memory management parameters. It will include, of course, the new XENIX system calls, some new memory management code for the 386 (which will replace the 3B2 regions paging code), and some example files for the configuration directory. In addition, the kernel will include some standard "hooks" for DOS subenvironments such as VP/ix.

How will the new SCO packaged product differ from the new porting base ?

The SCO packaged retail product that will be released concurrently with the new porting base will share the porting base's binary compatibility and kernel functionality, but will include value-added features not found on the porting base, such as the Microsoft C 5.0 compiler with DOS cross development support, SCO XENIX's custom PC/AT drivers for all standard AT devices and a wide range of add on hardware, and SCO XENIX's other add-on features.

A more efficient compiler

The Microsoft C 5.0 Compiler does things that the PCC just cannot do: it consistently generates smaller and faster code specifically optimized for the 386; it compiles binary code for both DOS and XENIX applications, thus allowing XENIX/DOS cross-development and use with VP/ix; and it is available for larger systems - SCO has ported it to the VAX™, for example - to allow enhanced XENIX and DOS development on those machines. As a result, those applications developed with the Microsoft C 5.0 Compiler will undoubtedly perform better on 386-based machines than those developed with the AT&T Portable C Compiler. The Microsoft C 5.0 compiler also utilizes Intel's standard OMF (identical with DOS) format designed specifically for Intel processors. This allows for a high degree of portability to and from DOS languages and applications that rely on OMF.

More drivers

The SCO product also includes a broad continuously expanding range of high-performance, flexible drivers that have been designed for all standard (and dozens of add-on) AT peripherals such as multiport cards, Iomega removable disks, ESDI high performance disks, tape drives, serial consoles, non-standard keyboard layouts, and many

other PC anomalies. These high quality drivers representing hundreds of man-years of development are already included in SCO's packaged XENIX products and will continue to provide the basis of its future offerings and value added features.

Flexible installation features

Installation is another aspect to consider. The new standard UNIX System porting base will not include the flexible installation and features that SCO product will provide - features such as selective package installations, auto-configuration of swap and file-system sizes, dynamic bad-track replacement, and a host of mapping files for 8-bit (European) printers and terminals. These features go well beyond the standard UNIX porting base.

Screen-based administration utility

The SCO product will also provide a sophisticated screen-based administration utility that includes smart back-up-and-restore facilities, as well as a user-friendly interface for adding and subtracting peripherals, for formatting media, for recovering lost files from distribution, and for keeping track of updates.

Europeanisation

As the first of many "internationalisation" enhancements, SCO XENIX System V Release 2.2 provides an 8-bit data path that can support European alphabets, a new data file mapping facility, and enhancements to various device drivers. This new framework enables software developers to write applications specifically for the European market.

Specifically, SCO XENIX System V Release 2.2 incorporates a generic tty driver that supports non-English alphabets which require 8-bit character sets, and simulates dead and compose keys on 7 or 8-bit terminals so that they generate 8-bit codes. This is further supported by implementing 8-bit data paths through all of the SCO XENIX 2.2 device drivers, and by using "channel mapping".

The channel mapping facility permits both "many-to-one" mapping of system input-- from a European terminal, for example, where two or three keys may be input to actually produce one language-specific character on the screen - and "one-to-many" mapping of output - to a European printer, for example, where a single 8-bit character in a XENIX file may be converted to a string of characters that form a constructed character on the hard copy. This feature enables SCO XENIX System V Release 2.2 to support the greatest number of European printers and CRT screens possible.

SCO XENIX 2.2 now also supports non-American time zones, northern or southern hemisphere, with flexible rules for specifying the local time-change laws.

SCO is deeply committed to the international market and has a dedicated team of software engineers based in its London office who are continuing to add important features in conformance with the emerging X-Open standard. In addition, translated software and documentation is being prepared. All of this work will be incorporated into SCO's international versions of the new UNIX product.

XENIX - A Unique Value-Added Product

In summary, the significance of the AT&T/Microsoft agreement is that AT&T has taken a great step forward in providing unity to the UNIX marketplace, but the porting base that will result from this agreement is the *chassis* upon which the *ultimate* UNIX System product must be built. The new porting base will share binary compatibility with the XENIX packaged product and even include some important XENIX extensions. SCO has

already laid the groundwork and continues to develop those features that will significantly enhance the performance and functionality of the *ultimate* UNIX system product. SCO XENIX will also offer other operating system extensions such as XENIX-NET, VP/ix, GSS*CGI, and MultiView™ windowing; all of these value added features will distiguish the SCO packaged product from the standard porting base to be delivered to AT&T.

What does the AT&T/Microsoft agreement mean to developers today ?

Software and system developers can rest assured that today's applications - whether written for XENIX System III, XENIX System V, or AT&T UNIX System V.2 (286) or V.3 (386) - will run on the new standard UNIX System porting base.

However, there are several considerations that point out the advantages of developing with XENIX today.

Those developers who select XENIX can use its cross-development environment to create a DOS or XENIX binary today that is fully optimized for 286 or 386 machines. With the XENIX 386 development system, developers will receive the latest the Microsoft C Compiler which is by far the most powerful and efficient 386 C compiler available today. In addition, the 386 development system contains a full 286 Microsoft C compiler to allow development of a common executable that will run on either 286 or 386 XENIX. XENIX applications will run unmodified on the new unified UNIX System when it is released, while DOS applications will run under VP/ix. XENIX also uses memory management paging code that has been highly tuned for the 386.

While UNIX System V Release 3 is clearly the future direction of the UNIX System, its special features - Remote File System (RFS), streams, etc. - are not yet of major significance to the vast majority of applications. In fact, major organizations that are concerned with UNIX standards (such as Sigma, POSIX, and IBM) currently endorse SVR2. Developers can confidently begin writing applications with today's SVR2-based XENIX 286 and XENIX 386 and expect full compatibility with the forthcoming unified UNIX System when it arrives.

Developing with XENIX also ensures that features such as termcap and terminfo, keyboard and mapping configuration files, 8 bit European device support, custom-based installation, and a bit-mapped graphics interface will be compatible with all new releases of the operating system and all new microprocessor generations from Intel. Only XENIX can provide developers with the kind of stable, but evolving, system they need to anticipate and meet the requirements of a rapidly advancing computer marketplace.

Summary

In conclusion, SCO applauds this major agreement toward developing a single UNIX System standard for Intel-based microprocessors. However, this step will in no way create a "commodity market" in the UNIX market, since the result of the agreement will only be a new standard UNIX System porting base.

It is important to remember that this porting base is primarily defined by kernel functionality and binary compatibility. It does not include the drivers or the many value-added extensions that SCO XENIX provides, and it does not specify the type of software generation system with which the code must be developed. There will always be the requirement, let alone the opportunity and motivation, for vendors to add to this porting base. SCO fully intends to continue providing the UNIX community with the leading-edge packaged product available across the widest possible range of popular hardware and peripherals, one that provides users with the greatest added value available

in the market.

XENIX and Microsoft are registered trademarks of Microsoft Corporation.
3B2 is a trademark of ATT Technologies, Inc.
PDP-11 and VAX are trademarks of Digital Equipment Corp.
MultiView is a trademark of The Santa Cruz Operation, Inc.
IBM, AT, and XT are registered trademarks of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
UniForum is a trademark of /usr/group.
VP/ix is a trademark of Phoenix Technologies Ltd.

Multiple Inheritance for C++

Bjarne Stroustrup

Bell Laboratories
Murray Hill
New Jersey 07974
U.S.A.

ABSTRACT

Multiple Inheritance is the ability of a class to have more than one base class (super class). In a language where multiple inheritance is supported a program can be structured as a set of inheritance lattices instead of (just) as a set of inheritance trees. This is widely believed to be an important structuring tool. I consider this conjecture "not proven", but provide a few examples where it does appear to be useful.

It is also widely believed that multiple inheritance complicates a programming language significantly, is hard to implement, and is expensive to run. I will demonstrate that none of these conjectures are true. The scheme described is fully implemented.

WARNING: The multiple inheritance scheme described here is still experimental and not yet considered part of the C++ language[4].

1 Introduction

This paper describes an implementation of a multiple inheritance mechanism for C++. It provides only the most rudimentary explanation of what multiple inheritance is in general and what it can be used for. The particular variation of the general concept implemented here is primarily explained in term of this implementation.

First a bit of background on multiple inheritance and C++ implementation technique is presented, then the multiple inheritance scheme implemented for C++ is introduced in three stages:

[1] The basic scheme for multiple inheritance, the basic strategy for ambiguity resolution, and the way to implement virtual functions.

[2] Handling of classes included more than once in an inheritance lattice; the programmer has the choice whether a multiply included base class will result in one or more sub-objects being created.

[3] The concept of delegation through a pointer.

Finally, some the complexities and overheads introduced by this multiple inheritance scheme are summarized.

2 Multiple Inheritance

Consider writing a simulation of a network of computers. Each node in the network is represented by an object of class **Switch**, each user or computer by an object of class **Terminal**, and each communication line by an object of class **Line**. One way to monitor the simulation (or a real network of the same structure) would be to display the state of objects of various classes on a screen. Each object to be displayed is represented as an object of class **Displayed**. Objects of class **Displayed** are under control of a display manager that ensures regular update of a screen and/or data base. The classes **Terminal** and **Switch** are derived from a class **Task** that provides the basic facilities for co-routine style behavior. Objects of class **Task** are under control of a task manager (scheduler) that manages the real processor(s).

Ideally **Task** and **Displayed** are classes from a standard library. If you want to display a terminal class **Terminal** must be derived from class **Displayed**. Class **Terminal**, however, is already derived from class **Task**. In a single inheritance language, such as C++ or Simula67, we have only two ways of solving this problem: deriving **Task** from **Displayed** or deriving **Displayed** from **Task**. Neither is ideal since they both create a dependency between the library versions of two fundamental and independent concepts. Ideally one would want to be able choose between saying that a **Terminal** is a **Task** *and* a **Displayed**; that a **Line** is a **Displayed** *but not* a **Task**; and that a **Switch** is a **Task** *but not* a **Displayed**.

The ability to express this using a class hierarchy, that is, to derive a class from more than one base class, is usually referred to as *multiple inheritance*. Other examples involve the representation of various kinds of windows in a window system[5] and the representation of various kinds of processors and compilers for a multi-machine, multi-environment debugger[2].

In general, multiple inheritance allows a user to combine independent (and not so independent) concepts represented as classes into a composite concept represented as a derived class. A common way of using multiple inheritance is for a designer to provide sets of base classes with the intention that a user creates new classes by choosing base classes from each of the relevant sets. Thus a programmer creates new concepts using a recipe like "pick an A and/or a B". In the window example, a user might specify a new kind of window by selecting a style of window interaction (from the set of interaction base classes) and a style of appearance (from the set of base classes defining display options). In the debugger example, a programmer would specify a debugger by choosing a processor and a compiler.

Given multiple inheritance and N concepts each of which might somehow be combined with one of M other concepts, we need N+M classes to represent all the combined concepts. Given only single inheritance, we need to replicate information and provide N+M+N*M classes. Single inheritance handles cases where N==1 or M==1. The usefulness of multiple inheritance for avoiding replication hinges on the importance of examples where the values of N and M are both larger than 1. It appears that examples with N>=2 and M>=2 are not uncommon; the window and debugger examples described above will typically have both N and M larger than 2.

3 C++ Implementation Strategy

Before discussing multiple inheritance and its implementation in C++ I will first describe the main points in the traditional implementation of the C++ single inheritance class concept.

An object of a C++ class is represented by a contiguous region of memory. A pointer to an object of a class points to the first byte of that region of memory. The compiler turns a call of a member function into an "ordinary" function call with an "extra" argument; that "extra" argument is a pointer to the object for which the member function is called.

Consider a simple class **A**†:

```
class A {
        int a;
        void f(int i);
};
```

An object of class A will look like this

```
----------------
|    int a;    |
----------------
```

No information is placed in an **A** except the integer **a** specified by the user. No information relating to (non-virtual) member functions is placed in the object.

A call of the member function **A::f**:

```
A* pa;
pa->f(2);
```

is transformed by the compiler to an "ordinary function call":

```
_A_f(pa,2);
```

Objects of derived classes are composed by concatenating the members of the classes involved:

```
class A { int a; void f(int); };
class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

Again, no "housekeeping" information is added, so an object of class **C** looks like this:

```
----------------
|    int a;    |
|    int b;    |
|    int c;    |
----------------
```

The compiler "knows" the position of all members in an object of a derived class exactly as it does for an object of a simple class and generates the same (optimal) code in both cases.

Implementing virtual functions involves a table of functions. Consider:

---

† In most of this paper data hiding issues are ignored to simplify the discussion and shorten the examples. This makes some examples illegal. Changing the word `class` to `struct` would make the examples legal, as would adding `public` specifiers in the appropriate places.

```
class A {
        int a;
        virtual void f(int);
        virtual void g(int);
        virtual void h(int);
};

class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

In this case, a table of virtual functions, the `vtbl`, contains the appropriate functions for a given class and a pointer to it is placed in every object. A class C object looks like this:

```
        -----------------
        |    int a;     |       vtbl:
        |       vptr .........> -----------------
        |    int b;     |       |      A::f     |
        |    int c;     |       |      B::g     |
        -----------------       |      C::h     |
                                -----------------
```

A call to a virtual function is transformed into an indirect call by the compiler. For example,

```
C* pc;
pc->g(2);
```

becomes something like:

```
(*(pc->vptr[1]))(pc,2);
```

A multiple inheritance mechanism for C++ must preserve the efficiency and the key features of this implementation scheme.

4 Multiple Base Classes

Given two classes

```
class A { ... };
class B { ... };
```

one can design a third using both as base classes:

```
class C : A , B { ... };
```

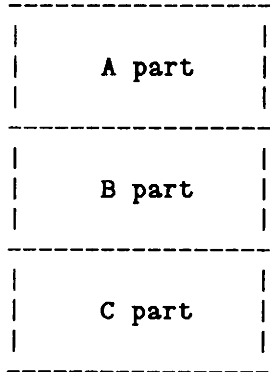
This means that a C is an A and a B. One might equivalently† define C like this:

```
class C : B , A { ... };
```

Object Layout

An object of class C can be laid out as a contiguous object like this:

† Except for possible side effects in constructors and destructors (access to global variables, input operations, output operations, etc.).



Accessing a member of classes A, B or C is handled exactly as before: the compiler knows the location in the object of each member and generates the appropriate code (without spurious indirections or other overhead).

Member Function Call

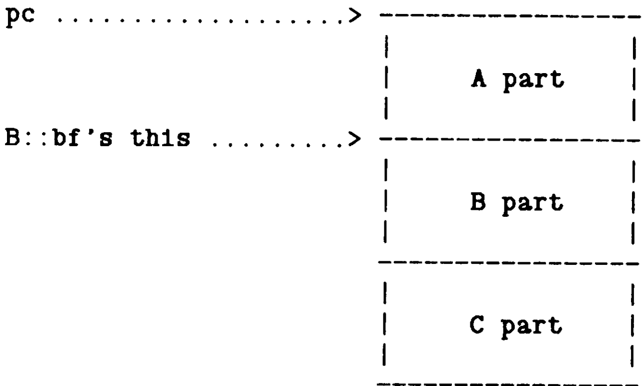
Calling a member function of A or C is identical to what was done in the single inheritance case. Calling a member function of B given a C* is slightly more involved:

```
C* pc;
pc->bf(2);          // assume that bf is a member of B
                    // and that C has no member named bf
                    // except the one inherited from B
```

Naturally, `B::bf()` expects a `B*` (to become its `this` pointer). To provide it, a constant must be added to pc. This constant, `delta(B)`, is the relative position of the B part of C. This delta is known to the compiler that transforms the call into:

```
_B_bf((B*)((char*)pc+delta(B)),2);
```

The overhead is one addition of a constant per call of this kind. During the execution of a member function of B the function's `this` pointer points to the B part of C:



Note that there is no space penalty involved in using a second base class and that the minimal time penalty is incurred only once per call.

Ambiguities

Consider potential ambiguities if both A and B have a public member ii:

```
class A { int ii; };
class B { char* ii; };
class C : A, B { };
```

In this case C will have two members called ii, A::ii and B::ii. Then

```
C* pc;
pc->ii;             // error: A::ii or B::ii ?
```

is illegal since it is ambiguous. Such ambiguities can be resolved by explicit qualification:

```
pc->A::ii;          // C's A's ii
pc->B::ii;          // C's B's ii
```

A similar ambiguity arises if both A and B have a function f():

```
class A { void f(); };
class B { int f(); };
class C : A, B { };

C* pc;
pc->f();            // error: A::f or B::f ?

pc->A::f();         // C's A's f
pc->B::f();         // C's B's f
```

As an alternative to specifying which base class in each call of an f(), one might define an f() for C. C::f() might call the base class functions. For example:

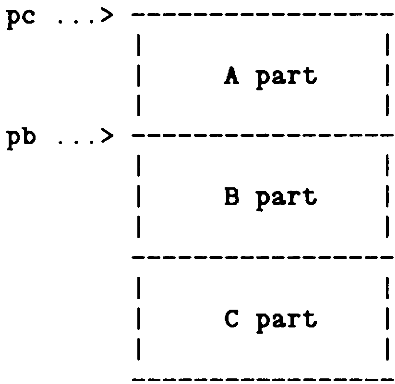
```
class C : A, B {
        int f() { A::f(); return B::f(); }
};

C* pc;
pc->f();            // C::f is called
```

Casting

Explicit and implicit casting may also involve modifying a pointer value with a delta:

```
C* pc;
B* pb;
pb = (B*)pc;        // pb = (B*)((char*)pc+delta(B))
pb = pc;            // pb = (B*)((char*)pc+delta(B))
pc = pb;            // error: cast needed
pc = (C*)pb;        // pc = (C*)((char*)pb-delta(B))
```

Casting yields the pointer referring to the appropriate part of the same object.



Comparisons are interpreted in the same way:

```
pc == pb;        // that is, pc == (C*)pb
                 //          or equivalently (B*)pc == pb

                 // that is, (B*)((char*)pc+delta(B)) == pb
                 //          or equivalently pc == (C*)((char*)pb-delta(B))
```

Note that in both C and C++ casting has always been an operator that produced one value given another rather than an operator that simply reinterpreted a bit pattern. For example, on almost all machines `(int).2` causes code to be executed; `(float)(int).2` is not equal to `.2`. Introducing multiple inheritance as described here will introduce cases where `(char*)(B*)v!=(char*)v` for some pointer type `B*`. Note, however, that when `B` is a base class of `C`, `(B*)v==(C*)v==v`.

Zero Valued Pointers

Pointers with the value zero cause a separate problem in the context of multiple base classes. Consider applying the rules presented above to a zero-valued pointer:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;        // pb = (B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The second test would fail since `pb` would have the value `(B*)((char*)0+delta(B))`.

The solution is to elaborate the conversion (casting) operation to test for the pointer-value `0`:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;                // pb = (pc==0)?0:(B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The added complexity and run-time overhead are a test and an increment.

5 Virtual Functions

Naturally, member functions may be virtual:

```
class A { virtual void f(); };
class B { virtual void f(); virtual void g(); };
class C : A , B { void f(); };

A* pa = new C;
B* pb = new C;
C* pc = new C;

pa->f();
pb->f();
pc->f();
```

All these calls will invoke `C::f()`. This follows directly from the definition of `virtual` since class `C` is derived from class `A` and from class `B`.

Implementation

On entry to `C::f`, the `this` pointer must point to the beginning of the `C` object (and not to the `B` part). However, it is not in general known at compile time that the `B` pointed to by `pb` is part of a `C` so the compiler cannot subtract the constant `delta(B)`. Consequently `delta(B)` must be stored so that it can be found at run time. Since it is only used when calling a virtual function the obvious place to store it is in the table of virtual functions (`vtbl`). For reasons that will be explained below the delta is stored with each function in the `vtbl` so that a `vtbl` entry will be of the form:

```
struct vtbl_entry {
        void    (*fct)();
        int     delta;
};
```

An object of class `C` will look like this:

```
        -----------------
        |               |       vtbl:
        |       vptr ..........>---------------------
        |       A part  |       |  C::f  |     0     |
        |               |       ---------------------
        -----------------
        |               |       vtbl:
        |       vptr ..........>---------------------
        |       B part  |       |  C::f  | -delta(B) |
        |               |       |  B::g  |     0     |
        -----------------       ---------------------
        |               |
        |       C part  |
        |               |
        -----------------

pb->f();        // call of C::f:
                // register vtbl_entry* vt = &pb->vtbl[index(f)];
                // (*vt->fct)((B*)((char*)pb+vt->delta))
```

Note that the object pointer may have to be adjusted to point to the correct sub-object before looking for the member pointing to the `vtbl`. Note also that each combination of base class and derived class has its own `vtbl`. For example, the `vtbl` for `B` in `C` is

different from the `vtbl` of a separately allocated `B`. This implies that in general an object of a derived class needs a `vtbl` for each base class plus one for the derived class. However, as with single inheritance, a derived class can share a `vtbl` with its first base so that in the example above only two `vtbls` are used for an object of type `C` (one for `A` in `C` combined with `C`'s own plus one for `B` in `C`).

Using an `int` as the type of a stored delta limits the size of a single object; that might not be a bad thing.

Ambiguities

The following demonstrates a problem:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B { void f(); };

C* pc = new C;

pc->f();

pc->A::f();
pc->B::f();
```

Explicit qualification "suppresses" `virtual` so the last two calls really invoke the base class functions. Is this a problem? Usually, no. Either `C` has an `f()` and there is no need to use explicit qualification or `C` has no `f()` and the explicit qualification is necessary and correct. Trouble can occur when a function `f()` is added to `C` in a program that already contains explicitly qualified names. In the latter case one could wonder why someone would want to both declare a function virtual and also call it using explicit qualification. If `f()` is virtual, adding an `f()` to the derived class is clearly the correct way of resolving the ambiguity.

The case where no `C::f` is declared cannot be handled by resolving ambiguities at the point of call. Consider:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B {  };          // error: C::f needed

C* pc = new C;
pc->f();          // ambiguous

A* pa = pc;       // implicit conversion of C* to A*
pa->f();          // not ambiguous: calls A::f();
```

The potential ambiguity in a call of `f()` is detected at the point where the virtual function tables for `A` and `B` in `C` are constructed. In other words, the declaration of C above is illegal because it would allow calls, such as `pa->f()`, which are unambigious *only* because type information has been "lost" through an implicit coercion; a call of `f()` for an object of type `C` is ambiguous.

6 Multiple Inclusions

A class can have any number of base classes. For example,

```
class A : B1, B2, B3, B4, B5, B6 { ... };
```

It illegal to specify the same class twice in a list of base classes. For example,

```
class A : B, B { ... }; // error
```

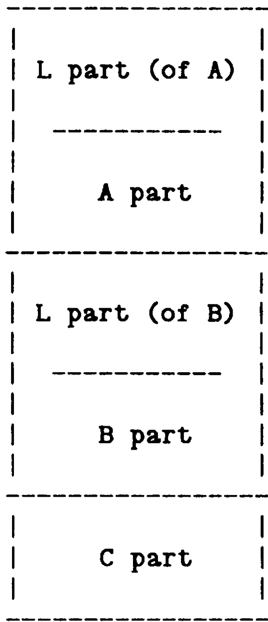
The reason for this restriction is that every access to a B member would be ambiguous and therefore illegal; this restriction also simplifies the compiler.

Multiple Sub-objects

A class may be included more than once as a base class. For example:

```
class L { ... };
class A : L { ... };
class B : L { ... };
class C : A , B { ... };
```

In such cases multiple objects of the base class are part of an object of the derived class. For example, an object of class C has two L's: one for A and one for B:



This can even be useful. Think of L as a link class for a Simula-style linked list. In this case a C can be on both the list of As and the list of Bs.

Naming

Assume that class L in the example above has a member m. How could a function C::f refer to L::m? The obvious answer is "by explicit qualification":

```
void C::f() { A::m = B::m; }
```

This will work nicely provided neither A nor B has a member m (except the one they inherited from L). If necessary, the qualification syntax of C++ could be extended to allow the more explicit:

```
void C::f() { A::L::m = B::L::m; }
```

Casting

Consider the example above again. The fact that there are two copies of L makes casting (both explicit and implicit) between L* and C* ambiguous, and consequently illegal:

```
C* pc = new C;
L* pl = pc;          // error: ambiguous
pl = (L*)pc;         // error: still ambiguous
pl = (L*)(A*)pc;     // The L in C's A
pc = pl;             // error: ambiguous
pc = (L*)pl;         // error: still ambiguous
pc = (C*)(A*)pl;     // The C containing A's L
```

I don't expect this to be a problem. The place where this will surface is in cases where As (or Bs) are handled by functions expecting an L; in these cases a C will not be acceptable despite a C being an A:

```
extern f(L*);   // some standard function

A aa;
C cc;

f(&aa);          // fine
f(&cc);          // error: ambiguous
f((A*)&cc);      // fine
```

Casting is used for explicit disambiguation.

7 Virtual Base Classes

When a class C has two base classes A and B these two base classes give rise to separate sub-objects that do not relate to each other in ways different from any other A and B objects. I call this *independent multiple inheritance*. However, many proposed uses of multiple inheritance assume a dependence among base classes (for example, the style of providing a selection of features for a window described in §2). Such dependencies can be expressed in term of an object shared between the various derived classes. In other words, there must be a way of specifying that a base class must give rise to only one object in the final derived class even if it is mentioned as a base class several times. To distinguish this usage from independent multiple inheritance such base classes are specified to be virtual:

```
class AW : virtual W { ... };
class BW : virtual W { ... };
class CW : AW , BW { ... };
```

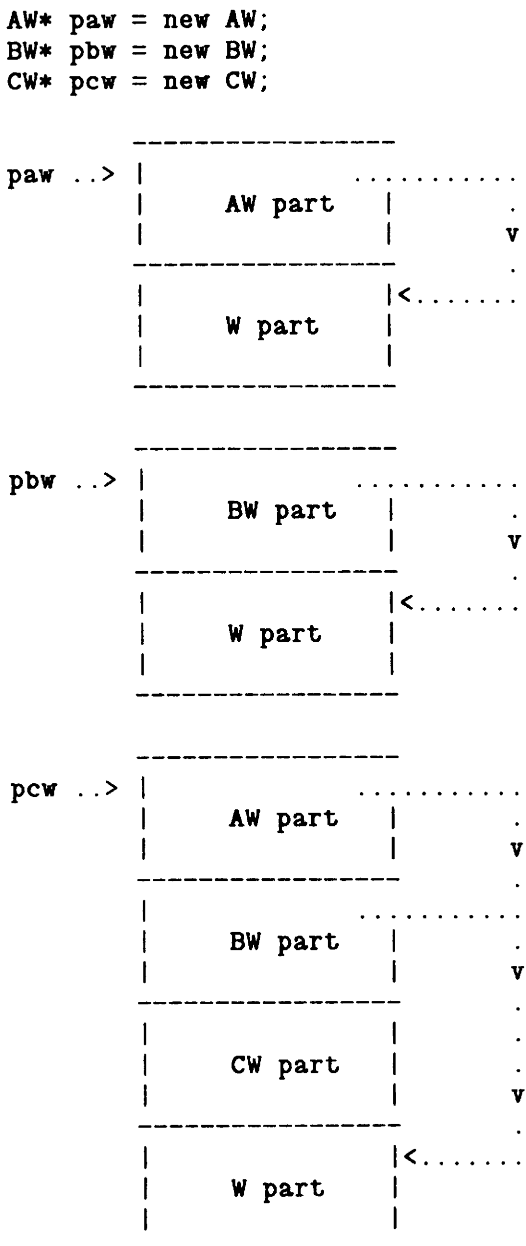
A single object of class W is to be shared between AW and BW; that is, only one W object must be included in CW as the result of deriving CW from AW and BW. Except for giving rise to a unique object in a derived class, a `virtual` base class behaves exactly like a non-virtual base class.

The "virtualness" of W is a property of the derivation specified by AW and BW and not a property of W itself. A set of virtual derivations from a base class W can be resolved provided there is at most one non-virtual derivation of W.

Representation

The object representing a virtual base class `W` object cannot be placed in a fixed position relative to both `AW` and `BW` in all objects. Consequently, a pointer to `W` must be stored in all objects directly accessing the `W` object to allow access independently of its relative position. For example:

```
AW* paw = new AW;
BW* pbw = new BW;
CW* pcw = new CW;
```



A class can have an arbitrary number of virtual base classes.

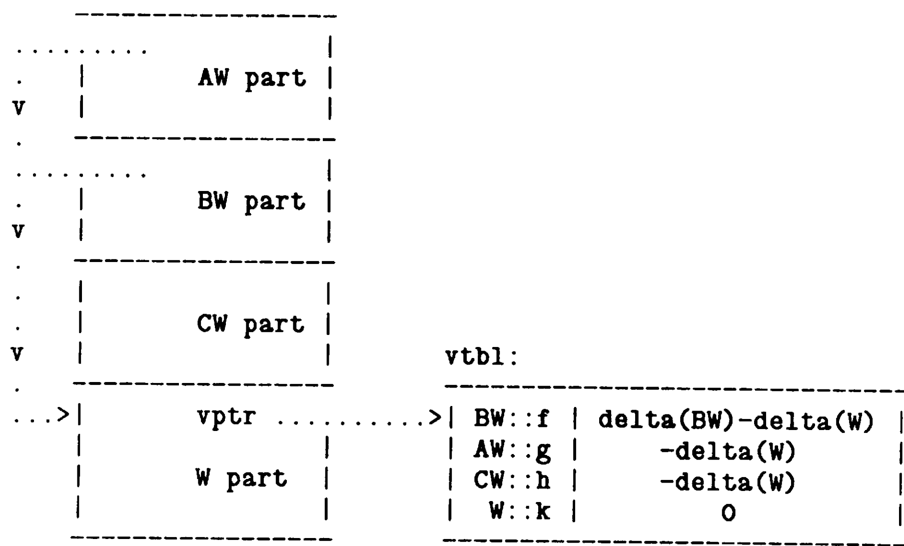
One can cast from a derived class to a virtual base class, but not from a virtual base class to a derived class. The former involves following the virtual base pointer; the latter cannot be done given the information available at run time. Storing a "back-pointer" to the enclosing object(s) is non-trivial in general and was considered unsuitable for C++ as was the alternative strategy of dynamically keeping track of the objects "for which" a given member function invocation operates.

Virtual Functions

Consider:

```
class W {
        virtual void f();
        virtual void g();
        virtual void h();
        virtual void k();
        ...
};

class AW : virtual W { void g(); ... };
class BW : virtual W { void f(); ... };
class CW : AW , BW { void h(); ... };

CW* pcw = new CW;

pcw->f();                // BW::f()
pcw->g();                // AW::g()
pcw->h();                // CW::h()
((AW*)pcw)->f();         // BW::f();
```

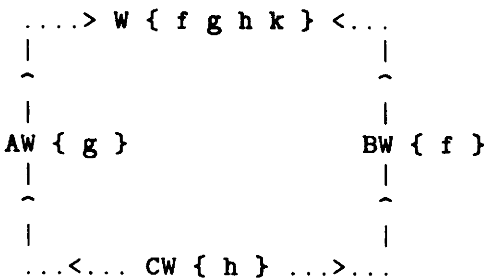
A CW object might look like this:



In general, the delta stored with a function pointer in a **vtbl** is the delta of the class defining the function minus the delta of the class for which the **vtbl** is constructed.

If W has a virtual function **f** that is re-defined in both **AW** and **BW** but not in CW an ambiguity results. Such ambiguities are easily detected at the point where CW's vtbl is constructed.

The rule for detecting ambiguities in a class lattice, or more precisely a directed acyclic graph (DAG) of classes, is that there all re-definitions of a virtual function from a virtual base class must occur on a single path through the DAG. The example above can be drawn as a DAG like this:



Note that a call "up" through one path of the DAG to a virtual function may result in the call of a function (re-defined) in another path (as happened in the call `((AW*)pcw)->f()` in the example above).

8 Delegation through a Pointer

One way of understanding inheritance is as an operation that extends the name space of a class with the names defined for a (base) class and directs the handling of operations specified (only) in a base class to that base. This "name space extension" operation is sometimes called *delegation*[1].

Delegation can be extended to apply to objects that need not be members of the delegating class. In this case, a pointer must be specified denoting the object delegated to. For example:

```
class C : *p {
        // ...
};
```

This means that C delegates to the object pointed to by **p**; **p** must be a global pointer or a pointer member of C (or a public member of one of C's delegates). If **p** is of type B*, the name space of C is extended to include the names from B exactly as if C had been derived from B. Operations on an object of class C that are defined for class B but not specifically for class C will be applied to **p**. For example:

```
class B { int b; void f(); };
class C : *p { B* p; int c; };

void f(C* q)
{
        q->f();          // that is, q->p->f()
}
```

An object of class C will look something like this after **C::p** has somehow been initialized:

```
-----------------
|    B* p; ............>-----------------
|    int c;     |       |    int b;     |
-----------------       -----------------
```

Casting is affected by delegation; casting a pointer to a class to a pointer to a class it has delegated to yields the value of the delegation pointer. For example:

```
void g(C* q)
{
        (B*)q == q->p;  // true
}
```

Delegation through a pointer looks promising for defining interfaces without providing "dummy functions" as is sometimes done like this:

```
class B {
        // ...
        void bf();
        // 15 other functions
};

class C {
        B* p;
        // ...
        void bf() { return p->bf(); }
        // 15 other ''dummy'' functions
};
```

Note that delegation though a pointer specifies a dynamic binding of one object to another: the value of p in the example above can be changed at run time.

Ambiguities

One can delegate through more than one pointer. For example:

```
class A { void f(); };
class B { void f(); };
class C { void f(); };

A* pa;

class D : *pa, B, *pc {
        C* pc;
        // ...
};
```

The usual ambiguity rules apply:

```
D* pd;
pd->f();                    // error: ambiguous
((A*)pd)->f();              // pa->f();
((B*)pd)->f();              // pd->B::f();
((C*)pd)->f();              // pd->pc->f();
```

Virtual Functions

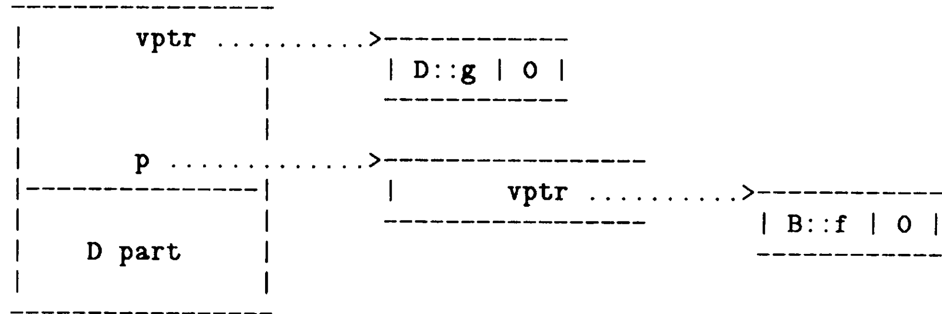
Delegation *through a pointer* differs from delegation *to an object* (that is, inheritance) in that function declarations in the delegating class do not affect the vtbl of the object delegated to. For example:

```
class B { virtual void f(); };
class C : *p { B* p; virtual void g(); };
class D : C { void f(); void g(); };
```

```
void f(C* q)
{
        q->f(); // calls B::f not D::f
                // since C has no virtual function f
                // except the one inherited from B
                // and B is not a base of D
        q->g(); // calls D::g() through C's vtbl
}
```

The reason for this is that `vtbls` are constructed at compile time when it is not known which object or objects a delegation pointer will point to during the execution of the program.

An object of class `D` will look something like this after `C::p` has somehow been initialized:



9 Constructors and Destructors

Constructors for base classes are called before the constructor for their derived class. The order of execution of constructors of multiple base classes is undefined. Destructors for base classes are called after the destructor for their derived class. The order of execution of destructors of multiple base classes is undefined.

Arguments to base class constructors can be specified like this:

```
class A { A(int); };
class B { B(int); };
class C : A , virtual B {
        C(int a, int b) : A(a) , B(b) { ... }
};
```

When a class has more than one base class *all* argument lists for its base class constructor *must* be qualified with the name of the base class. This rule applies even if only one of the base classes actually requires arguments.

A virtual base class can be explicitly initialized at most once in a derivation lattice. If it is not initialized the default initializer is used.

Assignment to `this` in the constructor of a class that takes part in a multiple inheritance lattice is likely to lead to disaster. This might be the final proof that a better way for a user to take control of memory allocation for a class is needed.

10 Visibility

The examples above ignored visibility considerations. A base class may be `public`, `protected`, or `private`. In addition, it may be `virtual`. For example:

```
class D
        : B1            // private (by default), non-virtual (by default)
        , virtual B2    // private (by default), virtual
        , public B3     // public, non-virtual (by default)
        , public virtual B4 {
        // ...
};
```

Note that a visibility or `virtual` specifier applies to a single base class only. For example,

```
class C : public A, B { ... };
```

declares a public base `A` and a private base `B`.

The usual visibility rules applies to objects delegated to through a pointer. For example:

```
A* pa;

class D
        : *pa           // private
        , public *pb
        , protected *pc {
        C* pc;
protected:
        B* pb;
};
```

The visibility of the pointer delegated through is determined by the declaration of the delegation pointer (and not by the delegation specification). For example, in the example above `pa` is global and therefore globally accessible, `pc` is private to `D` and `pb` is protected.

```
void f(D* pd)
{
        pa = new A;      // fine: pa is global
        ((A*)pd)->f();   // error: delegation through pa is private

        pd->pb = new B;  // error: pb protected
        ((B*)pd)->f();   // fine: delegation through pb is public
}
```

11 Overheads

The overhead in using this scheme is:

[1] One subtraction of a constant for each use of a member in a base class that is included as the second or subsequent base.

[2] One word per function in each `vtbl` (to hold the delta).

[3] One memory reference and one subtraction for each call of a virtual function.

[4] One memory reference and one subtraction for access of a base class member of a virtual base class or of a member of a class delegated to through a pointer.

Note that overheads [1] and [4] are only incurred where multiple inheritance is

actually used, but overheads [2] and [3] are incurred for each class with virtual functions and for each virtual function call even when multiple inheritance is not used. Overheads [1] and [4] are only incurred when members of a second or subsequent base are accessed "from the outside"; a member function of a virtual base class does not incur special overheads when accessing members of its class.

This implies that except for [2] and [3] you pay only for what you actually use; [2] and [3] impose a minor overhead on the virtual function mechanism even where only single inheritance is used.

Fortunately, these overheads are not significant. The time, space, and complexity overheads imposed on the compiler to implement multiple inheritance are not noticeable to the user.

12 But is it Simple to Use?

What makes a language facility hard to use?

[1] Lots of rules.

[2] Subtle differences between rules.

[3] Inability to automatically detect common errors.

[4] Lack of generality.

[5] Deficiencies.

The first two cases lead to difficulty of learning and remembering, causing bugs due to misuse and misunderstanding. The last two cases cause bugs and confusion as the programmer tries to circumvent the rules and "simulate" missing features. Case [3] causes frustration as the programmer discovers mistakes the hard way.

The multiple inheritance scheme presented here provides three ways of extending a class's name space:

[1] A base class.

[2] A virtual base class.

[3] A delegation through a pointer.

These are three ways of creating/specifying a new class rather than ways of creating three different kinds of classes. The rules for using the resulting classes do not depend on how the name space was extended:

[1] Ambiguities are illegal.

[2] Rules for use of members are what they were for single inheritance.

[3] Visibility rules are what they were for single inheritance.

[3] Initialization rules are what they were for single inheritance.

Violations of these rules are detected by the compiler.

In other words, the multiple inheritance scheme is only more complicated to use than the existing single inheritance scheme in that

[1] You can extend a class's name space more than once (with more than one base class).

[2] You can extend a class's name space in three ways rather than in only one way.

This appears minimal and constitutes an attempt to provide a formal and (comparatively) safe set of mechanisms for observed practices and needs. I think that the scheme described here is "as simple as possible, but no simpler."

A potential source of problems exists in the absence of "system provided back-pointers" from a virtual base class to its enclosing object and from an object delegated to through a pointer to the delegating object.

In some contexts, it might also be a problem that pointers to sub-objects are used extensively. This will affect programs that use explicit casting to non-object-pointer types (such as `char*`) and "extra linguistic" tools (such as debuggers and garbage collectors). Otherwise, and hopefully normally, all manipulation of object pointers follows the consistent rules explained in §4, §7, and §8 and is invisible to the user.

13 Conclusions

Multiple inheritance is reasonably simple to add to C++ in a way that makes it easy to use. Multiple inheritance is not too hard to implement, since it requires only very minor syntactic extensions, and fits naturally into the (static) type structure. The implementation is very efficient in both time and space. Compatibility with C is not affected. Portability is not affected.

14 Acknowledgements

In 1984 I had a long discussion with Stein Krogdahl from the University of Oslo, Norway. He had devised a scheme for implementing multiple inheritance in Simula using pointer manipulation based on addition and subtaction of constants. Reference 3 describes this work. Tom Cargill, Jim Coplien, Brian Kernighan, Andy Koenig, Larry Mayka, Doug McIlroy, and Jonathan Shopiro supplied many valuable suggestions and questions.

15 References

[1] Gul Agha:
An Overview of Actor languages.
SIGPLAN Notices, pp58-67, October 1986.

[2] Tom Cargill:
PI: A Case Study in Object-Oriented Programming.
OOPSLA'86 Proceedings, pp 350-360, September 1986.

[3] Stein Krogdahl:
An Efficient Implementation of Simula Classes with Multiple Prefixing.
Research Report No. 83 June 1984,
University of Oslo, Institute of Informatics.

[4] Bjarne Stroustrup:
The C++ Programming Language.
Addison-Wesley, 1986.

[5] Daniel Weinreb and David Moon:
Lisp Machine Manual.
Symbolics, Inc. 1981.

MuX: a lightweight multiprocessor subsystem under UNIX

Christian Tricot

LGI-IMAG
University of Grenoble

Electronic mail: tricot@imag.uucp

ABSTRACT

MuX is a subsystem which allows the multiplexing of a large number (thousands) of processes within a UNIX process. MuX has been defined in order to support the implementation of the OCCAM™ [5] parallel programming language on UNIX machines. This OCCAM programming system developped in our laboratory will be described elsewhere. MuX applications are developped in C. The small size of each MuX context allows efficient switching between a large number of MuX processes. Scheduling is done according to I/O requirement of processes and/or time slicing. An internal clock is also managed by MuX which allows each process to wait for a specific delay. MuX processes are blocked by the MuX kernel, when waiting for input, without blocking the supporting UNIX process. In this paper a brief overview of the UNIX process management is presented. Then we present the MuX kernel, some implementation considerations, and finally a brief description of the application of MuX kernel to build a run-time system for the implementation of an OCCAM compiler.

1. INTRODUCTION

The UNIX kernel[1] deals with two main activities: system calls and scheduling. Each of these, and also the process creation mechanism are sources of troubles when using a large number of interacting processes.

UNIX system calls

When a user program or a UNIX utility requires a service to the UNIX kernel, a system call is generated and the corresponding service identification and parameters are provided to the kernel. Then the processor switches to kernel mode. The user process becomes a kernel process which executes the requested function. When an I/O operation is performed, the kernel process has to wait for its completion. The execution of the process is suspended, and the scheduler allocates the processor to a new process. When the suspended kernel process is resumed, the end of the system call is executed, the processor switches back to user mode, and the kernel process back to the user process. When a kernel process is executed, no interrupt can cause the processor to be switched to the execution of another process, except if the kernel process itself, explicitly makes a call to

OCCAM is a Trademark of the INMOS Group of Companies.

the scheduler. The effect of any I/O interrupt is delayed until the end of the current system call. System calls are frequent and costly operations. The overhead is due to trap mechanism, and context switching. System calls provide services such as process creation, signal management etc. Process control is performed via the signalling mechanism. Each time a process wishes to signal an event to another process, it may loose the processor if a process of higher priority is in the ready queue. So it is impossible to efficiently control the process ordering.

UNIX scheduler

The UNIX kernel deals with process scheduling and process switching. Runnable processes share the processor via a time slicing mechanism. The time slicing is performed by a periodic interrupt. UNIX kernel allows two kinds of interrupts, interrupts initiated by an I/O device, or interrupts generated by the system clock. A user process cannot disable interrupts. When an interrupt occurs, the active user process, is suspended after the completion of the actual instruction, the processor then switches from user mode to kernel mode, handles the interrupt. After the completion of the interrupt handler, the scheduler is entered, it selects for resumption the user process of highest priority. In the case of a clock interrupt, the current user process is normally placed in a list of runnable processes, the highest priority process is then chosen by the scheduler. So due to an I/O device interrupt, a user process can be replaced by another process between any two instructions. User processes cannot know exactly how long time they will own the processor, when they will be replaced by another user process, and after a running period, how long they have been running. Therefore, it is very difficult for a given application to control in a specified manner the ordering of processes. One cannot build any type of process ordering on top of UNIX scheduling. Because of its process orientation and time slicing algorithm, the context switching is frequently performed. It occurs either at the end of a system call, or in response to interrupts that can occur in user mode. A UNIX context contains the process descriptor, different segments adresses (user data, user code, user stack) and an area to save the CPU state. This represents an important cost of memory space.

The process creation mechanism

A new process can be created by a fork system call which makes a copy of the invoking process. The child process is an exact copy of the original one, except that it has a different process identifier. This basic creation mechanism allows any kind of more complex creation schemes, but it causes system overhead due to memory space and CPU time for creating the new process memory image.

UNIX kernel overheads

UNIX scheduling, context switching, and system call mechanism are main causes of UNIX overheads when they are intensively used. Therefore overheads become important when the number of processes of a given application is very large. The UNIX system cannot support such a large number as thousands of processes. It allows only a small number of processes for each user, and some hundreds for the whole system, depending of the system configuration. It must be remembered that weighty processes cost as much as light ones. The process creation mechanism cannot support a high rate of process creation, these processes running for a short time. In this case, the process creation mechanism introduces an increasing overhead. In addition, if these processes are often switched, this also would increase the overall overhead. The whole application becomes totally inefficient. Such applications are more and more frequent with the use of parallel

language compiler in a UNIX environment which requires simulations of parallel execution on a single processor. We were faced to this problem while building a compiler and a run time system for the OCCAM parallel programming language. We need a system with a large number of lightweight processes, and a very high rate of quick context switching, for instant to be able to implement an efficient message passing and synchronization mechanism.

A way to solve this problem is to build a lower layer which multiplex lightweight processes within a UNIX process. We call this multiplexor layer MuX. It has been designed in such a way that it can be used to resolve other crontrol mechanism for UNIX application.

2. THE MUX SYSTEM

The MuX subsystem[10] offers process scheduling, input/output facilities, and clock management services. The primitives of the MuX system have been designed according to four main principles:

- possible explicit ordering of processes.
- offer a new model of lightweight processes which is able to support a large number of processes.
- minimize context switching and process creation overheads.
- avoid any operation that may block the MuX kernel and therefore causes delays for other MuX processes.

The MuX subsystem includes about 2000 lines of C code. MuX applications are written in C. Each process is represented by a small area of memory (used as a running stack), a process descriptor and a set of shared C functions that defines the process running code. Process data are represented as local variables placed in the process stack via the C function data. Global data of the whole C program are shared data between all the processes.

Process management

Each process shares the MuX "processor" via a round robin scheduling algorithm: the process with the highest priority gets the use of the processor first. Process switching is done according to I/O requirements (specific MuX primitives) and/or time slicing. The time slicing policy requires a clock that ticks periodically and counts the running time of the UNIX process. Such a service being available only on UNIX 4.2 BSD, our time slicing clock mechanism is build as a facility independent from the MuX system clock.

MuX offers two levels of process management services: a basic one, which identifies processes directly with a process number, and another one, using a list structure. Process management services are borowed to UNIX. The *new_process* function creates a child process with a programming interface similar to the fork() primitive. The child process receives 0 as return value; the father process gets the child process number. The new process stack contains only a copy of the frame of the function which has called the creation primitive: it will be used as initial context of the new process. This mechanism allows to minimize the process creation overhead. Services such as process killing (*mux_kill*), waiting for the completion of a child process (*mux_wait*), process activation (*mux_activate*), and process interrupt define the basic level of MuX. On top of this basic layer, MuX provides process management by means of process list such as suspending a process within a list (*mux_suspend*) or resuming a process from a list (*mux_resume*).

Clock management

The MuX clock defines a global time for the whole application. When the application requires time management, it must first initialize the clock mechanism by a call to *init_clock* primitive with the clock period as parameter. The basic period is function of the host system (100 micro seconds to 1 second). Two different implementations will be described below. The clock mechanism is performed via a signal handler which receives clock signals from the host system, and a queue for events. Process can wait for either a delay or a date. Event is placed in the queue, the process is reactivated when the date is reached or the delay is over.

MuX system clock also provides another alarm mechanism. A process indicates to the system by a non blocking primitive (*mux_alarm*) an upper limit of time for an alarm. When the clock reaches this limit, the system reactivates the process only if it is suspended. Such a mechanism is useful for implementing time constraint operations such as read with time out.

I/O operation

The MuX kernel manages only the input operations. The output operations are directly transmitted to the UNIX kernel, any other output mechanism would indubitably end with a call to the UNIX kernel to write. We just define some new primitives which write some characters, or a string and leave the MuX processor when terminated. On the other side, an input operation may suspend the execution of the UNIX process when no characters are available, a "buffered-read" mechanism has been implemented. A process requiring for characters adresses the request to the buffer manager, and will be suspended when no character is available. A specific mechanism fills the buffer when characters are available in the file. The standard input file (stdin) systematically uses this mechanism. Applications that require buffered-input files, must inform the MuX system by using the *mux_open* primitive with the descriptor of the opened file as parameter. In addition, this mechanism provides a pre-read primitive (*mux_pre_read*) which returns true when characters are available in the buffer, or false in the opposite case. In this later case, the system will reactivate the process (if suspended) when characters become available in the buffer. This could be used for example, to provide a mechanism which can read in the fisrt ready file among several input files.

3. IMPLEMENTATION

The MuX system is based on two levels: user and kernel. A simple mechanism is provided to switch from user mode to kernel mode (MuX mode) and backward. An interface librairy defines the user facilities. Other primitives can be used when MuX mode is set. Two functions allow the processor mode switch (*mux_to_kernel* and *mux_to_user*) and insure that no external events can interrupt the kernel.

Process definition

The MuX kernel contains a process table describing the state of each MuX process. The process table defines a maximum number of process which depends on a system parameter supplied at starting time. A MuX process has four possible states:

- running in user mode.
- running in MuX mode.
- ready to run.

- waiting.

A process descriptor has the following definition:

```
typedef struct process {
     struct proc *suiv,*pred;  /* link for list management */
     liste_p *head;            /* ptr to the list containing the process */
     int *stackadr;            /* ptr to stack base */
     int st_size;              /* process stack size */
     short pidcreat, mypid;    /* process number of creator, and pid */
     short prio;               /* process priority */
     short state, motive;      /* state and complementary information */
     short childnb;            /* number of existing child process */
     short lastchild;          /* most recently child process number
                                * that has terminated
                                */
     short stamp;              /* process creating time stamp */
     int save_area[SAVE_SIZE]; /* save CPU state area */
} proc_typ;
```

The motive field gives more information on the waiting state (waiting for I/O, waiting for delay, etc). The save area size depends of the host processor type and will contains informations for context switching. The stamp field provides a process ordering to build a synchronization mechanism based on process termination. Since the process turnover may be very important, a father process can terminate before a child process and a new process can be placed in the same descriptor as the father's one. So when the child process will terminate, it will compare stamp field and will detect if its own father is still active or not.

Process context switching

The process switching mechanism involves three MuX functions: *mux_save*, *mux_restore* and *mux_switch*. The *mux_save* function stores in the save_area field of the current process descriptor, the context created by this function on top of the process stack, and all the informations (Stack pointer register,frame context register, etc) which are needed to be able to restore this context in the same area within the stack. The *mux_restore* primitives switches from the current process stack to a new process stack using the save_area field provided as parameter. It reads the informations from this save_area, and build the image of the context of the previous call to the *mux_save* function, in the target stack. It updates the stack pointer register. As a result, it runs with the stack of the new process. The return adress is taken from this new stack and it returns to the restored process. *mux_switch* is the function which performs the selection of a new process and transfers control to that process. It does not save the current process context. According to process behavior, *mux_restore* is a function that never returns. On the other hand, *mux_save* seems to return twice. In fact, a *mux_save* call returns 0, indicating that it is the first return from *mux_save*. *mux_restore* simulates the second return from *mux_save*, it will return 1. These are the only machine dependent functions of MuX. Therefore we have implemented them in assembler. The save area structure depends of the host processor type, and of the context build by the C compiler. Registers are saved to and restored from the stack at link and unlink time. So the save area buffer will contains implicitly saved registers. (80 bytes on VAX, and 54 bytes on 68000 processor machine)

Process scheduling

The switching policy is based on a round robin algorithm. Processes which are able to run are placed in a list of ready processes (*ready_queue*). Processes which have already got the processor, wait for the next time in different list (*priolst*) according to their priority. The MuX system defines a new level of scheduling on top of UNIX. A global system clock (as provided by UNIX) does not allow a fair implementation of a time slicing mechanism. Due to the processor load, the UNIX process supporting the MuX application can get less frequently the processor, and each MuX process will not share the processor for the same period. So a fair time slicing mechanism requires a special clock which runs only when the UNIX process is running.

The UNIX 4.2BSD system provides mechanisms which allow to build such a process virtual clock. This system provides each process with two timers which can be automatically restarted at expiration. The first timer uses real time, and a signal SIGALRM is generated when the timer expires, it can be useful to implement a global clock for the MuX system. The second one uses process virtual time which is local to each UNIX process and a signal SIGVTALRM is delivered when it expires. Two functions allow to handle both of these two timers. The first one allows to set a timer (setitimer), and the second one to read the content of a timer (getitimer).

The time slicing algorithm will be based on that virtual timer. It generates a SIGVTALRM signal after a short period of running time. A signal function handler (*mux_slicing*) save the context of the current process and chooses and tranfers the control to a new process (*mux_switch*). The *mux_switch* function has to set a new period of time for the new process before restarting the process. From the UNIX point of view, the UNIX process switches stack when it handles a signal. In fact, the UNIX kernel when it start a signal handler, saves in the user process stack the state of the process and useful informations for the signal processing but does not modify the process descriptor. Some MuX primitives such as *new_process* must save the timer value for the current process before executing the requested service, and finally restore this value when returning to the process.

Process creation mechanism

The new_process function creates a new MuX process and switches back to the caller process. It is called once, and it returns twice, once in the calling process with the new created process number, and a second time, in a child process, with the value 0. The creation mechanism is based on the following algorithm.

```
short new_process( stack_sz, prio )
int stack_sz;             /* child process stack size */
int prio;                 /* child process priority */
{
     register proc_typ *p;
     int i;
     short npid;
     struct itimerval tt;
     mux_to_kernel() ;
     timer_virtual_save(&tt); /* save content of timer */
     i = mux_save(active->save_area);
     if (i==0) {               /* creating child process */
     /* active process is placed at the top of the ready queue */
          add_first(&ready_queue,active);
          npid = alloc_process(); /* allocate a free process descriptor */
          p = head_process_table + npid;
          p-> adrstack = (int *) malloc(sizeof(int)*stack_sz)
          if (p->adrstack==0) { /* no more memory space */
               mux_error("memory overflow while creating a new process ");
               /* NOTREACHED */
          }
          else {
               p->adrstack = p->adrstack + stack_sz -1;
          }
          p->st_size = stack_sz ;
          initstack(p->adrstack);
          /* save the new process state and switch back to the father process */
          active = p ; active->state = RUNNING ;
          i = mux_save(p->save_area);
          if (i==0) {
               /* _mux_suspend will choose first process in the ready queue */
               _mux_suspend(npid,&priolst[active->prio],READY,NIL);
               /* return to the caller process */
          }
          mux_to_user();
          return(0); /* return to child process */
     }
     else {
          /* this is the caller process, the child process is created */
          active->state = RUNNING ;
          timer_virtual_restore(&tt);    /* restore process quantum to continue */
          mux_to_user() ;
     }
}
```

The *init_stack* function creates the new process stack. It copies from the original stack the frame contexts in order to build a new environment on top of the new stack. Thus, three frame contexts are copied, and links between these contexts are recalculated for the update of the new stack:

- the frame of init_stack function.
- the frame of new_process function.
- the frame of the function which has called the new_process function. This represents the initial environment of the new process.

This function is written in the assembler language.

Clock mechanism

MuX provides a simple clock management mechanism. It uses a UNIX clock which periodically generates a signal. The clock signal handler is processed differently according to the current mode of the running process:

- user mode: it increases the global clock, and resume a process after its delay expiration.
- MuX mode: it does not increase the global clock, but a specific counter which counts the gap between the MuX clock and the true date. When the current MuX processor state will return from MuX mode to user mode, the clock moves forward and recovers the real date.Processes are resumed after the delay expiration.

This provides a mean to prevent any modification of the MuX processor in MuX mode, and to avoid loosing occurrence of clock signaling. This mechanism introduces some delay on the MuX clock and delays some critical time management such as process activating when the waiting period expires. MuX introduce a maximum limit for the clock gap, and if this limit is reached, the system will warn that the system clock is no more synchronized, and will possibly stop the execution.

The real timer as described above is useful to implement the clock mechanism on UNIX 4.2BSD. It allows clock periods from 100 micro seconds to some seconds. On other UNIX system, these possibilities are not provided, and we use the UNIX alarm system call which offers periods up to 1 second. So when a signal is received, the MuX clock is increased differently according to the basic period. The clock period, the clock gap's limit, and the clock increment are introduced at initial time with the *init_clock* call.

Input/Ouput

MuX system offers a "buffered read" file mechanism. It is based on a synchronized buffer mechanism providing two primitives. The read operation gets a character from the buffer (*read_buffer*); the caller process is suspended if the buffer is empty. The write operation puts a character into the buffer (*write_buffer*); the caller process is not suspended if the buffer is full, but the previous character is lost. The buffer structure is as following:

```
typedef struct sync_buffer {
     char buffer[MAXCHAR] ;
     int head,queue;       /* index to put in and get a character
                            * from the buffer
                            */
     short option;         /* process that requires to be alarmed
                            * when a character is available
                            */
     FILE *input;          /* NIL or input file associated to
                            * the buffer descriptor
                            */
     liste_p reader;       /* list of process waiting for characters */
  } sync_buffer_typ ;
```

The "buffered read" mechanism opens a synchronized buffer, attaches the opened file to the buffer and returns the buffer identifier. If a process requires a character from the file, it calls the *mux_getc* primitive with the buffer identifier. If no character is available, the *read_buffer* call suspends the process. Otherwise, the process receives the character and leaves the MuX processor. A specific mechanism fills the buffers which are associated to an open file. Two possibilities have been implemented.

First, on UNIX 4.2BSD, a specific system call (*fcntl*) introduce asynchronous I/O facilities for keyboard handling. A SIGIO signal is sent to the process when I/O is possible (a key has been hit). We use this facility to fill the buffer associated to standard input file. A signal handler is run each time a character is available on the keyboard input, it reads and puts characters into the buffer (*write_buffer*). This primitive activates processes waiting for characters, then the process waiting for a pre-read operation.

Second, another facility provided on all UNIX system is used on other cases. The *fstat*system call returns informations on an opened file, a field (st_size) of the returned structure described the characters available for I/O operations. The MuX system uses this facility to poll each buffered input file descriptor at each process switching time. Then it reads characters on the valid files and put them into the associated buffer (*write_buffer*).

4. EXAMPLE OF USE: A KERNEL FOR OCCAM PARALLEL PROGRAMMING LANGUAGE.

The MuX system provides services that allows synchronization between processes and asynchronous operations with the environment. The MuX system offers a suitable process model to build the run time support for the OCCAM parallel programming language. We present first a brief overview of OCCAM language, a description of our compiler, and then the description of our OCCAM kernel.

4.1. OCCAM language and OCCAM compiler under UNIX.

The OCCAM language[5] has been defined according to Hoare's CSP[4] model to be the language of TRANSPUTER family produce by Inmos[6][2]. The language explicitly provides primitives for concurrency and non-determinism.

The OCCAM language

The OCCAM language is designed around two basic concepts, the process concept and the channel concept. Processes are the basic working objects. Channel is the basic communication element that allows concurrent processes to communicate with each other. A channel can be shared by only two processes, and is unidirectionnal. The OCCAM language has only four elementary processes:

- assignment: **x:= a**
- channel input: **chan ? x**
- channel output: **chan ! x**
- delay: **TIME ? AFTER exp**

Two other basic processes are also available:

- a stop instruction: **STOP** (modeling deadlocks and divergence of a program).
- an instruction that does nothing: **SKIP** (modeling correct termination).

These processes can be combined in order to construct processes using the three OCCAM constructors SEQ, PAR, ALT and some more conventionnal constructors as IF and WHILE.

```
SEQ
    proc1
    proc2
    ...
```

The SEQ constructor defines a collection of processes which are executed sequentially one after each other (if all predecessors of a process terminate).

```
PAR
    proc1
    proc2
    ...
```

The PAR constructor defines a collection of processes running in parallel and terminating if all of them do.

```
ALT
    guard1
        proc1
    guard2
        proc2
    ...
```

A guard can be a delay process, an input process possibly associated with local conditional expressions or a SKIP process. The ALT process starts by evaluating its guards, and waits for at least one guard to become valid. From the subset of valid guards, one is randomly choosen, and the corresponding process is executed.

The iterative process:

```
WHILE expression
    proc
```

defines the classical iterative while statement.

An IF constructor is also provided. It allows only to test sequentially local conditions on processes and behaves like a STOP if the conditions are all false.

OCCAM compiler on UNIX.

The compiler is divided in two parts, a code generator which translates an OCCAM program to a C program, and a run-time system which includes scheduling of lightweigh processes and message passing mechanism. The main decisions to implement the OCCAM compiler have been how to represent the two basic structures, OCCAM processes and channels, and how to support the OCCAM model of processes. As UNIX doesn't provide efficient process management mechanism, we have given up the idea of mapping each OCCAM process to a UNIX process, and decided to introduce our own process management mechanisms. This was the origin of the MuX system. OCCAM processes, as defined in the model, are grouped together in a single MuX process as often as possible. OCCAM constructors which provide sequential behavior between part of their components are mapped to one MuX process. New MuX processes are created only when dealing with the PAR constructor. All the other constructors are mapped to a sequential code in a single process. The OCCAM process term is used hereafter in the paper as a MuX process which describes a sequential behavior of a part of an OCCAM program.

The generated C program is divided in a set of functions which provide either OCCAM processes code, or OCCAM procedures. A global vector mechanism provides acces to variables which are not local to the current function [8]. OCCAM process context is composed of a MuX process context, and an additional vector to allow variable management. The OCCAM program is translated either into C statements for the classical definition part of the OCCAM language, or into OCCAM kernel run-time calls.

4.2. OCCAM KERNEL FUNCTION.

The OCCAM kernel [9] on UNIX is divided in four parts:

- OCCAM processes creation and termination mechanism.
- channel managment (initialization/communication/synchronization).
- access to the environment.
- alternative resolution.

All the primitives of the OCCAM kernel are executed in MuX mode by means of the simple MuX switch mechanism. This allows to use internal MuX kernel functions and provides an efficient implementation.

OCCAM process creation.

The process creation is based on the MuX subsystem. The create_process primitive initializes the MuX process supporting the OCCAM process. The MuX process creates the vector supporting the static links.

Communication on channels.

A channel is identified by a channel number which represents its index in the channel descriptor array. A channel descriptor is composed of:

- suiv: index of next channel number.
- pid_option: process number waiting for a guard to become valid.
- pid_reader: reader process number.
- pid_writer: writer process number.
- value: transmit value.
- wait_lst: a process list to suspend process when communication is not yet available.

Free channels are placed in a specific list. The channel_alloc and channel_free primitives allow to allocate and release channels. Two additional primitives provide the message passing mechanism. *Channel_read* allows a process to read a given amount of bytes from a channel and to assign it to a specific variable, and *channel_write* is used to write an expression on a specific channel. These two primitives verify that no other reader or writer process are already arrived. In this case, it will stop the execution and send an error message to the user. The protocol implementation is quite simple. The writer process resumes the reader process if it is ready, then it is suspended, and will wait for the completion of the reader. The reader process waits for a writer process, then reads the message and resumes the writer process. The ALT resolution introduce alternative process which may read on that channel. The writer process resume the alternative process using the process number in the pid_option field. Then it behaves above and will wait for the resumed process to read on the channel. The alternative process as we will see later will choose a valid communication and behaves as a reader process (call to *channel_read* primitive) and executes the guarded process. A short description of read and write primitives is as follows.

```
channel_read (....)
{
     mux_to_kernel();
     /* ptr is a pointer to the channel */
     /* place process number in channel pid_reader field */
     if ( No ready writer process on the channel)
     {
          _mux_suspend(mypid(),&ptr->wait_lst,WAITING, CHANNEL_READ);
          /* the process is resumed by a writer */
          /* reads the channel value field, and terminates communication step */
          /* resumes writer process */
          _mux_resume(&ptr->wait_lst);
     }
     else
     { /* a writer process is already present */
          /* reads the channel value field and terminates communication step */
          /* resumes writer process */
          _mux_resume(&ptr->wait_lst);
          /* leaves the processor, and suspend for the next round */
          _mux_pause();
     }
     mux_to_user();
}


channel_write ( ...)
{
     mux_to_kernel();
     /* ptr is a pointer to the channel descriptor */
     /* place process number in channel pid_writer field */
     /* stores the value */
     if ( a reader process is ready)
     {
          /* resume the reader */
          _mux_resume(&ptr->wait_lst);
     }
     else
     if ( a guarded process is waiting for an input
          on that channel (using pid_option field))
     {
          /* activates it and the guard becomes valid */
          /* as we do not know in which list it is suspended,
           * we activate it by its number
           */
          _mux_activate(ptr->pid_option);
     }
     /* After that, we leave the reader complete the communication */
     _mux_suspend(mypid(),&ptr->wait_lst,WAITING,CHANNEL_WRITE);
     /* the reader process resume us, the communication is complete */
     mux_to_user();
}
```

Access to the environment.

The outside world is viewed from an OCCAM program as a set of channels which are mapped to facilities of the host system. These channels allow the OCCAM program:

- to obtain parameters from the shell command.
- to read from and write to the console.
- to open, close, read and write files.

The Screen channel is a simple buffer mechanism, and allows access to stdout file. The Keyboard channel allows a single OCCAM process to read stdin file. It uses the stdin synchronized buffer as provide in MuX system.

Other standard files are accessible to OCCAM via file handlers which handle one file at a time. Each file handler is connected to the OCCAM program by two specific dual channels, the first one for reading from (filein), the other one for writing (fileout). The file handler implements the protocol defined by the Inmos implementation[5]. When the file is opened for output, the program writes characters or control values on the fileout channel and reads the return value from the filein channel. When the file is opened for input, the OCCAM program writes command values on the fileout channel and reads characters or control values on the filein channel. File acces channels are implemented by a simple buffer for the fileout channel and by a synchronzed buffer for filein channel. Control values sent on fileout channel are interpreted and return values are placed in the filein buffer. The synchronized buffer takes into account the process management and allows a simple read file mechanism, it is supplied either with the return values by the file handler, or by the filer mechanism from the input associated file when the file handler is opened in input mode.

Alternative handling.

OCCAM provides two versions of the alternative constructor[Dij..]. The PRI ALT constructor gives the priority to its first valid guard. The ALT constructor provides a non deterministic choice between all the valid guards. These constructors may be nested to control procedures selection. If no guard is valid, these two constructors have to suspend the process and have to wait for a guard to become valid. Parts of the optional mechanism that the MuX system provides are very useful to implement these constructors. The ALT resolution is build in three phases.

First, we build an pre-evaluated tree. Nodes are either guards, or alternative contructors. The local conditions are already evaluated and the guard node indicates true or false, or a channel number where the process waits for a possible communication, or for a delay. The node contains a pointer to the guarded process function.

Second, the tree is completly evaluated. Each guard is evaluated using the following method: The evaluation algorithm is divided in two steps:

- no valid guard has been already evaluated.
- a valid guard has been detected.

First from the root of the tree and until a valid guard is detected, OCCAM kernel will read nodes and place some alarm and optionnal read as provided by the MuX system, or put an option mark on a channel.

As soon as a valid guard is detected, OCCAM kernel removes all the option marks already placed and then memorizes the valid guards.

After the evaluation, if no valid guard has been detected, the process is suspended and will wait for an option to become valid. When the process restarts, it will detect all the true guards (possibly several guards become valid nearly at the same time).

Third, at the end, it will choose the smallest priority guard among the valid guard list and executes the function which realises the OCCAM guarded process The priority is

managed as following: the nodes directly under an ALT construct node are of the same priority as the constructor node itself. Nodes directly under a PRI ALT constructor get less and less priority. The first node gets the same priority as the constructor node, and the following nodes will receive a priority which is increased from the previous node priority.

Part of the evaluation algorithm is as follows:

```
node_evaluation(node)
node_typ *node;                 /* pointer to the node to valuate */
{
     switch (node->node_type)       /* study of the node type */
     {
     case GUARD:                    /* Guard node */
          switch (node->expr)       /* study of guard contents */
          {
          case TRUE_EXPR:           /* expression evaluated to TRUE */
               value = TRUE;
               break ;
          case FALSE_EXPR:          /* expression evaluated to FALSE */
               value = FALSE;
               break;
          case TIMER:               /* guard TIME AFTER */
               value = (_mux_time() > node->date) ;
               if ( time is not already over and still in first step)
               {
                    /* place an option */
                    _mux_alarm(node->date);
               }
               break;
          case TIME:                /* guard with TIME channel:
                                     * always ready to communicate
                                     */
               value = TRUE;
               break;
          case FileIn0:             /* Input channel associated with file handlers */
          case FileIn1:
          case FileIn2:
          case FileIn3:
          case FileIn4:
          case FileIn5:
          case FileIn6:
          case FileIn7:
               /* place an option in the synchronized buffer
                * if no character is available
                */
               value = _mux_pre_read(node->expr);
               break;
          default:
               /* test valid channel number, otherwise stop */
               /* test communication on that channel */
               value = communication (node->expr) ;
               if ( not ready and first step )
               {
                    /* set option pid in the channel descriptor */
               }
          break ;
          }
          if (first step)
          {
               /* no valid guard has been evaluated */
               if (value)
               {
                    /* first valid */
                    /* remove all the option mark aready
                     * placed and memorized in the list
                     */
```

```
                    remove_options();
                    /* add this first valid guard in the valid list */
                    add_valid(node) ;
                    step = SECOND ; /* begins the second step */
                }
                else
                {
                    /* add a option in the option list */
                    add_option( node);
                }
            }
            else
            {
                /* second step: memorize valid guard only */
                if (value) /* the guard is valid */
                    add_valid(node);
            }
            if (node->guard_suiv != NIL) /* evaluate the next guard */
                node_evaluation(node->guard_suiv) ;
            break;
        case PRIALT: /* constructor nodes */
        case PRIALREP:
        case ALT:
        case ALTREP:
                /* evaluation of the sub-tree guards */
                node_evaluation( node->child);
        }
    }
```

The control evaluation algorithm is simply described by:

```
    evaluation (head)
    {
        mux_to_kernel();
        /* start evaluation */
        node_evaluation (node) ;
        if (valid list is empty */
        {
            /* suspend the process */
            _mux_suspend(mypid(), alternative_list, WAITING, ALTERNATIVE);
            /* resumed by a process */
            /* detect valid guards (may be several) and
             * placed them into the valid list
             */
        }
        /* choose a valid guard from the list */
        /* and run the function which realises the chosen guarded process */
        mux_to_user();
    }
```

This mechanism based on an evaluated tree introduces some overheads in the ALT resolution, but it allows different schemes of evaluation and choice, as for example a non-deterministic choice between the valid guards. It may be extend for OCCAM program observation, and program debugger tools.

5. CONCLUSION.

The implementation of OCCAM described above proves the adequation of the process management facilities of MuX as well as the management of requests with temporal features. MuX has also been applied to a quite different domain: the distributed management of Smalltalk objects. This object manager has been implemented using MuX on top of UNIX BSD 4.2 [3]. Here, MuX provides the object manager with facilities for shared data and for process management. Our short term plan is to integrate a debugger that would allow the implementer to observe the status of his running processes. Our OCCAM programming system will be extended with a new construct[7] to take into account the particular case of environmental system programming, and the MuX system will allow preliminary experiments to show that the new functionnality can be achieved at a reasonable cost.

Our early experiments show that MuX offers an easy way to program parallel applications while not degrading execution time too much.

Acknowledgments.

I would like to thank Traian Muntean (Imag/LGI) and Gerard Vandome (Bull research centre) for their support, guidance and their help with the English syntax and style.

REFERENCES

[1]. M.J.BACH, 'The design of the UNIX operating system', ATT Prentice/Hall .

[2]. I. M. BARRON, Inmos 'The TRANSPUTER and OCCAM', IFIP congress, Information Processing, 259-265, (1986).

[3]. D. DECOUCHANT, 'Design of a distributed object manager for the smalltalk-80 system', OOPSLA 86 congress, 444-450 (1986).

[4]. C. A. R. HOARE, 'Communicating sequential processes', Comm. ACM, 21 (8), 666-677 (1978).

[5]. Inmos Ltd., 'OCCAM programming manual', Prentice/Hall (1984).

[6]. Inmos Ltd., 'TRANSPUTER reference manual', (1985).

[7]. T.MUNTEAN, M.RIVEILL, 'An extented OCCAM model for timed parallel systems', IAS congress, 470-479 (1986)

[8]. C.TRICOT, M.RIVEILL, 'Compilateur OCCAM.UX: structure du code C genere', research report to be published, IMAG/LGI (1987).

[9]. C.TRICOT, 'Definition d'un noyau OCCAM sur UNIX', research report to be published, IMAG/LGI (1987).

[10]. C.TRICOT, 'MuX, un gestionnaire de processus sur UNIX', research report to be published, IMAG/LGI (1987).

Using a Unix Engine as an Intelligent Information Server

Martin D. Beer

Department of Computer Science,
University of Liverpool,
P. O. Box 147,
LIVERPOOL
L69 3BX.

Electronic Mail: SQMBEER @ UK.AC.LIVERPOOL.CSVAX

ABSTRACT

The philosophy of the intelligent information server is explained in the context of Office Systems. Particular emphasis is given to the sources and organisation of typical office information, and to the applications to which it is put. The advantages of using a variety of cheap workstations, connected to a series of powerful servers are discussed.

Experiences with a prototype server are described, with particular reference to the first pilot applications implementations:

- a scientific paper generator and
- an improved visual interface to a database.

These pilot implementations have been prototyped using the readily available Unix toolset at the information server end, showing the power and flexibility of the Unix approach. The initial results of these pilot studies are fully discussed, and indications are given as to the future direction of the Intelligent Information Server project

1. Introduction

The falling cost of computer hardware allows data processing capacity to be purchased and used wherever it is required in the modern office environment. This has led to the development of cheap, but extremely powerful personal computers. Not only the hardware, but also the software available to the manager at his desk, allows the manager to build a powerful decison support tool with ease.

There is no need for the typical personal computer user to know how to program, or even how to use an operating system. This is because a whole galaxy of software packages are readily available for all the jobs for which he is likely to want to use his computer. These can be catagorised into the following applications areas:

a) spreadsheets

b) small database applications

c) word processing and text processing

d) the preparation of graphical and other presentation type displays

e) accountancy and similar specialist applications.

These are usually independent programs which makes it difficult not pass information from one to the other. The flexibility provided by being able to configure one's own work environment does however compensate greatly for this. Users like the feelings of security and of controlling their own computing resources that the microcomputer revolution has brought them. This is primarily because of the availability of cheap, mass produced hardware and software for all of the most common office tasks.

It is also awkward to communicate with other information providers, except in the most primitive ways. It is still highly desirable however, to have immediate access to the most accurate and up-to-date corporate data available. This is best organised by a centralised data processing department with all the staffing and other resources necessary to provide such a service.

The solution usually adopted as a compromise between these two conflicting requirements is to install a computer network which connects distributed microcomputers to central mainframes [1]. Both the connection and hardware interface costs tend to be quite high, and a sophisticated microcomputer system is reduced to being a terminal with at best, data capture and file transfer capabilities. There is no other way of ensuring either the accuracy or currency of any data that has been transferred for storage on the microcomputer system.

This paper describes an alternative which combines the advantages of maintaining effective central databases with the requirements of distributed data processing. The physical and economic difficulties of network access can be greatly reduced by connecting the personal computer to local information servers by means of asynchronous serial lines. The information servers can then be connected to each other by means of the network [2]. Since the information servers are permanently coupled, difficulties with communication with workstations that are temporarily disconnected from the network do not arise [3,4].

Early thoughts were concerned with the problems associated with the distribution of files required by students performing practicals on microcomputers [5]. Whilst file management is quite practical on a typical microcomputer network [6], problems were encountered in attempting to collect completed work in from students electronically as it was found impossible to ascertain which student had originated the material to be marked. The requirement was therefore for a database structure to collect material presented for marking and to authenticate it. Initial designs for such a system, which has not so far been implemented, formed the basis for the Intelligent Information Server.

2. The Prototype Information Server

Information has a granularity that is often difficult to represent by splitting it into separate files. Several attempts have been made to organise groupings of information objects [7] so that they can be accessed in a much more natural way. The most notable examples of this has been the SMALLTALK project, based at Xerox PARC [8,9]. The Intelligent Information Server allows material to be collected from wherever it is best made available. This does not have to be in the form of files, but as individual data entities that will be collated and presented to the user's application program in the form of a single file.

The intelligent information server is intended to give a uniform means of accessing all the data available to the user of the system. This will include data that is stored:

a) on the workstation

b) in the information server and

c) on other machines on the network.

In many cases information from more than one of these sources requires merging before presentation to the user as if it has been drawn from a single source. It is not sufficient to give the user access to each of these information sources separately. They must be integrated into a single overall workstation environment. Since this environment must cater for the information processing requirements of many different users, each with very different capabilities and expectations, it must be extremely flexible.

The intelligent information server has a number of facilities aimed at correlating existing information sources and providing common processing:

a) local integration of personal software

b) import/export of data from the workstation in a compatible format

c) a file server locally which can respond to remote requests for data without the need to access personal workstations.

d) levels of security, for both local and remote data access.

The design of the intelligent information server makes it look like a 'virtual user' to the workstation. That is, it reacts in the same way as the real user to requests for data input and instructions. It may also initiate activity on the workstation, either by reformatting data or fulfilling a remote query.

It is essential for any office system to maintain the security and integrity of the information held within it. The division of functions between the workstation and the information server (and indeed between information servers) does much to assist this. Sensitive information can be separated and spread as appropriate between servers and workstations. It should then be possible in most environments to ensure that information is sufficiently separated to ensure that no single part of the system holds sufficient information to be of use to a third party.

For systems that include particularly sensitive information, the key data can be held on the workstation, on exchangeable media such as floppy disks. It can then be removed from the system when it is not required. The user is then required to make his own backups and maintain media security as with a conventional system. The particular advantage of the information server in this context is that by separating data storage, it is only really necessary to take protective action over the really sensitive data, which is usually only a small part of the total data storage requirement [10].

The separation of information management functions from the workstation:

a) increases security, because there is less risk of unauthorised intrusion - the workstation and network servers can act as sentries as well as access points.

b) allows the full power of the workstation to be made available to the user.

c) simplifies the user's interface to the office system and removes the need for routine housekeeping.

d) ensures that the latest available information is always provided to the manager.

e) the information server remains connected to the network whether the associated workstations are in use or not. This allows information to be transferred to other authorised users at any time. Equally the information server can collect information when it is available and collate it, ready for presentation to the manager when he returns.

3. The Results So Far

In the academic environment there is much less emphasis on shared data than is usual in corporate environments. Both academic staff and research students do however spend a considerable proportion of their time in individual document preparation and in managing their own research data. The first prototypes have therefore been designed for the following applications:

a) scientific document preparation and

b) database administration

where it is anticipated that the information server approach will have major benefits. In both cases there have been problems in encouraging staff that are not "experts" in the particular software, that is:

a) the standard Unix documentation tools:
   - the *troff* phototypsetting program
   - the *ms* macro set
   - the *tbl* table preprocessor and
   - the *eqn* equation preprocessor [11]

b) the database pilot study was based on entering and retrieving data from databases maintained by *Ingres*.

Since the implementation of each pilot study had to be restricted to what was practical in a single undergraduate student project, it was decided from the start to concentrate development effort at the workstation end, and to rely on the standard Unix utilities to implement as many Information Server functions as possible.

A first prototypes have been been implemented using a High Level Hardware Orion (running Berkley 4.2 Unix) as the information server, and Atari 1040 STs acting as the workstations. All communications are currently over asynchronous lines.

Two different approaches were taken in each of the pilot projects, so that different strategies for user-workstation-server interaction could be evaluated. These were:

a) in the case of the document handler, analysis of intended usage, showed that the primary need was for a system that was essentially portable, in that drafts for documents could be downloaded onto a "portable workstation", or more likely, onto a floppy disk that is compatible with a home workstation. The interaction between the workstation and the server is therefore restricted to:

  - raw documents that are transferred down to be worked on later, and
  - final documents that are uploaded to be printed on high quality printers, or to be transmitted electronically.

b) in the case of the database handler, the requirement was to ease the problems associated with formulating queries, and in developing the capability of integrating the results of successful queries into other documents. To this extent, the database handler, must be expected to interrelate with the document handler. It therefore became obvious at an early stage in the design that the database handler required online access to the server to function successfully.

These two applications are therefore useful in developing the concept of the information server as they represent quite different user aspirations.

3.1. The Document Handler

This software was the most urgently required of the two pilot projects, as there were immediately several potential users. Three problems with the current methods of editing and text processing on a single computer were quickly identified:

- the need to edit and preview documents on the larger Unix machines, which was often not convenient, particularly when members of staff needed to work at home, or in parts of the building which were not directly connected to a particular machine.
- the need to learn yet another different collection of unintelligible hieroglyphics that behaved differently, depending on which machine you were using at the time.
- the inability to obtain draft output whenever required, whether connected to the server or not.
- most people are now becoming used to editing documents viewed in their final form on-screen. They do not like having to modify text files that bear no relation to the final output.

Further investigation showed that proofs are used mainly for correcting the content of documents and their overall look, rather than to sort out particular layout problems. It is therefore unnecessary for proofs on dot matrix printers and the basic editing screen to show exactly what will appear on the final printed page, so long as the essential features are present. In the pilot program for instance, no attempt is made to place page boundaries on the editing screen, since these will invariably appear in different positions in the final document.

The program as it currently stands, allows the user to enter titles and section headings by means of dialog boxes which control the levels. Information, such as the author's name and address, which does not vary very often is collected from a simple database, so that it does not have to be entered for every paper. Once headings have been defined, text are entered from the keyboard, in the same way as with a normal wordprocessor. Paragraph starts are selected from pull-down menus.

The document is loaded and saved to disk as a straightforward text file that after transferring to the server using the Kermit [12] file transfer protocol, can be entered directly into the phototypsetting program. This simple user interface has proved extremely popular with potential users, as it hides most of the wizardry required to use the standard system. It is intended to develop the basic system further, using the same principles, to allow the user to layout tables and mathematical equations in a similar way.

Recent development have been in the layering of the user interface [13]. This will allow users to collect sections of a document from different sources, and collate them into a single document. This will allow outlining and dynamic reorganisation of the document on the workstation screen.

The next stage in development is to replace the document file with a template that indicates to the workstation and the information server where the various parts of the document are to be found. These entities can be text, tables or equations which can be stored on either machine. It is also intended to start work on a diagram and graphics preprocessor which will allow diagrams and figures to be designed on the workstation and integrated naturally into final documents. There is already limited capability to convert data objects as they are transferred between machines, and this will be expanded so that diagrams can be drawn using a standard drawing package, but stored on the information server in a form compatible with the other data objects that will make up the final document.

3.2. The Database Handler

As already stated, this study involved the development of a different type of application to the document handler. Whilst the real requirement is for a distributed database system, it was impractical to implement this at the pilot stage. The specification for the pilot system was therefore for a database front end, which although running on the workstation, would manipulate information held within the information server's database. The objective was to provide a simple visual interface that would ba attractive to "naive" users [14].

This program has now been developed and has the following functions:

a) a text editor for queries,

b) a help menu,

c) predefined queries and

d) output of the results of queries either to the screen, or into files on the workstation. These files are in a form that can be integrated into a wordprocessed document.

The Query Editor is a full function editor, which includes all the functions you would expect from a modern screen editor. The query text can be loaded and saved on disk by the workstation, and is transferred to the server only when it requires evaluation. A macro facility has been included, so that complex queries can be built up quickly and easily from pre-written sub-queries.

The current database handler will only function with data held within the *Ingres* database. Further development is required to integrate data held on the workstation into a form that a user can use. The results of queries are already translated into a form that can be conveniently by included into documents prepared using the document handler.

4. Conclusions

The pilot projects have just been completed so that evaluation is still at a very early stage. It is intended to make the programs as they are currently functioning available reasonably widely within the department, over the summer. This should provide a varied (and extremely critical) user base.

The text processing system in particular, is already developing its own user base, even though the facilities available in the current program are but a small fraction of the design aim. This is extremely encouraging as it does represent the minimum capability of the Information Server.

The decision to use the standard Unix utilities at this stage has been fully vindicated in allowing both pilot studies to be completed in the time available. It would be desirable for the workstation to be integrated more closely with the Information Server so that the user does not need to know on which computer the information he is currently using is stored. This is likely to be the subject of several more projects.

5. Acknowledgements

The programming of the document and database handlers described in this paper was undertaken by G. P. Pagett (the document handler) and M. Laybourne (the database handler) as part of their final year projects, under the supervision of the author.

Many of the ideas behind the intelligent Information Server were worked out by the author, in conjunction with his colleague, *Brian C. Walsh* , who was tragically killed in a car crash last summer before he could see the results of his work.

6. References

[1] J-C. Lienard, "Open Networking in a Closed Environment", Conference proceedings, Networks'86, London, Online, (1986), pp149-162.

[2] H. S. Spurney, "The Reality of Standards in Local Area Networking", Conference proceedings, Networks'86, London, Online, (1986), pp439-447.

[3] M. D. Beer and B. C. Walsh, "An Intelligent Information Server", Internal Report No CSR 85/5, Department of Computer Science, University of Liverpool (1985).

[4] M. D. Beer and B. C. Walsh, "An Intelligent Information Server for Office Computer Networks", Conference proceedings, Networks'86, London, Online, (1986).

[5] M. D. Beer, "File Management in Practical Classes", Contribution to "Trends in Computer Assisted Education" R. Lewis and E. D. Tagg, *eds.*, pp169-173, Blackwell Scientific Publications, Oxford (1987).

[6] Acorn Computers Ltd., "The Econet User Guide", Acorn Computers Ltd. (1982).

[7] R. Filkes and T. Kehler, "The Role of Frame-based Representation in Reasoning", CACM, 28 (9), pp 904-920, (1985).

[8] A. Goldberg, "Smalltalk-80: The Interactive Programming Environment", Addison-Wesley, (1983).

[9] J. Dohahue and J. Widom, "Whiteboards: A Graphical Database Tool", ACM Trans. on Office Systems, 4 , pp24-41 (1986).

[10] M. Shain, "Micro-Mainframe Security", Conference proceedings, Networks'86, London, Online, (1986), pp473-481.

[11] B. W. Kernighan, "The Unix Document Preparation Tools - A Retrospective", Conference Proceedings, PROTEXT I, Dublin, (1984),

[12] F. da Cruz and W. Catchings, "Kermit: A File Transfer Protocol for Universities", Byte, June and July 1984.

[13] J. Courtin, I. Kowarski and C. Michaux, "MIDOC: An Integrated Interactive System for Structured Editing", Conference Proceedings, PROTEXT I, Dublin, (1984), pp 120-125.

[14] J. W. Davidson and S. B. Zdonik, "A Visual Interface for a Database with Version Management", ACM Trans. on Office Systems, 4 , pp226-256 (1986).

A UNIX SVID Compatible System Based on PXROS

Dr. Rolf Strothmann,
HighTec EDV-Systeme GmbH.,
St. Johanner Str. 38,
6600 Saarbrücken,
West Germany.

Abstract

It is shown that an efficient UNIX SYSTEM V (SVID) compatible system can be built using the realtime operating system PXROS. Because of the portability and modularity of this operating system, this system is at least as portable as the UNIX system itself and can easily be implemented on a multiprocessor hardware. While the standard UNIX operating system exhibits a monolithic architecture PXROS/SV offers a standard SYSTEM V kernel interface on top of a network of PXROS tasks. The modularity of the system even allows the construction of small diskless systems offering a reduced set of UNIX compatible services.

Reduced Operating System Interface

A typical realtime system consists of tasks which can roughly be divided into two classes - asynchronous and synchronous tasks. Some of the tasks deal with external events and the inherent asynchronism and their structure has to reflect this. Driver tasks are usually members of this class. The tasks of the application system itself mostly operate in a synchronous manner like standard UNIX tasks do. Writing a synchronous task is much easier than writing a driver and thus less qualified programmers can do this job. On the other hand hardware specific tasks have to be tested within the target system while a typical application task is fully testable within the development system itself. The integration of such a task in the target operating environment is greatly simplified if the application tasks are be just the same in both environments. This can be achieved by making their operating environments

identical. Further analyzing application tasks one can find that only a small number of system calls like open, close, read, write, seek etc and a small number of sequentially accessed files are used - mostly as data channels. Many of these programs behave like UNIX filters and thus can be written in such a manner. Integrating the respective system calls into the target operating system objects generated and tested within the development system can be moved to the target system without any modification or linkage. PXROS's ability to offer arbitrary operating system interfaces is the basis for the simplification described.

PXROS Overview

PXROS is a processor-independant multitasking realtime operating system. It was developed for single- or multiprocessor-systems using 8-, 16- or 32-Bit micro-processors. The modularity of PXROS allows an individual configuration depending on the requirements of an user.

Under all system variants the user can apply the same tools to solve his problems whether these are simple or more complex realtime problems.

A PXROS system consists of a certain number of more or less independantly working tasks. PXROS itself is a system of tasks implementing the operating system's services. The user can define tasks, e.g. sequential programs or specific device driver, and then run them on the system. Tasks may be written in any suitable high level language, e.g. C or PASCAL. A sample PXROS system is shown in following figure.

PXROS features:

* PXROS is written in the high level language C (except a few lines of code) to allow portability and easy maintenance.

* Independant tasks can simultaneously execute under realtime conditions on one or more processors.

* User and system tasks may be written in high level languages.

* Tasks are dynamically configurable.

* PXROS offers powerful and efficient tools for communication and synchronization.

* User tasks can address I/O-devices via inter task communication.

* Scheduling is efficiently supported by a variable number of priorities.

* PXROS is suited for systems with extremely time-critical requirements.

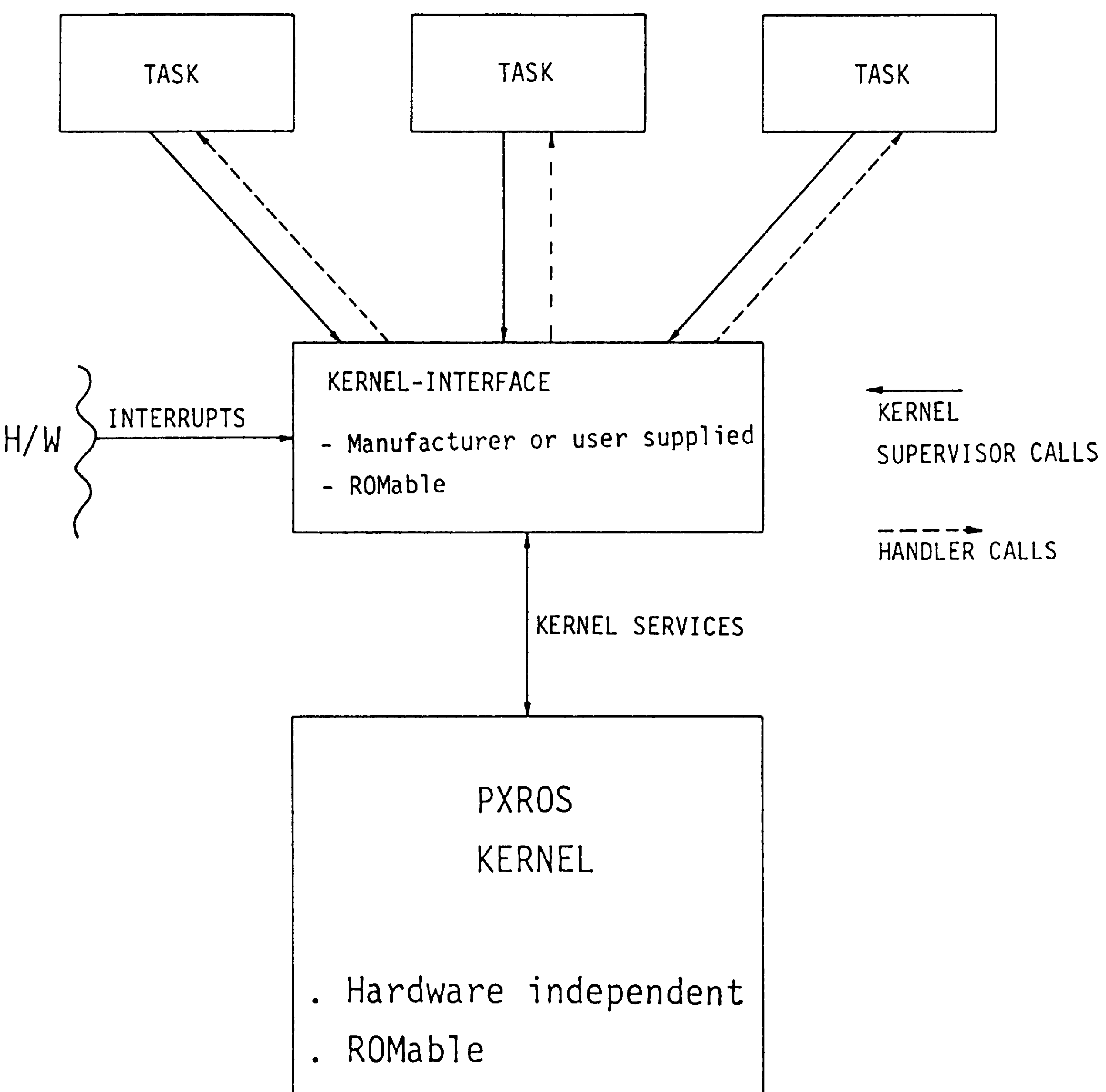
* A vectorized interrupt mechanism allows very fast reactions to external events.

* An universal mechanism is used to control timeout and other exception conditions.

* The memory management provides special services to dynamically allocate and release storage of different storage classes.

* Tasks can use common libraries.

FIGURE 1



PXROS KERNEL INTERFACE

* A general and easy to handle powerfail mechanism is supported.

Tasks and Intertask Communication (ITC)

A task is a regular sequential program together with its data, which can execute independently from other tasks. In this sense tasks are the fundamental concept of structured programming.

There is no strict separation of user and system tasks per se. The distinction can be made according to the services the tasks provide of the capabilities needed.

It is possible to introduce a strict and secure distinction between user and supervisor. In such a user/supervisor relationship the supervisor-task is the only interface to the rest of the system. This kind of communication is implemented by a special kind of ITC mechanism, the supervisor call.

The calling usertask becomes inactive while the supervisor can check and execute the demanded services.

Events are a further tool for performing ITC. A task can signal events to another task or a group of tasks, put together in a so called event group.

Events are "task-specific" variables. Each task has 16 predefined events. A task is able to ignore events or can perform a selective break, where the actual calling sequence is regularly terminated.

PXROS offers a message passing mechanism to send data from one to another task. It is possible to use both synchronous and asynchronous message transfer. PXROS also offers a mechanism which avoids any copying of messages on the same processor, thus being very efficient in a timecritical environment.

The remote procedure (RPC) call is a very "comfortable" form of ITC. It is used similarly to a regular procedure call with parameters. The called task performs the demanded service, while the calling task remains inactive until the actions are completed and the remote procedure returns.

Asynchronous initiation of code specified by the target task is done with the help of a socalled softinterrupt call (SIC). This mechanism extends the functionality of the RPC.

The PXROS Kernel and Kernel Tasks

The system kernel is a small unit independant of the actual configuration. It provides the basic operating functions which allow the implementation of all higher services.

Most of the services of the system kernel are used by system tasks, the so called kernel tasks.

The system kernel and the kernel tasks form the basic PXROS system.

The task management under realtime conditions is part of the system kernel and provides the following services:

create a task
kill a task
change the priority of a task
modify the current state of a task.
wake up a task.

All interrupt handling is done by tasks. The system kernel is responsible for:

interrupt level handling
propagation of external interrupts
scheduling.

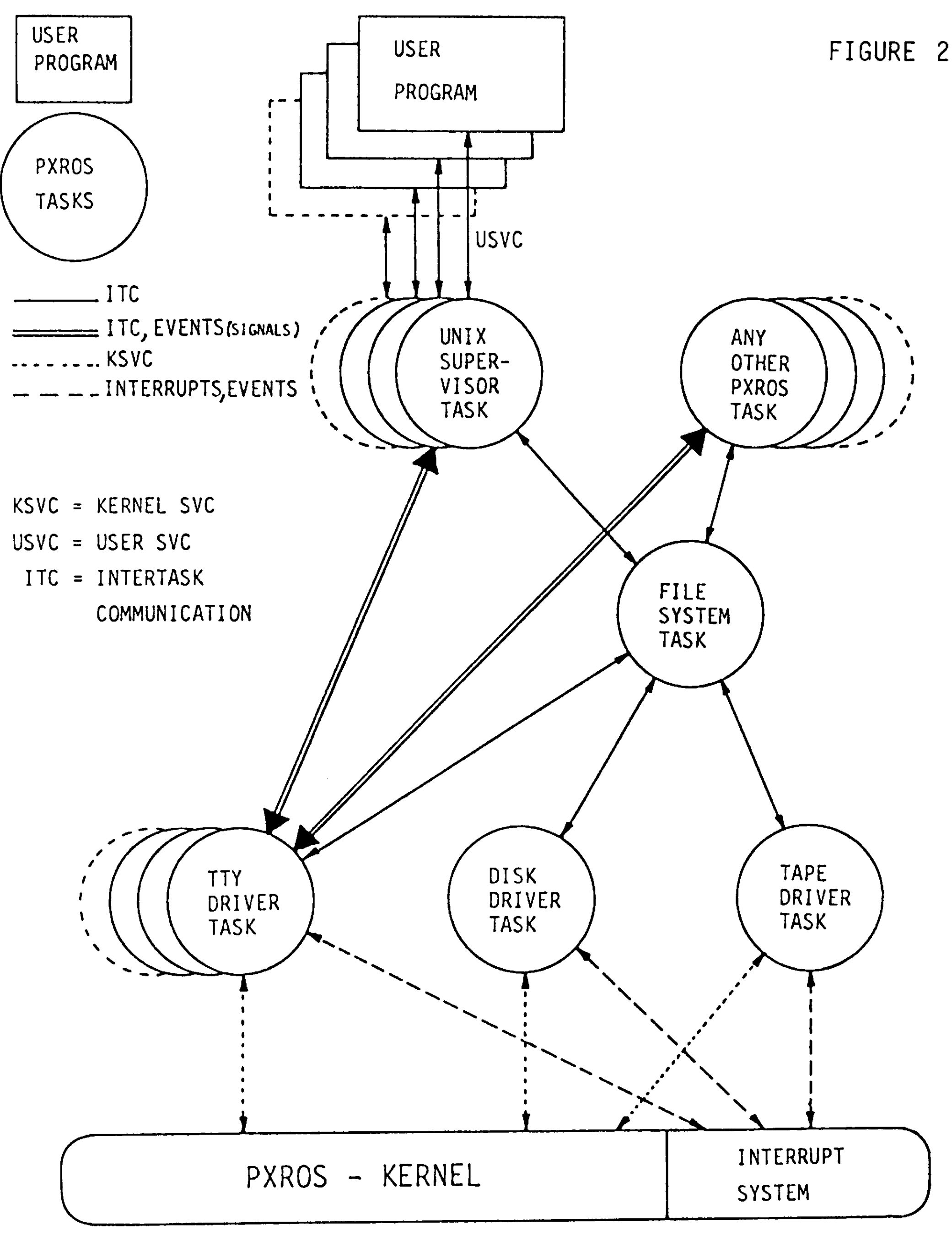
PXROS offers general tools for timing and exception control. A task can control actions under specific time conditions. The time control signals TIMEOUT events in the case of an timeout. Then the task itself is responsible for error handling. Since device driver are tasks it is their responsibility to handle errors or reinitializations where the hardware is not able to meet the time protocol requirements. The same applies to other exceptions like power fail handling.

In this way the time control of I/O operations can be performed by the driver tasks. Most of these errors produced by hardware can be handled by the driver task, thus hiding specific characteristics of a device. This means a complete virtualization of the device.

The memory management provides dynamic generation of data objects and tasks at runtime. This concept is based on a general memory management involving storage classes. The definition of a storage class depends on the individual system. Different storage classes are managed by different tasks.

Structure of PXROS/SV

On basis of the PXROS kernel a number of tasks provide the services needed to grant the request of a UNIX System V application program. The following figure 2 gives an overview of the system structure.



PXROS-SV - a sample PXROS system structure

The figure may only show part of a much larger system which just contains the UNIX V interface as part of it. Even different system interfaces may be coexistent. In the following the functionality of the shown tasks is described.

Supervisor Task

The most important task for the construction of the UNIX System V interface is the supervisor. This task embeddes the user supplied UNIX program and is responsible for the transformation of UNIX calls into suitable kernel services or services provided by other PXROS tasks. The supervisor provides additional structures and mechanism needed for the PXROS environment. In fact the supervisor is an extension of the user program running in a privileged mode. Before the startup of the UNIX program the supervisor is created with the help of PXROS services and then the supervisor incorporates the UNIX program into its own address space.

Synchronous UNIX Calls

The communication between the controlling supervisor and the program controlled by it is normally initiated by a SVC executed by the user program. The supervisor code is called after such a SVC and it has full access to the program's address space and resources. The request is transformed into a PXROS message, sent to the suitable task, i.e. the filesystem task, and termination is awaited. Because a message is basically a descriptor of a piece of memory any mapping of virtual address spaces can be done using this descriptor. The PXROS message mechanism does not depend on the localization of the communication partners and thus the filesystem task may run on a different processor or subsystem. Figure 3 shows the relevant piece of code for the transformation of a UNIX read SVC.

FIGURE 3

```
/*
 * the supervisor calls the filesystem interface routine uf_op with parameters
 * u    : points to the structure describing the user program
 * READ : Code for the read operation
 * a[0] : filedescriptor of read system call which is locally mapped
 * a[1] : buffer address of read
 * a[2] : length of operation
 */

r0 = uf_op(u,READ,map(a[0]),a[1],a[2]);

/*
 * the interface routine which transforms the system call into a message,
 * sends the message to the filesystem task and awaits the termination.
 */
uf_op(u)
struct user *u;
{
        char    ufs_flag;                  /* flag needed for flagwait */
        token_t ufs_token;                 /* message descriptor */
/*
 * the set_tb kernel service creates a message descriptor for the memory
 * starting at the address of the parameter u with a length of 5 double words.
 * no uf_op call in the supervisor has more than 5 parameters.
 */
        set_tb(&ufs_token,&u,5*sizeof(int),&ufs_flag,UNCOND);

/*
 * ufs_mbx is a static variable containing the id of the filesystems mailbox.
 * the message created by set_tb is sent to this mailbox.
 */
        send(ufs_mbx,ufs_token);
/*
 * the freewait kernel service waits for release of the data structure described
 * by ufs_token.
 */
        freewait(&ufs_flag);
/*
 * the filesystem task returns the return value of the request at the address
 * of u on the stack.
 */
        return((int)u);
}
```

SUPERVISOR FILESYSTEM INTERFACE

Asynchronous UNIX Calls

The PXROS event mechanism together with RPC and SIC is used to deal with asynchronous events like the UNIX signals. In order to handle those signals the supervisor allocates an RPC handler on demand of the user program (ALARM SVC), which is executed when a certain not masked event arrives. Because any handler may have parameters the UNIX signals can be treated selectively. The termination of a UNIX program due to a killing signal is easily achieved by executing the program as a procedure under breakpoint conditions, i.e. the BREAKPT kernel service is used. Figure 4 shows the piece of code called in the case of signalling.

FIGURE 4

```
/*
 * the routine sighnd is activated by traps or a call of the tty task
 * the only parameter is the signal number as defined in UNIX V
 */
sighnd(num)
{
        if(super_flag) {                    /* if task in supervisor mode */
                u->u_remflag = num;         /* just store the signal number */
                return;                     /* and return
        }
        u->u_remflag = 0;                   /* reset stored value */
        if(u->u_signal[num] == SIG_DFL)     /* if default action */
                restore(num);               /* restore the default handler
        calluser(u->u_signal,num);          /* call the user handler */
}
```

SUPERVISOR SIGNAL INTERFACE

Filesystem Task

This task is just a normal PXROS task providing filesystem services like creation, deletion, opening, reading, writing, closing etc. of files. As the figure 2 shows it offers its services not only to supervisors but any other task in the system, i.e. it is a server task. An incoming request is classified and eventually sent directly to a driver task, which carries out the operation and sends it back to the filesystem task, which in turn sends it back to the requesting task. File allocation information is held within this task and swapped in or out by demand, the data itself is only cashed in the drivers. Thus device specific operation is completely done within the drivers. This method allows the operation of the filesystem task on different devices without any knowledge of the physical structure of the devices. The sync mechanism of UNIX is achieved by driver operation thus even CMOS-RAM based filesystems without the requirement of a sync can be built without changing the filesystem task. Because of the fast intertask communication (no copy) virtually no overhead arises by the separation of the filesystem and disk task.

Disk and Tape Task

These tasks are responsible for all operations on their respective device, i.e. reading, writing and buffering of datablocks. These tasks provide consistent "errorfree" access to their device data. Because of the message oriented ITC these tasks can also reside in strong or losely coupled subsystems.

TTY Task

While the other driver tasks do not exhibit specific UNIX features this driver incorporates the tty relevant features, i.e. it does all line editing functions required by UNIX as well as the respective signalling. The latter is done by activation of the RPC/SIC entry of the supervisor which in turn completes the signal action.

Efficency of PXROS/SV

We compared the behaviour of PXROS/SV with Nationals ICM3216 running under standard UNIX 5.2 - which is a very fast implementation of UNIX 5.2 - by running a number of tools under both systems. The disk used in the ICM3216 has similar characteristics as the one we use. While the ICM has 4 MByte our system has 2 MByte of no wait state RAM. Both systems have the same CPU NS32016 running at 10 MHz. We found that the ICM having a separate I/O - system was just about 20 % faster than our PXROS/SV in its current version. Because the current PXROS/SV was generated using the GENIX 4.1 C compiler, which produces a 30 % slower code than the SYSTEM V compiler does, we expect that the currently prepared version of PXROS/SV generated by the SYSTEM V compiler, will show the same performance as standard SYSTEM V.

In contrast to the standard UNIX V the PXROS/SV operating system does not need an MMU, if no more than one UNIX program has to run in parallel. In this case the FORK and EXEC system calls are still provided, but the programs do not run quasi-

simultanously. Even without MMU the number of PXROS tasks is not restricted, i.e. the full realtime operating system functionality is available.

Integrating the Apple Macintosh in a UNIX Environment

Nick Nei

University of Glasgow
Computing Science Department
17 Lilybank Gardens
Glasgow
Scotland

(inei@cs.glasgow.ac.uk)

ABSTRACT

Work is underway in the Computing Science Department of the University of Glasgow to integrate clusters of Apple Macintoshes with clusters of UNIX machines. The Apple Macintoshes are connected to the AppleTalk local area network, and the UNIX machines (mostly flavours of 4.2 BSD) are connected to the Ethernet local area network. The challenge arises in integrating the alien PC environment of the Macintosh with a UNIX environment.

Many configurations are possible. A Macintosh can be used as a host on the Ethernet and thus behave (when connected) as an intelligent terminal emulator via its serial port. Files can be transferred between any hosts using ASCII file transfer or reliable methods like Kermit. More interestingly, a gateway between an AppleTalk network and an Ethernet network will allow any Macintosh to communicate using TCP/IP/UDP protocols with any UNIX machine on the Ethernet. A Macintosh can then connect to any UNIX host on the Ethernet using telnet, and transfer files using tftp and ftp. In addition, a UNIX host can be used as a file server to a cluster of Macintoshes, thus providing each with an enormous floppy diskette.

In the Department, an experiment is being carried out to use clusters of Macintoshes integrated into a UNIX environment for student teaching. This paper will report on some of the experiences and problems, not the least being infrastructural and logistical ones.

April 10, 1987

UNIX is a trademark of AT&T Laboratories. Apple, AppleTalk, MacWrite, MacDraw, LaserWriter and MacTerminal are trademarks of Apple Computer, Inc. Macintosh is a trademark licensed to Apple Computer, Inc. IBM is a registered trademark of International Business Machines. FastPath is a trademark of Kinetics, Inc. VersaTerm is a trademark applied for by Lonnie R. Abelbeck. DEC, VT100 and VAX are trademarks of Digital Equipment Corporation. Ethernet is a trademark of Xerox Corporation. Tektronix is a trademark of Tektronix, Inc. D200 is a trademark of Data General Corporation.

Integrating the Apple Macintosh in a UNIX Environment

Nick Nei

University of Glasgow
Computing Science Department
17 Lilybank Gardens
Glasgow
Scotland

(inei@cs.glasgow.ac.uk)

When Apple announced the Macintosh and called it *"a horse of a different colour"*, it attracted a following that included UNIX developers and users who worked on harnessing the features of the Macintosh with UNIX. The converse could also be true: the Macintosh exploiting the power of UNIX. Whichever is true, there are clear signs that a symphysis is taking place.

This paper surveys the growing integration of the Apple Macintosh in a UNIX environment. Starting from simple communications programs such as terminal emulators, a panoramic view of file transfer applications, multi-window management systems and file servers is given.

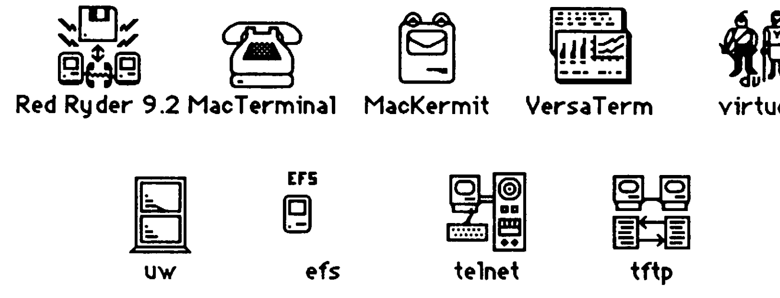


Figure 1

An abundance of terminal emulation programs are available for the Macintosh. Since many of them provide more than mere emulation of common terminals like the DEC VT100, Tektronix 4010 series and ANSI terminals, the looser term "asynchronous communications program" is better. They all communicate with a host computer via the Macintosh's modem serial port or printer port through a local area network, or even connected back-to-back with another machine. The simplest of these emulate one or two terminal types while the most sophisticated ones feature window management with different terminal emulations for any window. Most communications programs provide some sort of file transfer facility. They range from simple and unreliable ASCII file transfer methods to reliable methods like Kermit. Some will even upload and download properly encoded applications programs to and from the hosts.

The more sophisticated communications programs exhibit varying degrees of integration with the host systems. By integration we mean forms of synergy whereby two systems function co-operatively to provide a service which would not otherwise be available in the absence of one of the systems. UW (acronym for UNIX Windows) is a good example. It provides up to seven independant multiple windows using a window manager and server on the UNIX host. Windows can receive and send input and output independantly of each other. Windows overlap and can be stowed away, resized and moved around the screen: things an ordinary terminal cannot do.

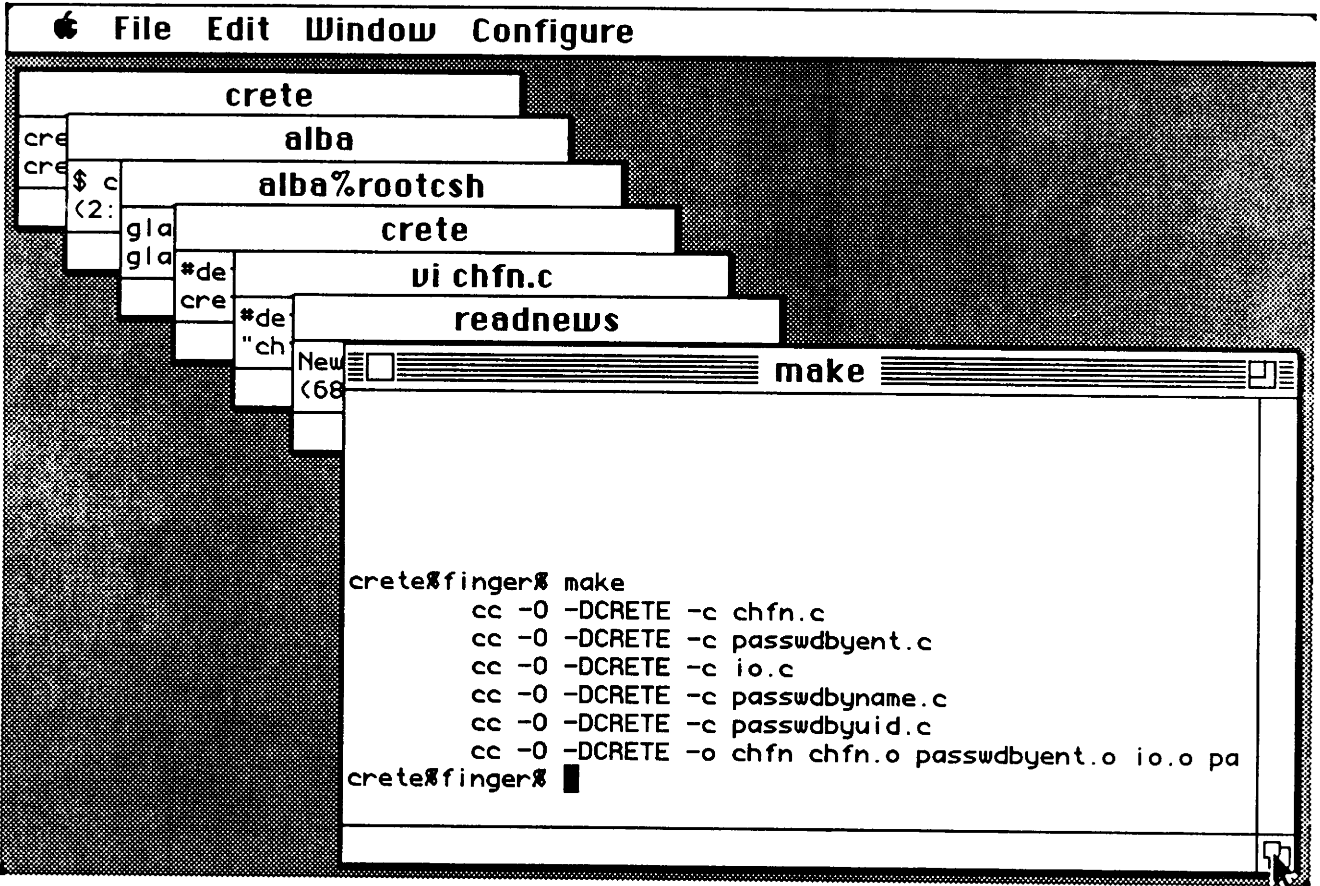


Figure 2: UW - Macintosh and UNIX Synergy

Many communications programs provide text editing and re-input. MacTerminal for example will allow terminal input and output to be edited in Macintosh mouse click-drag-and-select fashion and re-input to the host. Thus corrections can be made to long or commonly used commands before sending them to the host, saving much prestigidation. Another communications program that has further honed this method is Dumb Virtue which provides multi-window text-editing and single-window terminal emulation. It associates one window for input and another for output. A third window emulates an intelligent terminal. User input will appear in one window but will not be sent to the host until the carriage-return is typed. The output from the host then appears in the other window. Dumb Virtue also supports the Macintosh's cut-and-paste metaphor. Text can be selected with the mouse from one window and then saved on the Scrapbook (a temporary file) and later pasted to another window.

This idea of massaging captured data (from the keyboard and the host) can be taken further. With Red Ryder, for instance, text on the screen can be selected using the Macintosh mouse and then sent to the printer. Whole screens can also be dumped or archived on a printer or a text file. Alternatively, incoming data can be stored and/or routed to a printer and/or a disk file. Similar data manipulation techniques also often feature in graphics terminal emulation programs. For example VersaTerm emulates the Tektronix 4014 graphics terminal but will also create Macintosh MacDraw files.

The ability to transfer files reliably is an important function of any communications program. Most will support ASCII file transfer which is unreliable, but which may be the only recourse when other reliable methods are impossible. At the very least, some sort of flow-control mechanism for tandem ASCII file transfer ought to be available. The common unofficial standard method of reliable file transfer is called Kermit. The array of communications programs which support Kermit file transfer include MacKermit, Red Ryder and VersaTerm. They assume that the host machine also provide a complementary Kermit facility. In this way, both ASCII and binary files can be transferred reliably.

In addition to mere transfers, executable files and data files can be downloaded (or uploaded) from the Macintosh. A degree of integration results, which is borne of necessity because many Macintosh programs and data files arrive at UNIX sites via electronic mail or network news in encoded and compacted form. The most elegant uploading method uses MacTerminal. A utility called **macbin** decodes and uncompresses files in preparation for uploading, and produces three files with similar names (but with distinct suffixes), corresponding to the resource and data forks of a Macintosh file, and an information file. These are then uploaded using a utility called **macput**. Reversing the process (using **macget**), will download a file from the Macintosh. However, this method works only when flow-control is inhibited and the Macintosh must be connected back-to-back to the UNIX host. In many situations, such as in a local area network this arrangement is not possible. When a Macintosh is connected to an Ethernet communications server, 8-bit XON/XOFF cannot be supported and therefore MacTerminal cannot be used. Two alternatives are available. The encoded (text) file can be first transferred using Kermit and then decoded (using **BinHex**) on the Macintosh, or the file can be decoded and uncompressed on the UNIX host using **xbin** and **macbin** and then transferred in 8-bit fashion using Kermit to the Macintosh, arriving as an executable or data file.

Kermit file transfers are slow but reliable. However, the efficiency is very dependant on the implementation. In a benchmark of three communications programs, MacKermit was the fastest in transferring a file of 10240 bytes to a VAX-11/780. The following table summarises:

Program	Bytes/Second
Red Ryder	146
VersaTerm	151
MacKermit	329

Table 1

Kermit File Transfer Benchmark of 3 Communications Programs

Other file transfer methods are usually available: notably the XModem or Ward Christiansen method, tftp and ftp.

Most communications programs allow the Macintosh to automatically integrate itself into the host system. This is almost essential since serial communications protocols differ widely from site to site, and even within the same site. A user should at least be able to configure the baud rate, parity and the number of stop bits, and save the configuration for subsequent use. But some programs will also allow macros and even small procedures to be tailored by the user. Red Ryder is most notable for this. Macros and pull-down menus can be set up and invoked with the mouse. In this way, frequently used commands or key-strokes can be programmed and fully integrated into the UNIX host system.

Table 2 concludes this survey with a summary of terminal emulators and file transfer programs.

Program	Source Status	Terminal Emulation	File Transfer
Red Ryder	shareware	VT100, VT52	XModem, Kermit, ASCII
VersaTerm	Third Party	Tektronix 4014, Data General D200	XModem, Kermit, ASCII
MacKermit	public domain	VT102	Kermit
MacTerminal	Apple	VT100, IBM 3278	XModem
Dumb Virtue	shareware	ANSI, VT100, ASCII	ASCII (tandem)
UW	shareware	ADM 31, VT52, ANSI, Tektronix 4014	none
MacIP	public domain	VT100	Telnet, TFTP, FTP, EFS

Table 2: Summary of Communications Programs

UW does not provide file transfers although the author intends to at some later date. It will communicate with any system, but its multi-window facility will only operate under UNIX 4.2 BSD and later systems. **MacIP** is a collection of 3 programs: telnet, tftp (also ftp) and EFS. Details on MacIP are given below.

Several of the programs are either shareware or public domain products, and are more often subject to new releases and upgrades than proprietary ones, presumably because it will encourage users to honour the authors pecuniarily.

Whereas these communications programs discussed so far integrate by only providing terminal emulation and file transfer services, another breed of communications programs permit the facilities of both the Macintosh and UNIX to be exploited. These are: EFS (External File Service), MacIP and MacNIX which depend on an Ethernet to AppleTalk[1] gateway. One such Internet AppleTalk bridge is the Kinetics FastPath. It is a programmable device which permits AppleTalk networks to be bridged together, or allows a Macintosh to become a member of an ARPA-defined internet. The latter is accomplished by TCP/IP software to do IP routing and protocol encapsulation:

- IP packets, originating from the Macintosh, are encapsulated in AppleTalk Data Delivery Protocol (DDP) datagrams. At the FastPath, the IP is decapsulated from DDP and routed using IP routing. IP packets originating at an IP host and destined for a Macintosh are first encapsulated in DDP, then routed on AppleTalk.

[1] AppleTalk is the Macintosh local area network.

- DDP datagrams originating from an AppleTalk host may be encapsulated in UDP/IP datagrams and routed to a peer process on an IP host which will decapsulate the DDP from within the UDP. DDP datagrams originating from an IP host will be encapsulated in UDP first, then decapsulated at the FastPath.

Figure 3 illustrates a configuration which uses a FastPath to bridge a cluster of Macintoshes on an Ethernet backbone.

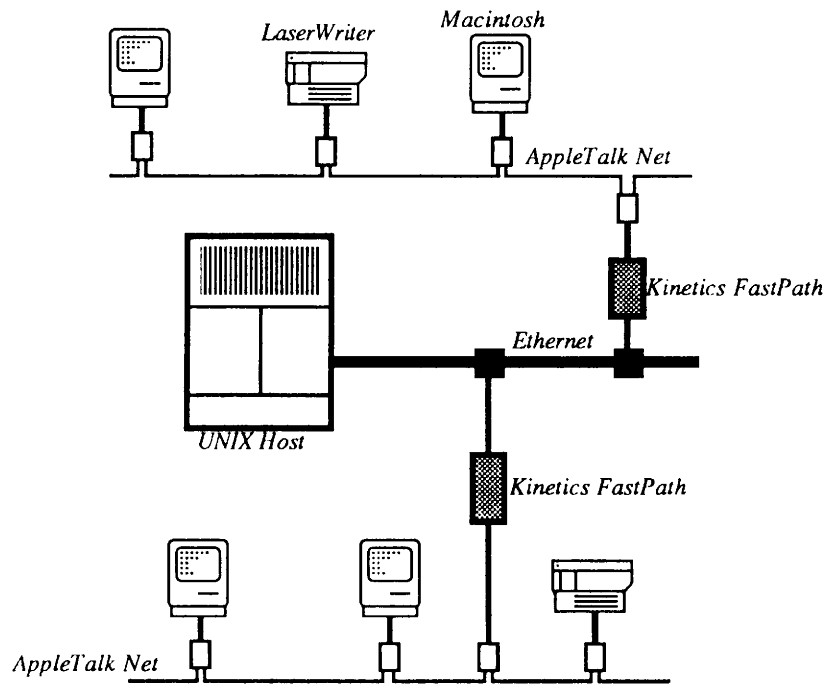


Figure 3: The Kinetics FastPath Bridge

A great deal of TCP/IP software for the FastPath has emerged in recent years from the research community. MacIP originated from Carnegie-Mellon University, EFS originated from LucasFilm and was adapted at Stanford University and CAP (which implements a subset of AppleTalk protocols for UNIX) was developed at Columbia University.

MacIP comprises a set of libraries and programs for telnet and tftp with other TCP/IP hosts. The FastPath is programmed to perform packet routing by either subnetting or by direct IP to DDP net mapping. Sites not using Class C Ethernets use the direct net mapping method. A table is used to describe a specific routing between an AppleTalk net and node number (which corresponds to a particular Macintosh) to/from an Ethernet Internet address. The UNIX host must then be told how to get to the AppleTalk network. Under UNIX 4.2, this is done by **route**(8C).

With the subnet routing approach, the FastPath is instructed to route packets if it detects that the packet has originated on a subnet on one side of the gateway and is destined for a different subnet on the other side. An internet address is divided into a network number and a host number, and in this scheme, the network number is the same for all internet addresses on the network. The host number is divided into the subnet number and a node number, and within each subnet, the subnet part of the host number must be the same. Each AppleTalk

net is a different subnet and the Ethernet is another, different subnet.

The **tftp** program uses the DARPA trivial file transfer protocol running on top of UDP to allow reliable file transfer between the Macintosh and an Internet site. Both server and user modes are supported. There are also four modes to control a file transfer. Three modes: ASCII, IMAGE and OCTET are used for transfers with non-Macintosh computers, and a fourth, MACINTOSH, is used for exchange with another Macintosh. This diversity is needed because of a silly design decision in the Macintosh to store a file in two parts or "forks": the data fork and the resource fork. In ASCII and IMAGE modes, only the data fork is sent, in OCTET mode, only the resource fork, and in MACINTOSH mode, both forks are sent. The OCTET mode is mostly used when transferring compressed programs over long-haul networks such as ARPANET and USENET.

The **telnet** program provides login connections with hosts on the Internet. It emulates a DEC VT100 terminal and it is also possible to use tftp from inside **telnet**.

The External File System (EFS) allows a UNIX host to act as a file server for the Macintosh. A UNIX directory containing Macintosh files can be mounted as a volume on a Macintosh and used like any Macintosh file. The effect is to make the UNIX host pretend to be an enormous floppy disc to the Macintosh.

EFS installs itself in the Macintosh's operating system the first time it is executed. This is best done when the Macintosh is started up. Subsequently, executing EFS will prompt for a connection to the server. A name server (Name Binding Protocol) daemon on the host resolves the hostname and replies with an Internet address to the Macintosh[2]. Connection with the EFS server on the target UNIX host can then be established, and the directory of Macintosh files mounted. EFS loads its driver portion into the system heap (or in high memory) and then detaches. It then sets a timer via the Macintosh vertical retrace manager to call the EFS driver and produce a "disk inserted" event. Once it has been notified of the new drive the remote volume will be mounted and the EFS driver intercepts all subsequent requests to the drive. The Macintosh DeskTop should then appear like Figure 4. The mounted volume is named after the directory's full pathname. In this example, the two hosts (muck and arran)[3] are Whitechapel MG1 workstations networked on NFS (Network File System).

Volumes can be unmounted by placing the volume in the Macintosh trash (or wastebasket) icon. This will cause the server process on the host to terminate. The shutdown command on the Macintosh will also unmount EFS volumes properly.

File transfer between the UNIX server and the Macintosh is intuitively transparent because EFS supports the Macintosh file/folder/DeskTop metaphor. Click and dragging the source Macintosh file/folder to a destination EFS volume effects the transfer. On the server, a single Macintosh file is stored as three files each with the same base name and the following suffixes: ".IF", ".DF" and ".RF" which correspond to the Information file, the Data fork and the Resource fork, respectively. Usually, either the Data fork or the Resource fork will be empty. Downloading files from the UNIX host to the Macintosh is very simple. Using **xbin**, an encoded application (say from USENET), is decoded to produce the resource fork in a file suffixed ".rsrc". Renaming the file with an ".RF" suffix and then copying the file to the local disk on the Macintosh effects the download.

At the Computing Science Department, University of Glasgow, such an integration of Macintosh and UNIX systems using EFS is used in an experiment to teach 40 undergraduate students programming in Pascal. A laboratory of twelve Macintoshes are used with two Whitechapel MG1 graphics workstations which act as EFS servers. The MG1s operates under a UNIX 4.2 BSD clone and are networked with 24 other MG1s using NFS. The Macintoshes are chained together on a single AppleTalk network with the Kinetics FastPath providing the

[2] The CAP version of EFS uses the AppleTalk Information Server, **atis**. Its function is to act as a name registry for the particular host **atis** is running on.

[3] Muck and Arran are the names of two Scottish islands.

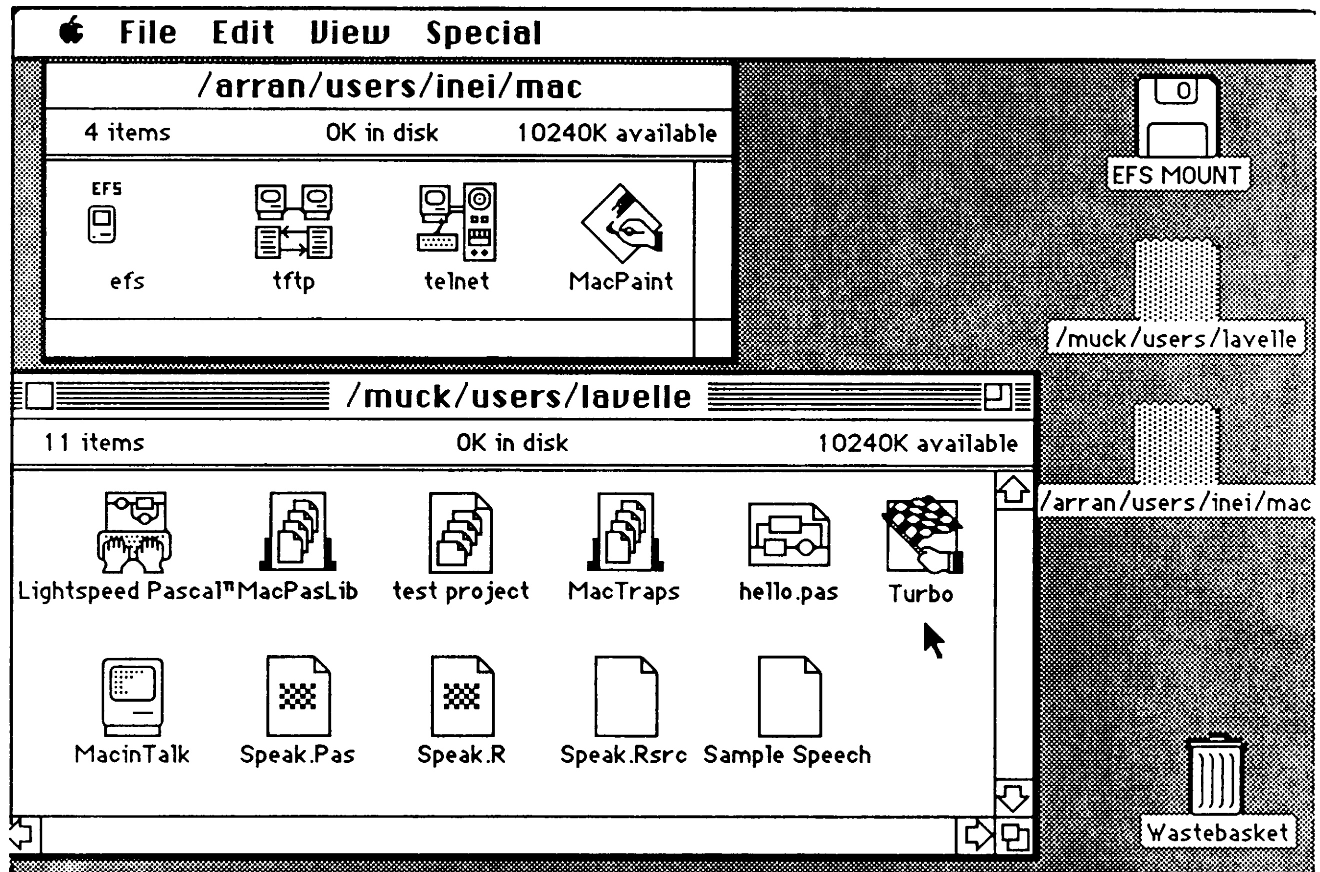


Figure 4:

The Macintosh DeskTop with Several Mounted EFS.

gateway between AppleTalk and Ethernet. Some expected advantages were:

[1] Administrative ease. Under UNIX, the tutor is able to conveniently control each student's files. For example, files and libraries can be easily distributed (and removed) using UNIX tools. Housekeeping and backups are also easily managed, using tools already available with UNIX.

[2] Logistic simplicity. Since EFS provided the peripheral storage, external drives for the Macintosh are not necessary. Without EFS, the logistics of carefully labelling and storing numerous floppy discs can be a nightmare.

[3] It is easy and safer to loan copyright applications to students if they are stored on the EFS servers and not on floppy discs. Since there are no external drives (and the EFS mount floppy disc must be returned at the end of a session), it is difficult to make illegal copies of copyrighted material.

The integration of the Macintosh and the UNIX system using EFS provided an excellent teaching environment. The students appreciated the Macintosh interface because it was intuitively simple. This enabled them to quickly settle into the computing environment and devote themselves to the laboratory work. This contrasts sharply to the previous laboratory environment of a single computer with terminal connections. The complexities of the operating system, the editor and commands were excessive and distracted the student from their intended laboratory work. From the tutor's point of view, the use of individual microcomputers is more satisfactory. In the old environment, when the central machine malfunctions, the entire laboratory is held up. With EFS, should the UNIX server fail, the laboratory can still continue with the use of floppy discs.

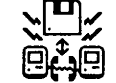
In addition to providing EFS, an Internet host can be used as a spooler. There is no printer spooler in the Macintosh system software. This is inconvenient and wasteful because the Macintosh is mostly idle and unavailable while the microprocessor in the LaserWriter does the hard work during printing. This slavish behaviour of the Macintosh can be remedied by using UNIX's own spooling facility.

A commercial product similar to EFS called MacNIX has recently become available. It provides similar file server and spooling functions. With MacNIX host printers may be accessed from a Macintosh to print UNIX files. It also works with TranScript to permit printing of PostScript, troff and nroff files on a LaserWriter connected directly to the UNIX host[4].

With the release of the Macintosh II, the integration is about to take on a new dimension. Apple has announced Apple UNIX for the Macintosh II. Details are not yet available, but many UNIX and Macintosh devotees are looking forward to an integration that brings out the best in the Macintosh and in UNIX.

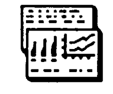
These are clear signs of growing integration of the Apple Macintosh in a UNIX environment. From crude terminal emulators and multi-window managers to file servers, this integration between two apparently dissimilar environments has come a long way. Whether the symphysis is a credit to the Macintosh or to UNIX is an interesting topic for debate. Nevertheless, the successful marriage at least attests to the good design of UNIX.

[4] There is also a utility called **mw2troff** which translates MacWrite documents to troff input. In this way, the data fork of a MacWrite document stored with EFS can be easily formatted and printed under UNIX.


Red Ryder 9.2

MacTerminal

MacKermit

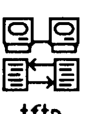

VersaTerm

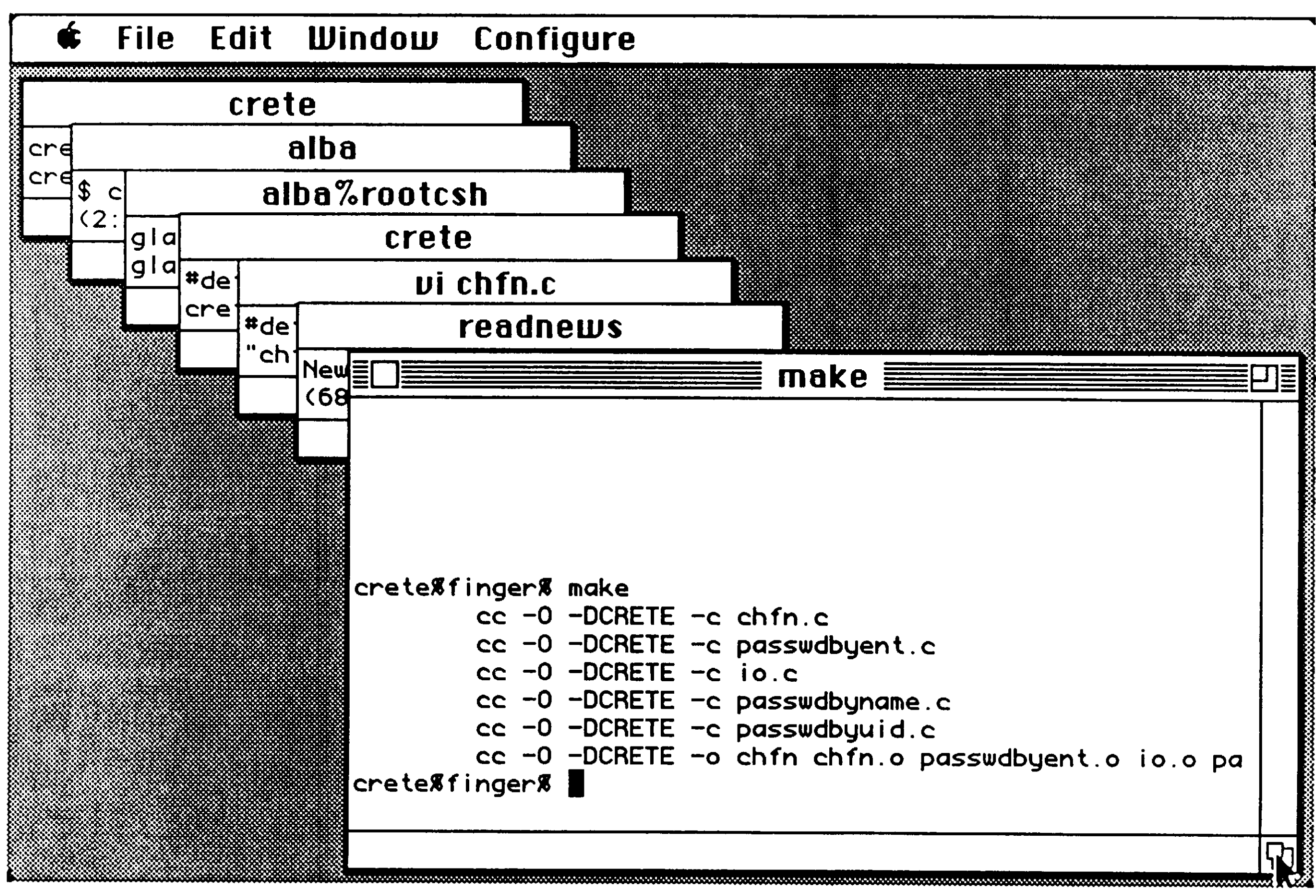
virtue

uw

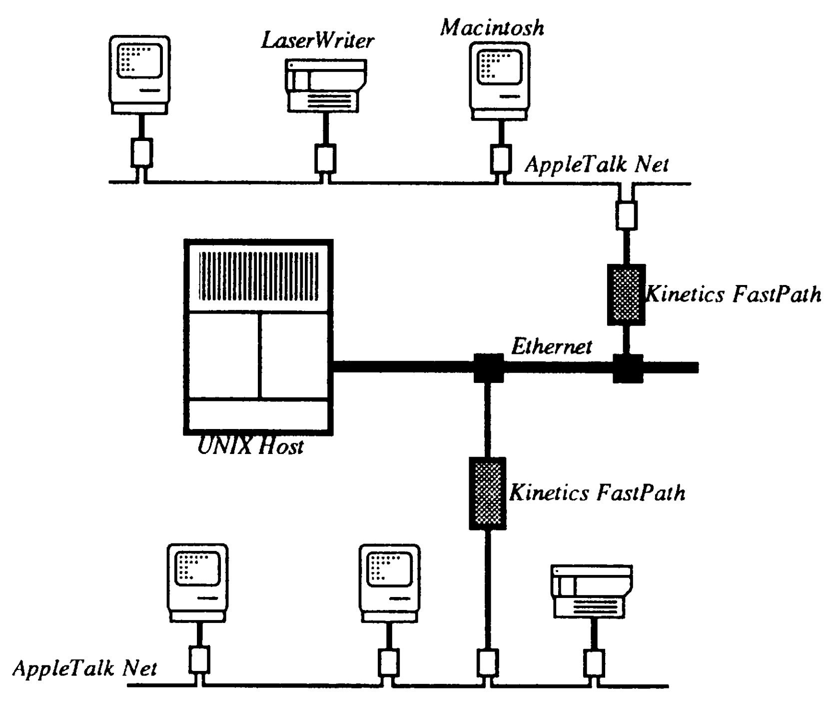
efs

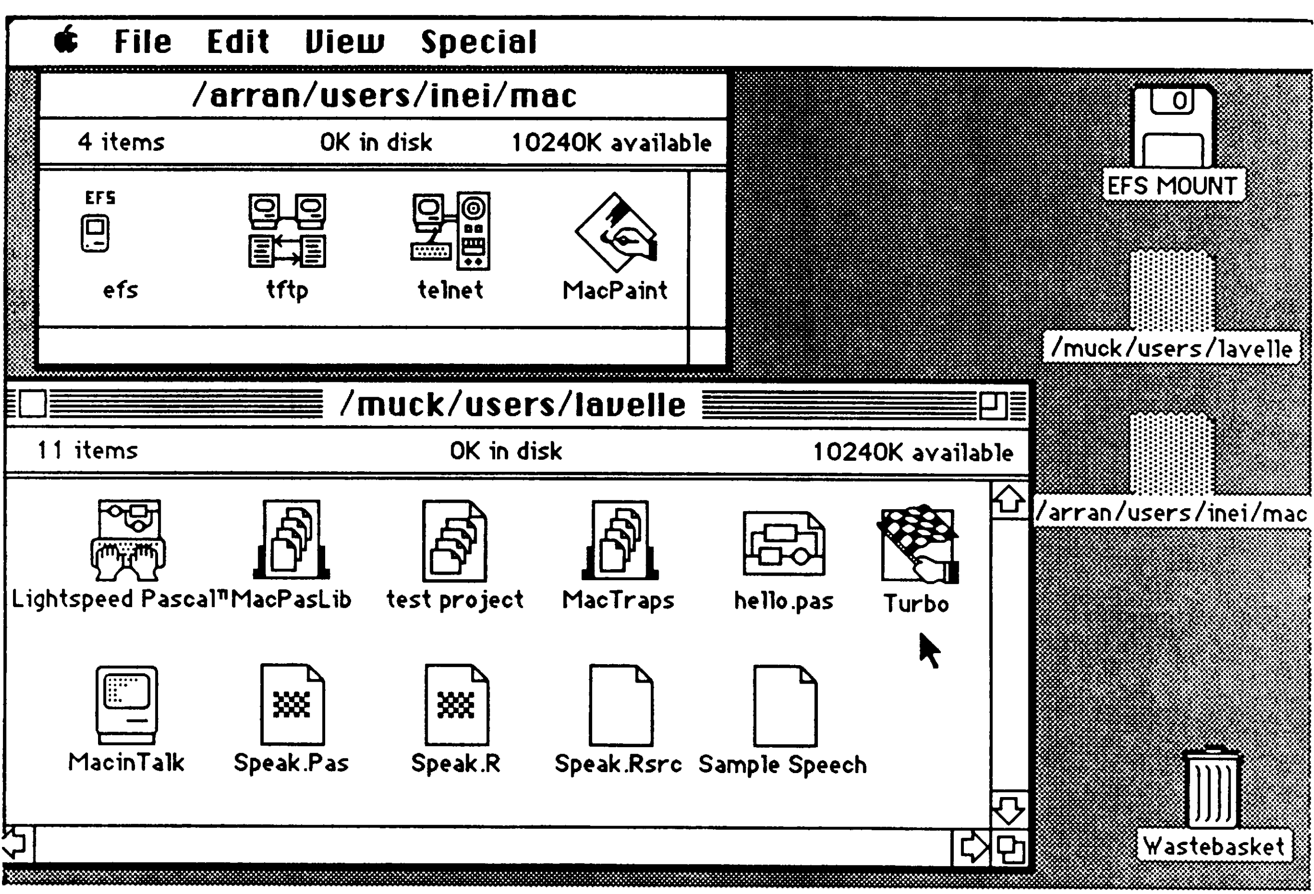
telnet


tftp



File Edit Window Configure
crete
alba
alba%rootcsh
crete
vi chfn.c
readnews
make
crete%finger% make
cc -O -DCRETE -c chfn.c
cc -O -DCRETE -c passwdbyent.c
cc -O -DCRETE -c io.c
cc -O -DCRETE -c passwdbyname.c
cc -O -DCRETE -c passwdbyuid.c
cc -O -DCRETE -o chfn chfn.o passwdbyent.o io.o pa
crete%finger%


LaserWriter
Macintosh
AppleTalk Net
Kinetics FastPath
Ethernet
UNIX Host
Kinetics FastPath
AppleTalk Net


File
Edit
View
Special
/arran/users/inei/mac
4 items
0K in disk
10240K available
EFS
efs
tftp
telnet
MacPaint
EFS MOUNT
/muck/users/lavelle
/muck/users/lavelle
11 items
0K in disk
10240K available
Lightspeed Pascal™
MacPasLib
test project
MacTraps
hello.pas
Turbo
MacinTalk
Speak.Pas
Speak.R
Speak.Rsrc
Sample Speech
/arran/users/inei/mac
Wastebasket

NAMES & ADDRESSES OF SPEAKERS

Mike Forsyth,
MEMEX Research Unit,
9-11 Maritime Street,
EDINBURGH,
UK.

Rob Pike,
Bell Laboratories,
Murray Hill,
New Jersey 07974, research!rob
USA.

David J. Brown,
Cambridge Uni. Comp. Lab.,
Corn Exchange Street,
Cambridge
CB2 3QG.
UK.

Jonathan P. Bowen,
Oxford Uni. Comp. Lab.,
Programming Research Group,
8-11 Keble Road,
Oxford
OX1 3QD.
UK.

Dominic Dunlop,
Sphinx Ltd.,
43-53 Moorbridge Road,
Maidenhead,
Berks SL6 8PL.
UK.

Pascal Beyls,
BULL,
1 Rue de Provence,
38432 Echirolles,
FRANCE.

Bertram Halt,
SIEMENS,
Charles de Gualle Strasse 2,
D8000 Munchen 83,
GERMANY.

Dr. Gunther Kruse,
Root Business Systems Ltd.,
Sauderson House,
Hayne Street,
London EC1A 9HH.
UK.

Peter S. Langston,
Bell Communications Research,
Morristown,
New Jersey 07960,
USA.

Dale Shipley,
Tolerant Systems Inc.,
81 East Daggett Drive,
San Jose,
California 95134,
USA.

Douglas V. Larson,
Hewlett-Packard Company,
19447 Pruneridge Avenue,
Cupertino,
CA 95014
USA.

Andrew S. Tantenbaum,
Dept. of Mathematics and
Computer Science,
Vrije Universiteit,
Amsterdam,
THE NETHERLANDS.

Brian E. Redman,
Bell Communications Research,
Morristown,
New Jersey 07960,
USA.

Rupert Grafendorfer,
Elektronikbau,
Dept. for Systems Software,
Gewerbehof Halle C,
A-4040 LINZ,
AUSTRIA.

Osmo Hamalainen,
Posti- ja Telelaitos,
Radio-osasto (Finnish PTT),
Helsinki,
FINLAND.

Markus Rosenstrom,
Oy Penetron Ab,
Espoo,
FINLAND.

Alan Chantler,
Dept. of Computer Science,
Coventry Lanchester Polytechnic,
Priory Street,
Coventry,
CV1 5FB.
UK

Philip H. Dorn,
Dorn Computer Consultants Inc.,
25 East 86th Street,
New York,
NY 10028.
USA.

Dr. Ernst Janich,
Universitat Ulm,
Sektion Informatik,
Postfach 4066,
D-7900 Ulm,
WEST GERMANY.

Doug Michels,
400 Encinal Street,
PO Box 1900,
Santa Cruz,
CA 95061,
USA.

Bjarne Stroustrup,
Bell Laboratories,
Murray Hill,
New Jersey 07974,
USA.

Christian Tricot,
LGI-IMAG,
University of Grenoble,
FRANCE.

Martin D. Beer,
Dept. of Comp. Science,
University of Liverpool,
PO Box 147,
Liverpool
L69 3BX.
UK.

Dr. Rolf Strothmann,
HighTec EDV-Systeme GmbH,
St. Johanner Str. 38,
6600 Saarbrucken,
WEST GERMANY

Nick Nei,
University of Glasgow,
Comp. Science Dept.,
17 Lilybank Gardens,
Glasgow,
SCOTLAND.