AUUGN

Australian Unix systems User Group Newsletter

Volume 6 Number 5

Registered by Australia Post Publication No. NBG6524

The Australian UNIX* systems User Group Newsletter

Volume 6 Number 5

April 1986

CONTENTS

Editorial	2
Assessing Interactive Programs via Batch Processing	3
UNIX Fighting Fires	7
A Controlled, Productive Applications Environment at Esso Australia's Production Department	19
CNFX - File Transfer Between UNIX systems and CSIRONET	30
Folding Regular Polyhedra	40
Initial Experiences with a Gould UTX/32 System in a University Environment	46
An Introduction to the UNIFY Data Base Program	51
fsh - A Functional UNIX Command Interpreter	57
The Perth AUUG Meeting: A Visual Perspective	70
AUUG Meeting in Canberra	79
AUUG Membership and Subscription Forms	81

Copyright © 1985. AUUGN is the journal of the Australian UNIX systems User Group. Copying without fee is permitted provided that copies are not made or distributed for commercial advantage and credit to the source is given. Abstracting with credit is permitted. No other reproduction is permitted without the prior permission of the Australian UNIX systems User Group.

* UNIX is a trademark of AT&T Bell Laboratories.

Editorial

This issue is devoted entirely to the proceedings of the Perth conference. It contains eight of the sixteen papers presented; the others I have so far been unable to obtain.

The group is also looking for a new newsletter editor as I find that I am unable to devote enough time to the exercise. If you are interested, contact me as soon as possible at the address and phone number appearing on the last page of this issue.

Memberships and Subscriptions

Membership and Subscription forms may be found at the end of this issue and all correspondence should be addressed to

Greg Rose
Honorary Secretary, AUUG
PO Box 366
Kensington NSW 2033
Australia

Next AUUG Meeting

The next meeting will be in Canberra at the Australian National University. Further information appears at the end of this issue.

Contributions

Come on you people out there in UNIX-land, send me your views, ideas, gripes, likes or whatevers.

Opinions expressed by authors and reviewers are not necessarily those of the Australian UNIX systems User Group, its Newsletter or the editorial committee.

Assessing Interactive Programs via Batch Processing

John Lions
Department of Computer Science,
University of New South Wales,
Kensington, NSW 2033.

For the past several years, I have had to teach C programming to students about to begin a formal study of operating systems. This is because the subject is taught via a case study of the UNIX system which is written in C, as are nearly all the user programs that react directly with the UNIX system kernel via system calls.

Since 95% of my students do not know C to start with, a programming exercise (or two) is the time honoured way to introduce them to the language. Since they are going to be studying the internals of the operating system, especially the kernel program's interface to user programs, it is a good idea that they write at least one program that uses system calls.

System calls allow a program to both sense and change its environment. The most striking environmental change is the use of *fork* to create a new clone of the current process, and *exec* to change it into something new (yes, UNIX does remind me of Aladdin's cave sometimes). Somewhat more mundane is the act of creating or deleting files. Environment sensing includes determining whether a file by a certain name exists, and if so, finding its type, size, etc.

An important class of programs that sense environments are the interactive programs such as *ed*, *vi*, *mail*, etc. (Actually the majority of UNIX commands such as *sh* and *pr* are not interactive at all.) Interactive programs are not trivial, but they are more fun to develop—so that is why they are chosen.

scat

Initially the exercise was to write a program dubbed *scat*, short for *super cat*. This command was to copy its input file(s) to its standard output, except that if an input line began with '!', the line was to be treated as a command to be executed, and its output was to be merged with the output stream in place of the input line. The input files were either given as arguments to an invocation of *scat*, or else the standard input was to be used. To make sure that students actually got to use *fork* and *exec*, use of the *system* library routine was (and still is) forbidden.

This exercise was not without its challenges, but was not used again. Instead I have tended to use variations on the following two exercises:

update

In this exercise the problem is to allow a terminal user interactively to append an arbitrary string to the end of each line of a data file, e.g. a file of student marks, with one line (record) for each student. Each new appendage is preceded by a tab character. In operation, a line from the original data file is written to the user's terminal (standard output file) except that the newline character is replaced by a tab, so that the cursor remains on the same line. The user then types the new string followed by a Return character. To the user program, the new string appears as a line on the standard input which can be read and written to the revised data file that the program is creating. Thus the user program has to deal with four separate files.

One of the problems I find when running such programs is that inevitably, every so often, I get interrupted to do something else, or I need to get some more information, or I want to revise one of the earlier entries. Thus the shell escape mechanism is important here too, and a suitable interpretation of the '%' in a command string is very useful for referring to the revised data file.

inspect

This exercise involves writing a program that can be used interactively to browse through a set of files (like ones I haven't seen for a while

and have half forgotten). For each file on the command argument list, the program displays a status line giving the file name and some type information. The user can then interactively decide whether to view the file from sequentially, or to go on to the next file. File contents are displayed using *cat*, or *pcat*, or *ls -l*, or nothing, depending on whether file is just ordinary, or packed, or a directory, or a special file. If a short sample of the file contents is enough, the Delete key can be used to terminate the display. Once again a shell escape via a '!' command line, with interpretation for '%' is extremely useful.

This exercise is my favourite. I keep a copy of my own private *inspect* program on hand for house-keeping tasks all the time.

Students generally enjoy this assignment, but they have relatively little time to learn C, to read all the relevant manuals and to get down to work. Since it is a learning exercise, I am quite happy for students to work in pairs for mutual support, but actually relatively few do. A characteristic of modern-day students is that they are very reluctant to undertake any exercise, no matter how relevant and worthwhile it may seem, unless there is a tangible reward (marks towards their final grade) at the end of it. Moreover they expect the mark will be weighted to reflect the relative effort that they have to expend. Thus I have been regularly faced with the problem of assessing with reasonable accuracy the relative values ofsome 60 to 80 interactive programs written by some active, enterprising, but not fully aware students. Anything is possible, and it usually does. The programs may be interactive but the testing has to be done in batch mode.

Testing

Students submit a source version of their program via the network. This is checked for certain properties and then compiled (woe betide the owner if the compilation fails). The object program is then executed in a number of test situations. I don't like the word 'situation' as it is now commonly used, but here it reflects not only a set of input data, but also an environment (like a dungeon) of data files that the program may or may not notice or manipulate (rogue fans will be pleased to note).

The general plan is to devise a series of tests to check that a program does what it is supposed to do, and then to check that the program behaves reasonably when it is asked to do unreasonable things like read an unreadable file, or execute a non-existent file, or create a file in a directory where there is no write permission. Before the test begins, the initial environment has to be set correctly: for example it may be empty, or it may include three data files whose creation times have a prescribed relationship. After the test, the environment has to be checked for any non-regulation changes.

For each test, there is a 'correct' output and final disposition of data files in the environment, determined by running a 'correct' (i.e. my) version of the program. The outputs have to be compared: if they match exactly, well and good; if they don't, then the differences have to be identified, and assessed. In the absence of expert systems that can follow students' thought processes, this assessment can be a painfully painstaking procedure. When all the testing is done, a printout is generated listing all the results: for a test, one line will suffice if it was successful; maybe half a page or more will be needed if it was not. My long standing ambition is to keep the amount of paper printed (and subsequently scanned by me) to less than half a box of paper each time!

Murphy's Department

Over the years I have encountered problems which cast light on the UNIX system and its users. Here are some observations:

a. When the standard output file is directed to a terminal, there is no output buffering; when the output is directed to a file, there is output buffering. This usually works fine unless the output file is a merger of the outputs of two or more processes, or the standard output file and the standard

error file. There are ways of dealing with this problem, but the only universal solution is to suppress output buffering explicitly by including a *setbuf* command at the beginning of each program.

b. Actually the output of student test programs is never sent straight to a file because programs that enter infinite loops still exist. The output has to be piped to a program that will 'die' as soon as it has seen enough, and hence cause the termination of the test program. Getting a sufficiently incisive criterion for 'enough' is a problem. A simple output line limit is sufficient, but may allow too much output in many situations. A simple line comparator is an improvement: see the same line three times in a row, then die. However output loops involving cycles of two or three lines are a problem too.

c. An infinite loop does not necessarily produce output lines. Programs that execute too long must also be terminated. By rights, 'too long' should be measured in cpu time consumed. There is a weakness here in UNIX: on our system, the criterion has to be in terms of elapsed time, and to be on the safe side (given the wild transient fluctuations of load on our system), I allow one minute of elapsed time before termination. Even at this level I have noticed some peculiar behaviour at times which I can only put down to a scheduler glitch somewhere when heavy swapping is occurring. The solution I have adopted is to send an interrupt signal to the running test program every ten seconds until the minute is up, when a kill signal is sent. The interrupt signal is set to be ignored so it doesn't terminate the program, but it does stir up the scheduler.

d. These signals have to be sent by a separate process from the one receiving the piped output, so every execution of a test program involves three processes, not just one.

e. Setting up a suitable execution environment is a problem. A directory for testing is established. This is cleared before each test, and then filled with a selection of other files. Ordinary files are copied from master versions. A directory file may be created if needed. Access to a special file is obtained by linking */dev/tty* into the test directory.

f. In order to provide reasonable protection from a 'rogue' program, each program is initially 'frisked' for signs of *chdir* or *chmod* or *unlink* system calls which would never be needed in the normal way. As a further precaution, the test directory, which must fully accessible to the test program, has a parent directory for which write permission is turned off during the testing period. In order to be really defensive, the test program should be executed with a separate user number from that of the test supervisor (so far that precaution has not been needed).

g. When data files are copied into the test directory, if the creation times have to be distinct, then at least one second must elapse between successive creations. The script that does this calls *sleep 2* after each file is copied. This slows the testing process considerably, but I think it may be appreciated by other users sometimes.

h. Assessing the environment after testing presents problems. Usually the input data files are unchanged, but sometimes they are not. A technique I have used is simply to 'gather up' all the files into one big one to express the end result. Some care is needed, e.g. not to include a *core* file if the program dropped one. *Diff* can then be used to compare the test and target results, usually quite satisfactorily.

i. Comparison of the test and target output files presents the most problems. The worst problem occurs when the output is almost correct but there are methodical differences e.g. with punctuation or

spelling or with a few additional blank lines here and there. (Some students can see no differences between e.g. 'non-existent' and 'non-existant' or 'non existent' or 'Non-existent' or 'non-existing'.) The problems raised here point to a need for smarter file comparators than *diff*, with or without the *-b* option.

j. Testing for a program's reaction to Deletes i.e. interrupt signals is *not* something I have managed to do batch-wise. The requested behaviour is invariably quite simple: the parent program should ignore interrupt signals, but child processes should not (at least not by default). This testing is done interactively under the control of a shell script so that only two keys (Delete and Return) are used.

k. When the testing is all over and printing is the order of the day, a numbered listing for each program is printed ahead of its test results. I have found that printing a summary of the program obtained by *grep*ing the number program listing for the following terms: *access*, *bin*, *chdir*, *chmod*, *close*, *exec*, *fork*, *open*, *read*, *seek*, *signal*, *system*, *wait*, and *write*, reveals important details of the structure of the program, and also serves very well as an index into the whole program.

UNIX Fighting Fires

Glenn Huxtable

Department of Computer Science
University of Western Australia

1. Introduction

This paper describes a relatively simple, and efficient system for remote monitoring of fire alarms.

Traditionally the task of remote fire alarm monitoring requires a high degree of reliability. The classical solution has been to duplicate vital components of the system to guard against failure. This is often expensive and leads to complex systems of data collectors, and arbitrators to determine which system to use when a fault occurs.

The Western Australian Fire Brigade Operations Management System (BOMS) is a distributed network of control systems reporting to a central host; each control system is capable of autonomous control in the event of a failure. There is no duplication of components. BOMS is also an information system, containing metropolitan street and building data, and Brigade status information to rapidly assist in efficient mobilisation of Brigade resources.

At the core of the system is a PDP 11/73 running UNIX (Version 7). This may be somewhat of a surprise, given that UNIX has attracted much criticism of its capabilities in "real world" situations.

This paper will overview the design and working of the complete system, and describe the UNIX interface and associated software.

2. Historical Perspective

For nearly 10 years the Western Australian Fire Brigades Board (WAFBB) have scanned every 2.6 seconds, some 10000 alarm functions in nearly 1000 buildings in the metropolitan area. The monitoring of Direct Brigade Alarms (DBA) is represented in figure 1. This configuration is widely used for many telemetry monitoring systems including fire, security and process control tasks.

'End Units', or alarms to be monitored, are connected via leased lines to the nearest district Concentrator, located at a Fire Station. This maintains the shortest physical distance, and minimises Telecom line costs between the alarm and Concentrator. The Fire Station Concentrators are connected in turn to a Multiplexer located, together with its associated control computers, at the Operations Center of the WAFBB.

The task of monitoring fire alarms is highly sensitive to faults in the monitoring system, where failure of the smallest component can mean loss of property, or loss of life.

Elaborate techniques have been developed to provide the necessary level of reliability, usually involving the duplication of many of the vital components of the system. Duplic ating host computers, or using tandem CPU systems, is expensive and can introduce greater software complexities. Duplication can be taken to extremes, with complete systems being duplicated, down to having Telecom lines leaving each station in diametrically opposite directions to reduce the risk of both being cut. Complicated arbitrators are needed to automatically switch over to the 'spare' circuits when a failure occurs. The system can easily become so complex as to present a reliability problem!

2.1 DBA System

The early DBA system, shown in figure 2, was designed largely of TTL logic, with the only processing power being in the dual host computers. These were connected to the Multiplexer by a high speed bus, so that telemetry could be processed in 'real-time'. Software in the host computer, programmed in assembly language for speed, directly controlled the Multiplexer.

System integrity was maintained by an Arbitrator to detect the failure of the 'active' computer and switch automatically to the 'spare'. The provision of dual computers was the extent of duplication in this system. Failure of the Arbitrator or Multiplexer caused the central monitoring of alarms to fail.

While relatively advanced for its time, the TTL based design was inflexible and difficult to change to meet the growing needs of the Brigade.

3. BOMS System

The new BOMS system, shown in figure 3, has a similar topology to the early system, however micro processors are now used in both the

Multiplexers and the Concentrators. A single host computer controls the system, the Multiplexer and Concentrators are intelligent and there is no need for an Arbitrator.

By moving processing power into the Multiplexer and Concentrators, much of the 'real-time' processing may be removed from the host computer, which is now able to perform many other less demanding tasks.

Modular design of Multiplexer and Concentrators units enables easy maintenance and replacement of components. Each unit consists of a standard backplane, into which may be placed the following custom designed boards.

- A Controller Module:
  A board containing the processor, memory and EPROM'ed software which drive the Multiplexer or Concentrator unit. New functions may be programmed into the EPROM store.

- Power Module:
  A plugin power supply which provides power, through the backplane, to all other modules.

- Modem Modules:
  Each modem module connects to a Telecom leased line. The Multiplexer and Concentrators scan modem lines for telemetry information and report line failures.

- Line Modules:
  Line modules monitor and display the status of 'End Line' units (alarms).

- Speech Modules:
  Speech Modules allow the Multiplexer and Concentrator Modules to manually report active alarms. The speech modules use a high quality synthesised (American) voice with a tailored dictionary of about one hundred words to broadcast warnings over speakers in the Fire Stations, and via Telecom lines to the Centrol Operations Center. These modules also enable the user to manually interact with the Controller Module, to select an alarm number on a thumbwheel, display its status on a row of LED's, and acknowledge active alarms.

Alarm Modules and Speech Modules enable the Concentrators to be operated manually during Multiplexer communication failure. Speech modules are able to broadcast through Fire Station PA systems, and enable alarms to be

monitored from unmanned country Fire Stations, or by Duty Firemen with relatively little training.

The Multiplexer also connects to a log printer to record all alarm activity, and during Host failure will respond to manual operator commands via a local terminal in the Central Operations Room.

Duplication is unnecessary as the Multiplexer and Concentrators can be easily operated manually. Failed components can be quickly identified and the module replaced. Complete modules can be easily built from stock spares, and programmed in a few minutes. Future development of the system is a matter of reprogramming the Controller modules.

4. UNIX Software

The computers of the early DBA system were dedicated to monitoring telemetry information in the Multiplexer, and reporting alarm activity to operator terminals in the Operations Room. By placing much of the monitoring activity in the Multiplexer Module, the host computer can perform many other administrative functions.

UNIX was chosen as the operating system for the host computer because it offered the greatest freedom of choice of hardware, was widely availability from many sources and provided the level of flexibility needed. UNIX further had the advantage that the user interface could be designed to suit the needs of the operations staff, professional Firemen with little experience of computers.

The INFORMIX relational data base was selected for data management as it provides both a library of interface routines and a flexible 4th Generation Language (4GL) package. The 4GL interface allows Fire Brigade staff, with little or no programming experience, to generate and maintain programs for information retrieval, administration and report generation. The C language interface is used in the various control programs maintaining telemetry data and interfacing with telemetry systems.

The BOMS system consists of the following software modules.

4.1 BOMS Data Base

At the heart of the system is the BOMS data base, comprising of:

- DBA alarms -
  extensive building information for all metropolitan DBA alrams including address, nearest cross street, contact numbers, special

instructions and 'standard applicance turnout'. The latter is a list of fire vehicles which should attend the DBA alarm.

- Brigade Availability Tables (BAT) -
  type, current status and location of all Brigade vehicles. This information is referenced by the 'standard appliance turnout' above.

- Street Directory -
  an online Metropolitan Street Directory including street names, high and low street number range and Metropolitan Street Directory map coordinates.

4.2 Mux Communications

Mux is a program which maintains communications, via a standard tty interface, with the Multiplexer. Communication is governed by simple Master/Slave handshake, where either Multiplexer or *Mux* may initiate a message only when it is 'Master'. The 'Slave' may become 'Master' only when passed a POLL message from the Master,

<start-of-header><POLL>

thus enabling it to send any waiting messages. This occurs once a second, or when the 'Master' has no further messages.

Messages are packaged in a simple protocol, consisting of

<start-of-header><message-type><message-data><checksum>

and positive or negative acknowledgement replies

<start-of-header><ESC><ACK>

or

<start-of-header><ESC><NAK>

The checksum in a data message is a Cyclic Redundancy Checksum (CRC).

The Multiplexer sends telemetry information about DBA alarm activity, and alarm and Concentrator failure and recovery. Fire and Fault alarm calls are reported to operators in the Operations Room. Alarms can be tested remotely by placing the 'end line' unit in TEST mode, the TEST will be recorded but will not cause a turnout.

Operators may direct *Mux* to send messages to the Multiplexer to request the status of any alarm, to acknowledge DBA alarms (this also turns off alarm bells in the Operations Room), to bring alarms or Concentrators online or offline or to report a count of errors on a Concentrator.

Mux maintains a file of current alarm status for each DBA alarm.

Multiplexer failure is detected and reported by *Mux*, which will

automatically restart communication when the Multiplexer restores. During Multiplexer failure the BOMS system remains operational, however alarms must be handled manually at the Concentrators where DBA alarm calls will be announced over the PA system.

4.3 BAT Display

The BAT display is a large bank of lights in the Operations Center which reflects the current status and availability of all Brigade vehicles. The display logic is interfaced to the computer via a standard tty interface. Messages of the form

<vehicle-number><status-code>

cause the display logic to update the display. *Bat* is a simple program which periodically reads the BAT data relation (containing vehicle status) and updates the BAT Display board.

4.4 AUTOLINK Communications

The AUTOLINK is a device which interfaces to the Brigade's two-way radio system, to send a data burst through the radio to electrostatic printers mounted in vehicles. The printer has a number of 'status' buttons which send a status message back through the radio system to the AUTOLINK. The AUTOLINK is interfaced to the host system via a standard tty interface.

Autolink is a program which periodically polls the AUTOLINK for status updates from vehicles, to maintain the BAT data relation.

Operators can direct *Autolink* to send short (256 byte) messages to any vehicle fitted with a printer. The AUTOLINK device can send to 3 printers at a time.

Autolink protocol is simpler than that of *Mux*, since it is always master, it initiates data messages or polls for vehicle status updates. The AUTOLINK provides positive and negative acknowledgement to these messages.

4.5 OPERATOR Interface

Opr is the login shell and UNIX interface for operators in the Operations Center. It uses the curses window package to provide a secure, menu driven interface for the operators. Single keystrokes select menu functions enabling the user to:

- Display Concentrator status.
- Accept DBS FIRE, and FAULT alarm calls.
- Display DBA building/alarm information.
- Display and update vehicle availability data.
- Search for, and display TEST, FIRE and FAULT alarms.
- Bring alarms and Concentrators online and offline.
- Edit and send messages via the AUTOLINK.
- Display street information and maps.
- Call shell scripts and Informix 4GL routine to perform other administrative tasks.
- Send and receive mail and other 'UNIX' - like tasks.
- Produce a multitude of reports via the INFORMIX report writer (Informer).

Of these the most crucial is the processing of DBA calls (known as 'Mode 2'). When a fire call is received the Multiplexer rings a buzzer in the Operations Center. *Mux* receives the alarm message and places it in the alarm table. *Opr* fetches the alarm and sends *Mux* a message directing it to send an acknowledge message to the Multiplexer, which then turns off the buzzer in the Operations Center. This mode displays 2 windows of DBA/Building information, the 'standard turnout' for the alarm indicating which vehicles are available (from vehicle status relation), and which backup vehicles may be called on. With this information the operator is able to mobilise the required resources. Provision is made to transmit Building information to printers in the vehicles as they 'turnout'.

4.6 Street Search

The Street data base compirses information from the Metropolitan Street Directory, including names, street numbers, suburb, and gird reference for every section of street, intersection to intersection in the Perth Metropolitan Area. The grid reference relates to map files which can be used to display high resolution, colour maps of grid rectangles taken from the Metroploitan Street Directory.

Streets, invoked from *Opr*, performs a 'wildcard' lookup on a street name, with or without suburb or street number, and displays the best match. The opterator may expand the display to show other records with a lesser degree of conformity. If a record is selected, the 'standard turnout' for the area is displayed and the street map is transmitted to an IBM PC in the Operations Center, where it is displayed on a high resolution colour graphics display. Map files store drawing commands, so display time can take up to a minute, while storage overheads are kept low.

The street search provides mobilising information for '000' telephone

calls, and replaces an earlier system of microfiche reproductions of the Metropolitan Street Directory. While Brigade vehicles still carry fiche maps, it may one day be possible to transmit maps directly to vehicles via the AUTOLINK system.

Special maps and plans are also kept for a number of major industrial, commercial and service complexes, highlighting points of relevance to the Fire Brigade.

5. Kernel Modifications

The current BOMS implementation relies on two 'pseudo-drivers' (device drivers which are not attached to real devices).

The passing of active alarms from *Mux* to *Opr* must be fast and efficient. Alarms must be processed in a FIFO fashion, (difficult to implement in simple text files), and have to be efficiently cleared when the Multiplexer is reset or alarm telemetry is restarted (expensive in a database). The alarm device manages a linked list of active alarms which is stored in a table in kernel memory. Library calls put active alarms on the tail of the list, and remove them from the head, or reset the entire list. Synchronisation and mutual exclusion are ensured by the driver.

Opr communicates with *Mux* and *Autolink* to pass messages to the telemetry systems. The messages require acknowledgement which is difficult to synchronise through data files. The message 'pseudo-driver' allows programs to write fixed length messages into a table in kernel memory. Programs may nominate which program to send to and receive from.

An advantage of the 'driver' approach is that it can inform *Opr* if *Mux* is not active or has failed. *Opr* does not have to restart the message connection when *Mux* restarts. Simple UNIX pipes were insufficient to the task, as they must be ingerited from a parent process. BOMS programs are not 'related' in this manner.

The use of these drivers arose because Version 7 UNIX has insufficient shared memory and message passing facilities for the task. While the above could be achieved with simple files, the pseudo-driver approach was an expedient compromise. Both UNIX System V and 4.2Bsd UNIX implement message passing mechanisms which appear suited to this application.

6. Conclusion

BOMS is a fault tolerant system for monitoring real-time telemetry. The

modular hardware design minimizes component downtime, while the distribution of 'intelligence' throughout the system provides a reliable failsafe against failure and relieves the host computer of the real-time constraints. The monitoring task is reduced to one of simple communications and data management.

UNIX provides a secure environment, and enables the telemetry monitoring and communications tasks to be further separated into individual modules. With the correct interprocess communication mechanisms (IPC), UNIX can be used to build complex control systems from simple communicating programs.

The finished system allows a high degree of flexibility in changing the host computer, data base management system or any of the peripheral systems.

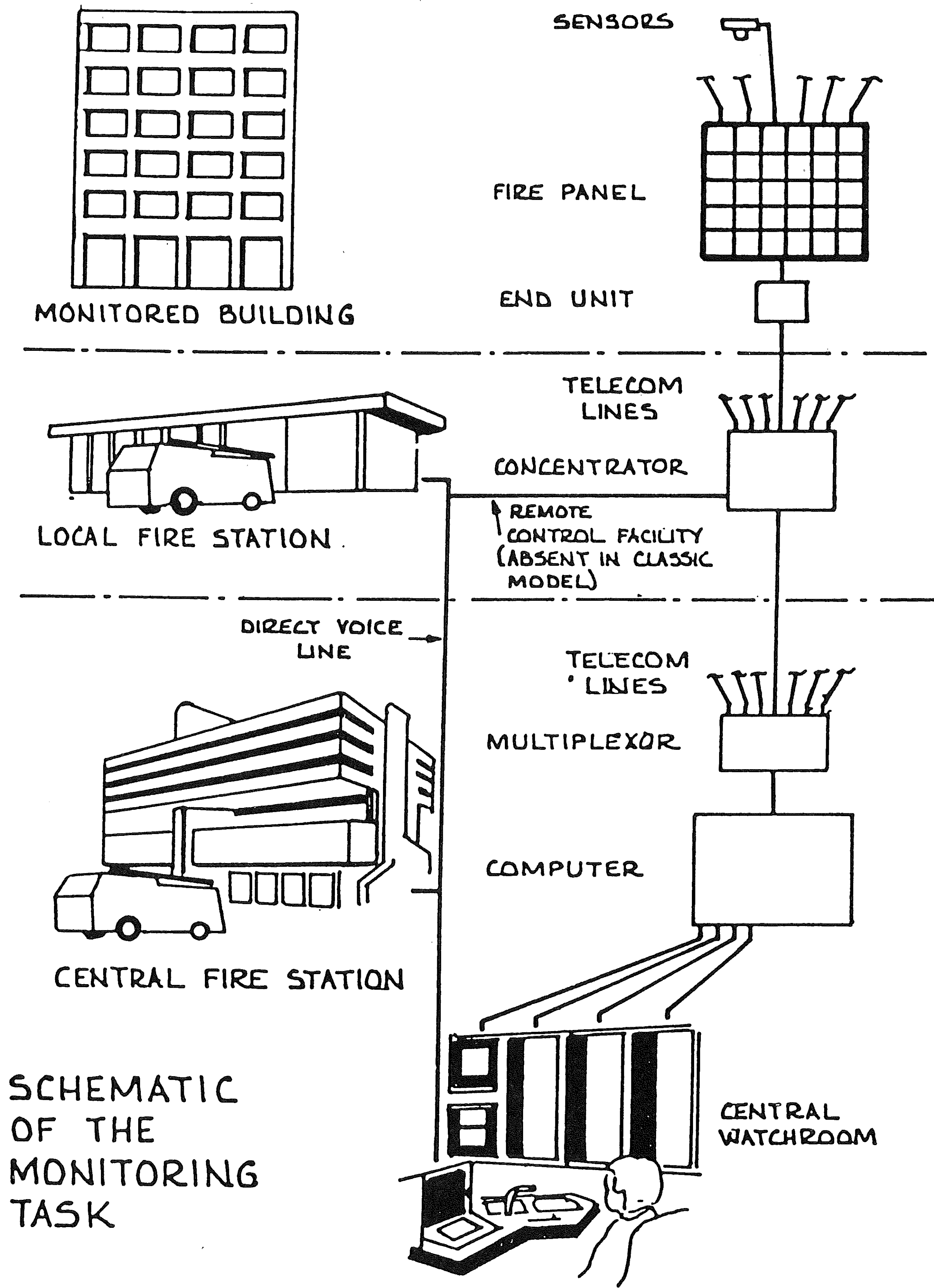


Figure 1

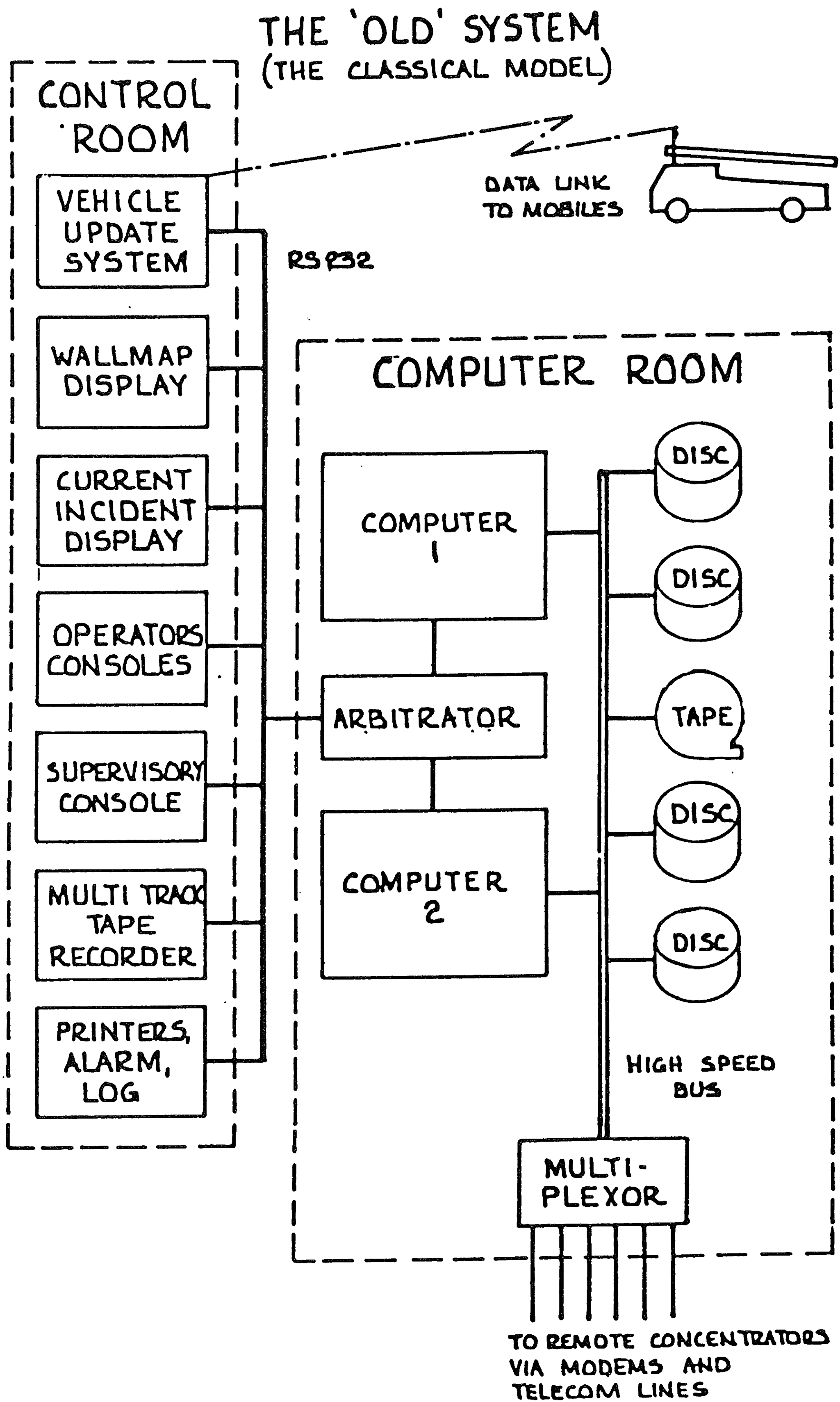
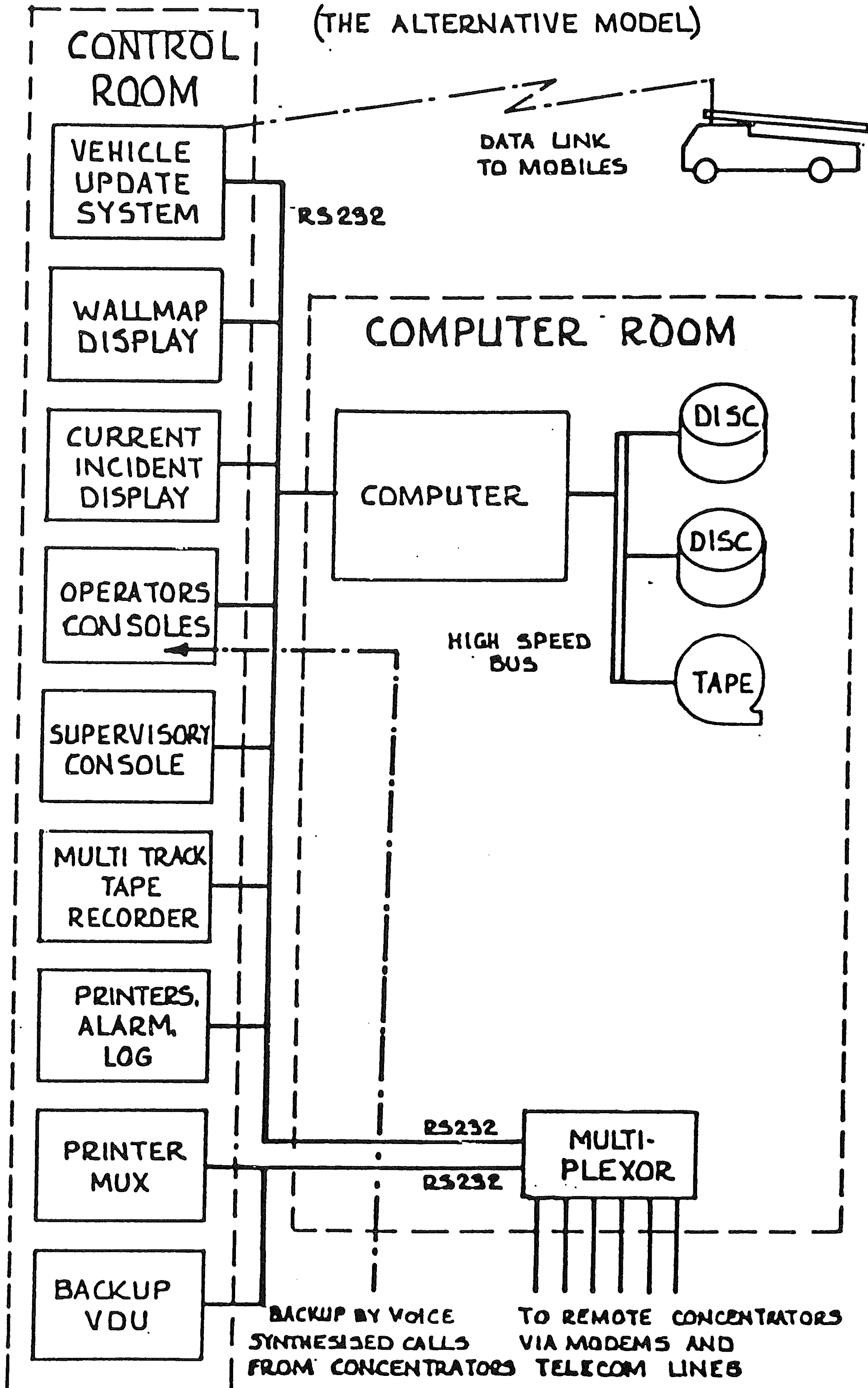


Figure 2


THE NEW SYSTEM
(THE ALTERNATIVE MODEL)
CONTROL ROOM
VEHICLE UPDATE SYSTEM
DATA LINK TO MOBILES
RS232
WALLMAP DISPLAY
COMPUTER ROOM
CURRENT INCIDENT DISPLAY
COMPUTER
DISC
DISC
OPERATORS CONSOLES
HIGH SPEED BUS
TAPE
SUPERVISORY CONSOLE
MULTI TRACK TAPE RECORDER
PRINTERS, ALARM, LOG
PRINTER MUX
RS232
RS232
MULTI-PLEXOR
BACKUP VDU
BACKUP BY VOICE SYNTHESISED CALLS FROM CONCENTRATORS
TO REMOTE CONCENTRATORS VIA MODEMS AND TELECOM LINES

A Controlled, Productive Applications Environment at Esso Australia's Production Department

Steven C. Landers*

For presentation at the Australian UNIX Systems User Group 1986 Summer Meeting, Perth February 10-11 1986.

ABSTRACT

A common problem facing business today is reconciling demand for new applications with the available workforce.

This paper describes an Applications Design Environment that facilitates controlled applications development and dramatically improves developer productivity. The philosophy behind the environment is described along with several of the components.

---

* Senior Computing Specialist,
Production Department,
Esso Australia Limited,
G.P.O. Box 372,
Sale. 3850. Australia

This paper was typeset on a VAX 11/750 running Unix System V, using the Documenters Workbench software, and printed on a Canon-Imagen LBP-10 laser printer.

A Controlled, Productive Applications Environment at Esso Australia's Production Department

January 1986

CONTENTS

1. Overview 1
2. The Background 1
3. Philosophy/Aims 1
4. Analysis of Program Content 2
5. ADE Components 2
   5.1 Command/Option Selection 2
   5.2 Argument/File Selection 6
   5.3 Other Facilities 8
6. Examples of Applications Developed 8
   6.1 Land Rates and Taxes 8
   6.2 WorkCare 9
7. Conclusion 9
8. Acknowledgements 9

LIST OF FIGURES

Figure 1. Example of *Vsh* Menu Description File 3
Figure 2. Example of *Vsh* Display 4
Figure 3. Printer Destination Selection Menu 5
Figure 4. Sample *Man* Command Using *Pick* 5
Figure 5. Example of *Pick* Display 6
Figure 6. Utilities Menu Using *Pick* 7

A Controlled, Productive Applications Environment at Esso Australia's Production Department

Steve Landers

Esso Australia Ltd

1. Overview

The Esso Australia Ltd (EAL) Production Department has been using UNIX™ System V and the ORACLE™ Relational Database System, on a VAX 11/750 since 1983. During this time an Application Design Environment (ADE) has been implemented using UNIX utilities, ORACLE , and locally developed tools and standards.

The ADE addresses many of the problems associated with traditional applications development. Productivity is enhanced whilst maintaining controls and standards.

Acknowledgments are due to my colleagues Steve Adams and Ralph Youie who were part of the team that developed ADE.

2. The Background

Traditionally, the applications design cycle has comprised:

1. The development of a detailed project specification/design.
2. A long period of programming in a compiled language such as C or FORTRAN.
3. Extensive debugging.
4. Commissioning of the application.
5. Major revision based on user feedback.
6. High maintenance requirements.

The effect of this strategy is:

- Lack of standards and controls.
- Diverse user interfaces.
- Long lead times.
- Limited chance for user feedback before commissioning.
- Lower quality applications.
- Low developer productivity.

The use of so-called "fourth generation languages" has improved the developer productivity somewhat. However, standardisation is still an issue: Real programmers can code FORTRAN in any language.

To avoid these problems an Applications Design Environment (ADE) has been implemented on EAL's Gippsland Area VAX 11/750.

The ADE comprises standard UNIX System V utilities, ORACLE , and locally developed tools. ADE was layered on UNIX and required no changes to UNIX .

3. Philosophy/Aims

The philosophy of ADE is essentially the UNIX philosophy. The emphasis is on building applications from a series of tools. The guidelines for such software tools are:

- Each tool performs one clearly defined function, and is optimised for that function.
- All tools are regarded as *building blocks* that may be combined without constraint.
- Software development is reduced so far as possible to assembling building blocks.
- When no suitable tool is available, a new one should be strongly considered for development or acquisition in lieu of building an

* UNIX™ is a trademark of AT&T Bell Laboratories

** ORACLE™ is a trademark of Oracle Corporation

application specific program.

The *tools* approach to software development has minimised duplication of effort and freed programmers to concentrate on analysis and design.

The operating system run on the Area Computer is UNIX System V (Release 2).

The applications development philosophy applied is that of "evolutionary prototyping". As users and programmers alike are often unsure of the scope and requirements of a new application, it is regarded as important that the users be involved in the development cycle. This strategy is aimed at avoiding time wasted in the development of prototypes that must thereafter be "thrown away".

The problem is how to get developers to use the tools, and to overcome the temptation to "re-invent the wheel". The approach used has been termed "Standards by Carrots". There must be a positive incentive for the developer to use the tool because it is quicker, or more convenient; that use of the tool complies with a standard should be incidental.

4. Analysis of Program Content

To ascertain what tools are appropriate one can conduct an analysis of program content. Application program code can be divided into four main categories:

— setup (including argument processing)

— input (both disk file and terminal)

— processing

— output

An analysis of typical programs has shown that processing is often the smallest and certainly the most specific. The other areas, however, offer great potential for the *toolsmith*. Many tools already exist under UNIX . For example:

- *getopt* for command option and argument parsing.
- *curses* for screen based terminal I/O.

All such tools are characterised by the developer describing what needs to be done, rather than how to do it.

ADE contains even higher level tools. These tools include the following:

vsh command/option selection

pick argument/file selection

sel database querying

As the other categories are simplified, the processing can be handled, in most instances, by an interpretive language such as *awk*. This further contracts the development cycle as the developer does not have to wait for compilations.

5. ADE Components

ADE comprises several categories of tools:

— development support tools

— software building blocks

— runtime support

— maintenance tools

We will examine two tools in some detail, and briefly discuss others.

5.1 Command/Option Selection

Command and option selection is performed by the Visual Shell (*vsh*) . *Vsh* was originally designed to insulate novice users from a more typical UNIX shell (eg: Bourne shell). It has subsequently been developed to the stage where is performs additional functions, such as:

— security insurance

— applications usage tracking

— productivity enhancement

— standardisation of user interface

5.1.1 Menu Layout

Vsh displays a menu from which the user may select an action or submenu. The menu description is read from a file. *Vsh* interprets the file and decides the menu layout. A typical *vsh* file is shown

```
menu      Unix Utilities

option    Working directory
action    pwd
pause

option    List file names
action    ls -C | pg -c

option    Long listing
action    ls -al | pg -c

option    View a file
action    pg -c $file_name
help      demonstrates the use of substitution variables

option    Edit a file
action    $${EDITOR:-vedit} $file_name
help      demonstrates the use of shell meta characters

option    Remove a file
action    rm -i $Remove_file
pause

option    Rename a file
action    {
              echo "Rename which file "
              read old
              echo "New name "
              read new
              mv $$old $$new
          }
help      demonstrates compound statements
pause

option    Process status
action    ps -f | pg -c
wait
```

Figure 1. Example of *Vsh* Menu Description File

in Figure 1 and its appearance to the user in Figure 2.

A selection is made by positioning the cursor next to an option (usually with the SPACE bar) and pressing the RETURN key.

Other command keys are:

?	display an explanation of the current option
??	display the help message
^N	toggle NOVICE mode (for verbose help)
^A	invoke a particular application
^U	invoke utilities submenu
!	invoke a UNIX shell
^L	redraw the screen
DEL	backup to previous menu level
^D	exit *vsh* directly
^E	edit menu description file (debug mode)

5.1.2 Menu Description File

If the first part of the menu description file name is of the form "~name", it is expanded to the home directory of the

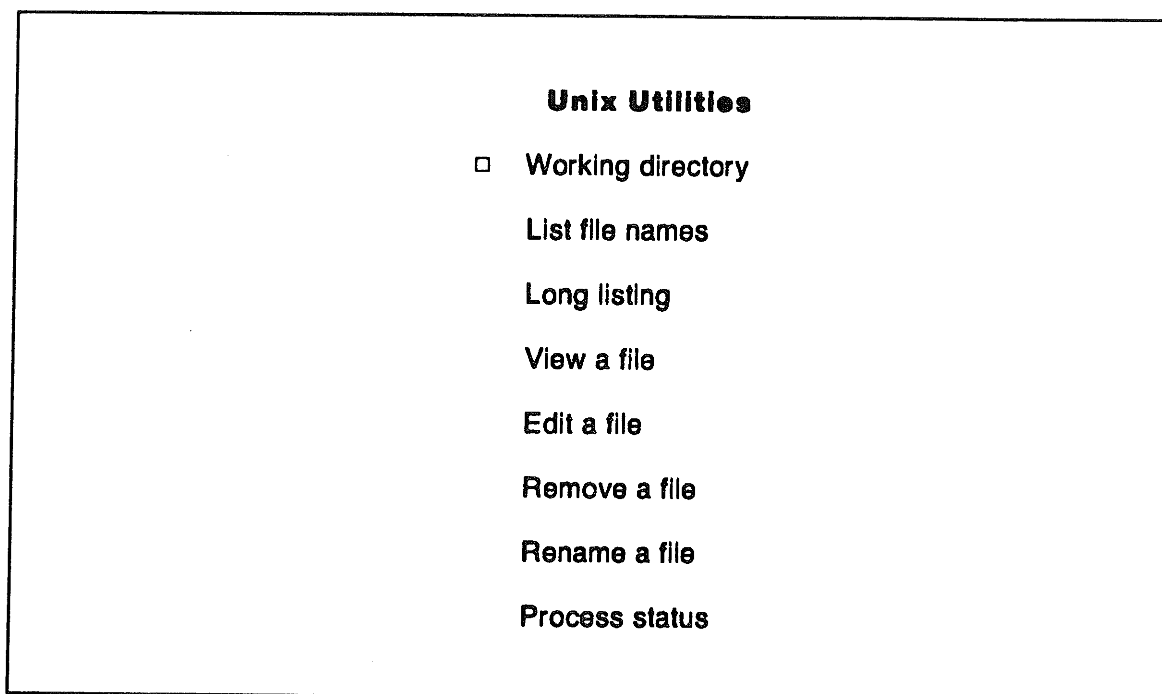


Figure 2. Example of *Vsh* Display

user or group so named.

Vsh changes to the directory in which the file is resident to discourage the use of absolute pathnames.

Menu description files are composed of blank lines and lines of the form:

keyword [value string]

The following keywords are recognised:

menu — Supplies the menu heading

application — Same as menu, but has two additional affects. The **bin** directory under the menu directory is prepended to the PATH environment variable and an entry is made in a usage file.

oracle — Cause *vsh* to prompt the user for a database username and password.

option — The prompt for the option.

help — An explanation of the option or menu.

submenu — Names the menu description file for a submenu. If of the form "~name", it is expanded to the home directory of the user or group so named.

action — The action to be executed. May be any valid shell statement. Actions may include substitution variable that the user is prompted for.

pause — Cause *vsh* to pause after an action.

wait — Cause *vsh* to display the message "Please wait ..." before begining an action.

noreturn — Execute the action but do not return (ie: exec without forking).

return — Exit with the supplied exit status. Allows *vsh* to be used from other programs for menu work.

setenv — Set the specified environment variable.

One of the mutually exclusive keywords action, setenv, submenu and return

```
menu     Select Required Printer

option   Main printer
action   lp -d main
pause

option   Dot matrix printer (draft mode)
action   lp -d matrix -o draft
pause

option   Dot matrix printer (letter mode)
action   lp -d matrix -o letter
pause

option   laser printer (landscape mode)
action   lp -d laser -o L
pause

option   laser printer (portrait mode)
action   lp -d laser
pause

option   Printer attached to terminal
action   {
             tput prtr_on
             cat
             tput prtr_off
         }
pause
```

Figure 3. Printer Destination Selection Menu

```
IFS=":"
export IFS
man `ls $PATH | pick -h "Please select a program"`
```

Figure 4. Sample *Man* Command Using *Pick*

must directly follow an option.

Vsh exercises a degree of intelligence by changing groups where the users is permitted access to the group of the menu directory.

5.1.3 Future Enhancements

The current version of *vsh* cannot be used as a filter. The next version, currently under development, will allow use in a pipeline. An example of the use of such a facility would be a Printer Destination Menu:

```
pr $file | vsh -f printer.vsh
```

where **printer.vsh** is shown in Figure 3. When used as a filter, the *vsh* **noreturn** option would apply to all actions.

5.1.4 Philosophy

The philosophy of providing reusable software components has led to the addition of the *vsh* instructions mentioned above.

For example, the use of the **pause** option, which enables the user to see output before *vsh* clears the screen and

```
Please select a program

□ acctcom
  adb
  ar
  as
  basename
  binmail
  bs
  cat
  cc
  chgrp
  chmod
  chown
  cmp
  convert
  cp
  cpio
  crypt
  csh
  date
At first line
```

Figure 5. Example of *Pick* Display

redisplays the menu, has four affects:

- It saves the developer from writing and debugging a small piece of program code in each application.
- It standardises the user interface.
- It contains a timeout mechanism that the average developer would not provide.
- It makes the user program smaller.

Reusable software is a key factor in reconciling resources with demand. It is also a key to ensuring standardisation and control.

5.2 Argument/File Selection

A common task in an applications program is the selection of arguments or files. Typically, the user is offered a list of several files and prompted for a choice. The program would then act upon the file, (eg: invoking an editor).

Whilst the actual programming to achieve this is not difficult, the task is nevertheless non-trivial. Amongst the tasks to be performed are:

- Read each directory entry.
- Display each file name.
- Maintain a line count so as to avoid terminal scrolling.
- Prompt for the required file (perhaps using a number scheme).
- Recognise if the user has selected a file, or is requesting scrolling to the next page.
- Scroll the display (forward or backward) if necessary.

In a workforce-constrained environment it is not unusual for another approach to be taken: Prompt the user and hope the filename is known. This solution is not satisfactory.

In ADE, argument selection is accomplished by means of the *pick* tool. *Pick* is used to select an input line from a list. The specified input file (or standard input) is displayed on the screen. The user positions the cursor next to the desired argument in much the same way as in *vsh*. Should there be more input lines than can fit on one screen, scrolling is supported. As in *vsh*, the user selects

```
menu    Unix Utilities

option  Working directory
action  pwd
pause

option  List file names
action  ls -C | pg -c

option  Long listing
action  ls -al | pg -c

option  View a file
action  pg -c `ls | pick -h "Please select a file"`
help    demonstrates the use of pick

option  Edit a file
action  $${EDITOR:-vedit} `ls | pick -h "Please select a file for editing"`
help    demonstrates the use of pick

option  Remove a file
action  rm -i `ls | pick -h "Remove which file"`
pause

option  Rename a file
action  {
            old=`ls | pick -h "Rename which file"`
            echo "New name "
            read new
            mv $$old $$new
        }
help    demonstrates compound statements and pick
pause

option  Process status
action  ps -f | pg -c
wait
```

Figure 6. Utilities Menu Using *Pick*

a line by pressing the RETURN key. The selected line is written to standard output. The user is also informed when positioned at the first and last lines.

Other *pick* commands are:

?	display available commands
/	search for a regular expression
f	forward one page
b	backward one page
!	invoke a UNIX shell
^L	redraw the screen
DEL	exit with no selection
^D	exit with no selection
RETURN	select line and exit

Conceptually, *pick* is a data reduction filter.

Now we can apply *pick* to the previously mentioned task: Choosing a file from a list. We could construct a pipeline that uses the *ls* command to list directory entries and pipe this into *pick*.

```
ls | pick
```

Ls would write a list of files in the current working directory to standard output. *Pick* would read this list and display on a scrolling region of the

screen. A default heading would be displayed and the cursor positioned next to the first line.

This scheme could be used to construct a screen based front end to the *man* command, as shown in Figure 4. The screen seen by the user is shown in Figure 5.

Figure 6 shows the utilities menu modified to use *pick* for argument selection.

5.3 Other Facilities

There are many other components in ADE. These include:

appsdir	Prints home directory of an application
asgroup	Executes a command as a given group
dirtree	Prints directory tree schematic
getpass	Program to get a password in shell scripts
group	Prints effective group name
home	Prints the home directory of a user
hs	Horizontal scroll filter
index	Automatically creates an index of all files in a directory
keep	Filter which keeps first n lines on screen
lockedit	Edit a file exclusively
lockfile	Create a lock file, waiting if necessary
mmake	Pseudo make for source directories with multiple load modules. Enforces a standard directory structure.
newer	Test age relationship of files
night	Defer job to offpeak period
pages	Report pagination with multi-line headings
printer	Construct print command string
remask	File system access permission maintenance utility
report	Run a sel/awk report
safety	Print a safety message (**Safety is Esso's number one priority)**
sccs	SCCS user friendly interface program
sel	Database query facility. Returns tables with *TAB* separated columns.
start	Automatic top comment generator
width	Print max line width of stdin or files
xedit	Use $EDITOR on encrypted file
yes	Ask a yes/no question

The key feature of all of these tools is that they perform frequent tasks in a controlled manner.

In addition, there is an Applications Support Facility, comprising an extended data dictionary, applications register and several reporting utilities. These were built using ADE.

Separate test and production ORACLE database systems are run. The database to be accessed is specified by setting an environment variable in the *vsh* applications menu. Prior to installation on the production database, an application undergoes a quality assurance and the details entered into the extended data dictionary. The use of standard software building blocks facilitates the quality assurance phase.

6. Examples of Applications Developed

6.1 Land Rates and Taxes

This is a sophisticated financial analysis and information system for Esso/BHP joint venture owned land.

The system is able to highlight inappropriate increases in land valuations (and hence rates) despite the

basis for valuations changing yearly.

The system was developed in 20 workforce days, including analysis, design, development and documentation. The contractor developing the application had no prior experience with ADE.

6.2 WorkCare

An application designed to fulfill the statutory requirements of self-insurers under the Victorian Government WorkCare legislation.

The application records all workers compensation claims, payments and recoveries. Monthly tapes are prepared for the Victorian Accident Compensation Commission.

Development required 20 workforce days, of which 13 were to develop a 1400 line C program for the preparation of the tape.

7. Conclusion

ADE has been successfully used to develop over 40 applications. The development and maintenance of these applications with a relatively small workforce is testimony to the:

— productivity

— quality

— standardisation

inherent in the use of ADE.

The productivity is achieved by providing tools that address frequent and time consuming tasks.

The quality is achieved by tools that perform one clearly defined function well.

Standardisation is achieved by tools designed to implement standards without affecting flexibility and productivity.

8. Acknowledgements

The work described in this paper includes contributions from a variety of EAL and contract staff. The suggestions of the following are gratefully acknowledged:

M.S. (Steve) Adams§
K.D. (Ken) Doig §
D.W. (Bill) Goss
M.E. (Mark) Horn §
S.W. (Steve) Shepard
R.A. (Ralph) Youie

§ Contract

CNFX - File Transfer Between UNIX systems and CSIRONET.

Jeremy Firth and Jennifer Hudson

CSIRONET, c/- CSIRO Tasmanian Regional Laboratory, Stowell Avenue, Battery Point, Tas. 7000.

INTRODUCTION

There is a continuing need for UNIX sites to communicate with CSIRONET hosts and peripherals. Various pieces of software already exist to enable data to be transferred between particular CSIRONET file systems and UNIX. Perhaps the best known of these are **ugate** and **csiro**. On the CSIRONET side they tend to rely on various bongles and fudges such as fetching a file by logging into CSIRONET ED on a Cyber and giving ED the *@p* instruction then collecting whatever emerges on the line from CSIRONET.

Towards the end of 1985 CSIRONET introduced a File Transfer and Spooling system (called CNFTS) which is available on all major CSIRONET hosts. In doing so the spooling function on CSIRONET has been decentralised from being a single set of queues on a central host (the Cyber 76 - now pensioned off) to several sets of queues, one set for each CSIRONET host. Any host equipped with CNFTS can send files to or fetch files from any other host also equipped with CNFTS. Apart from interhost file transfers, all spooling to peripherals at CSIRONET nodes (such as printers and plotters) is performed by CNFTS.

Integral to the concept of CNFTS is the idea of 'auxiliary' connection of various kinds of systems to CNFTS for file tranfer purposes. In these cases, file sends and fetches can be initiated from the auxiliary system end only. Unsolicited file transfers cannot be initiated from the CNFTS host at the other (CSIRONET) end.

One large, identifiable group of users and potential users of CSIRONET in this auxiliary connection category is the UNIX world. CNFX is the software module intended to satisfy the needs of this group. Other systems for which this software is being converted are the Convergent Technology NGEN running CTOS which CSIRO Administration has selected as its preferred microcomputer, and the IBM/PC running PC/DOS and its extended family.

FUNCTIONAL DESCRIPTION

CNFX consists of 3 components:

1. The UNIX user/file interface which interprets the 4 CNFX commands and their associated parameter strings, and accesses the local UNIX file system.
2. The transfer program which executes the commands by interacting with CNFTS. This part of the software hides the CNFTS transfer protocols.
3. The mechanism by which the data is actually transferred to and from CSIRONET. The functionality of this component is provided by the **mx** logical multiplexer (sometimes referred to as the **xt** driver on 4.2BSD).

All transfers must be initiated from the local UNIX system end. The

actual transfer process is made via a CNFTS queue on a specified CSIRONET host. CNFX transfers from one UNIX system to another must be performed in two steps: the sending UNIX system transfers the file to a CSIRONET host, and the receiving UNIX system can then fetch the file from that host.

File transfers from a UNIX system to CSIRONET occur in one step using the **cnsend** command. Options allow the UNIX user:

. to make a new file on a CSIRONET host,
. to append to or replace an existing file on a CSIRONET host,
. to submit a file as a job to be run on a CSIRONET host,
. to spool a file to a CSIRONET output peripheral.

File transfers from a CSIRONET host to a UNIX system occur in two steps. The **cnfetch** command transfers a file from a CSIRONET host file storage system to an CNFTS queue. The **cnpoll** command actually transfers the file to the UNIX system.

Although not really a file transfer facility, a further function is available in CNFX called **cnlogin.** As its name implies, this provides interactive access to CSIRONET. **cnlogin** operates in 2 modes.

In *C-mode* it makes a UNIX terminal into a CSIRONET terminal. Lines are delimited by CR/LF sequences, *dle* sequences are interpreted in the CSIRONET vernacular.

In *U-mode,* **cnlogin** makes a local UNIX terminal into a remote UNIX terminal to enable a user to log into another UNIX system also connected to CSIRONET. In *U-mode,* lines are delimited by a single carriage return. All ASCII characters with a value less than 40 octal are transmitted over CSIRONET as soon as they are received (except for the *dle* character which is handled separately).

OTHER FEATURES

The CSIRONET login sequence used by CNFX may be specified either as an element in the CNFX **site.h** file (in which case it is hidden from UNIX users) or as a parameter to a CNFX command. Anything that is missing (e.g. passwords) is prompted for. Several CNFX sites have found it convenient to have a shell script (called **cnprint**) with fixed CNFX parameters to enable users more easily to dispose files to be printed at a specified CSIRONET node.

An *end-to-end* packeting facility is in the final stages of development (hopefully it will be ready by the time this paper is presented). If invoked, this allows checks to be made to ensure correct data delivery, regardless of the underlying link conventions. The packeting facility is not by itself suitable for use over unreliable links because it does not provide a mechanism for error recovery. It can be used over a non-transparent link, provided such a link can carry a full set of ASCII printed characters. The packet format is based on KERMIT. The only incompatible change is the use of a 2-byte packet length field, permitting packets longer than 94 characters. Given that the UNIX version of CNFX uses the **mx** driver this facility may not be of great interest to the UNIX fraternity. It is mentioned here for completeness.

A spooling option has been provided to allow **nroff** output files to be disposed to CSIRONET printers.

Other features currently on the development wish list include:-

. An interrupt facility to enable interactive CSIRONET sessions via **cnlogin** to be echoed selectively to a file on the UNIX system. This is already available in **ugate**.
. An interface between the UNIX mail network and CSIRONET MAIL.

INSTALLATION AND PERFORMANCE DETAILS

CNFX was originally developed for UNIX level 7. A version is also available for Berkeley 4 2BSD. A version for UNIX V.2 is currently being developed.

In order to run CNFX, it is necessary to have an **mx** driver on the UNIX side, and a CSIRONET node with UNIX gateway code installed in it. A CSIRONET Micronode version of the existing PDP11 gateway code will be available in February 1986.

The method of installation is for an encrypted set of source modules to be supplied to the UNIX site on a **tar** tape. CSIRONET then installs the software remotely using an account on the local UNIX system. When all is ready, the final object code has to be installed by **root**. UNIX Programmers Manual entries are also provided.

To obtain CNFX the formal contact is to:
The Chief Executive
CSIRONET
G.P.O. Box 1800
CANBERRA. ACT 2601.

Further informal information can be obtained from the authors at
CSIRONET, Hobart
(Tel. 002-201444 or CSIRONET MAIL
Aliases JEREMY and JENNIFER)

The charge for CNFX is $350.00 for installation and $300 p.a. ($25 per month) for maintenance.

Using a PDP11 as the CSIRONET gateway with a 9600 bps asynchronous line to the **mx** port on a PDP11/34 running UNIX level 7 we have achieved transfer rates of up to 10K bytes/minute.

ACKNOWLEDGEMENTS

The authors embarked on this project having little but a theoretical appreciation of UNIX, mostly absorbed from the professional literature. They wish to note with appreciation the continuing and patient assistance of John Field from the CSIRO Division of Mathematics and Statistics in Adelaide in helping us convert to the UNIX view of the world. DMS Adelaide also provided us with beta-test facilities. The help on the data communications side of things was by John Gibbons of CSIRONET Sydney. Warwick Ford of CSIRONET Canberra assisted with the intricacies of CNFTS. The above mentioned plus John Paine from CSIRONET Canberra, contributed comments on early drafts of this paper. Nevertheless, any errors of omission or commission are the sole responsibility of the authors.

APPENDIX: EXTRACTS FROM UNIX PROGRAMMERS MANUAL ENTRIES FOR CNFX COMMANDS

NAME

cnsend - send a file to a CSIRONET host or output peripheral

SYNOPSIS

cnsend [options] file

DESCRIPTION

cnsend transfers a local file to a remote host or CSIRONET output peripheral. The file may be a sequence of job control language (jcl) statements for the remote host to execute, or it may contain data for storage on a remote host file system, or it may be intended for disposal to a CSIRONET output peripheral.

Note that the arguments **-m**, **-r**, **-a**, **-j**, and **-n**, which indicate the purpose for which the file is being sent, are mutually exclusive. One and only one may be included as a parameter for **cnsend**. Other arguments are also only relevant for certain types of destinations, but should not be specified for other types. Most arguments have defaults and are not explicitly required very often.

If the file being sent is a job containing jcl statements to be executed on the remote host, only its name is necessary. Dayfile output will normally be returned to the default destination specified in the CSIRONET login sequence (this will either be a CSIRONET node peripheral or a CNFTS queue name such as PUB).

If the file is being sent to a remote host file system, a file name must be given, with volume, user name and cycle number where appropriate. Passwords may also be included if necessary.

If the device is a CSIRONET output peripheral, the destination node mnemonic must be given. If the peripheral is not a line printer, the device type is also required.

If the file name is omitted from the **cnsend** command, it is assumed to be standard input.

OPTIONS

-m*name* Make a new file called *name* on a remote CSIRONET host.

-r*name* Replace an existing file called *name* on a remote host if it exists, otherwise make a new file.

-a*name* Append contents of the transferred file to an existing file called *name* if it exists, else make a new file.

-p*pw* Remote file password. If no *pw* is supplied, the user will be prompted for a password.

-v*vol*	*vol* is the volume, or similar file structuring feature on the remote CSIRONET host where the transferred file is to be located.
-y*cycle*	*cycle* is the cycle number of *name*.
-x*so*	Other remote host specific options. Each element is separated by whatever is the delimiter for jcl elements on that host. For hosts that use a blank as the delimiter between elements of a jcl statement, the *so* string should be in quotes.
-b	The file *name* contains binary data and is to be handled in a transparent manner, with no code changes, on the remote host and during transfer processing.
-u*user[/pw]*	*user* is the username on a non-CSIRONET remote host, *pw* is the password for the username. If no password is given, the user is prompted for the password, which can be given interactively with no echo on the terminal.
-j	The UNIX file given as a parameter to this command contains jcl statements to be submitted for execution by the remote host.
-n*node[,dt]*	Destination CSIRONET node and device type where no *dt* is specified, the default is LP.
-c	Forces the character set for storage of the transferred file on the remote host to be 7-bit ASCII (with the 8th bit set to zero). If omitted, the default is the standard or native code for that remote host.
-f*fc*	The CSIRONET forms code (varies depending on device type).
-t*[nn]*	If present, this parameter indicates that line printer carriage control is to be taken from col 1 of each record. It can only be used if *dt* = LP (see the **-n** parameter, above). If *nn* = cc (or is blank) carriage control is taken from col 1 of each record. If *nn* = bs, then embedded backspaces are converted into overprinted lines so that **nroff** output can be printed on CSIRONET node printers. (Warning: using the **-t**bs option requires many wheels to go round and may be quite slow!)
-z*cnlog*	*cnlog* is a UNIX file in which a CSIRONET login sequence may be found. If the filename *cnlog* is omitted, the user will be prompted for elements of the login sequence.

EXAMPLES

cnsend -nSY printfil

Send the file **printfil** to the line printer at CSIRONET node SY (CSIRONET Sydney).

man 1 cnfx | cnsend -nhh -tbs
Print the **nroff** output for the **UNIX** Programmer's Manual entry for **cnfx** at node HH (CSIRONET Hobart).

cnsend -mQDATA /usr/user/survey.qdata
Send the file **/usr/user/survey.qdata** to the default remote host, and store it as a file called QDATA.

cnsend -j -zM180 facomjcl
Submit the jcl statements contained in the file **facomjcl** as a job on the remote host specified in the **M180** login sequence.

NAME

cnfetch - request the transfer of a file from CSIRONET.

SYNOPSIS

cnfetch [options]

DESCRIPTION

cnfetch fetches a file from a remote host file system and places it in a CNFTS queue. A file thus fetched can be retrieved from the CNFTS queue by using the **cnpoll** command.
cnfetch always uses the queue relevant for the remote host specified in the *cnlog* sequence.

OPTIONS

-e*name* File name on a remote CSIRONET host.

-p*pw* Remote file password. If no *pw* is supplied, the user will be prompted for a password.

-v*vol* *vol* is the volume, or similar file structuring feature on the remote CSIRONET host where the transferred file is located.

-y*cycle* *cycle* is the cycle number of *name*.

-x*so* Other remote host specific options. Each element is separated by whatever is the delimiter for jcl elements on that host. For hosts that use a blank as the delimiter between elements of a jcl statement, the *so* string should be in quotes.

-b The file *name* contains binary data and is to be handled in a transparent manner, with no code changes, on the remote host and during transfer processing.

-u*user[/pw]* *user* is the username on a non-CSIRONET remote host, *pw* is the password for the username. If no password is given, the user is prompted for the password, which can be given interactively with no echo on the terminal.

-c Forces the character set for storage of the transferred file on the remote host to be 7-bit ASCII (with the 8th bit set to zero). If omitted, the default is the standard or native code for that remote host.

-z*cnlog* *cnlog* is a UNIX file in which a CSIRONET login sequence may be found. If the filename *cnlog* is omitted, the user will be prompted for elements of the login sequence.

EXAMPLES

cnfetch -eDATA01 -pXXX -vCCS123
Fetch a file named DATA01 on volume CCS123 with file password XXX from the

default remote host.

cnfetch -eMYFILE -z/usr/user/NOSlogin; cnpoll NOS.dayfile

Fetch a file called MYFILE from a remote host specified in **/usr/user/NOSlogin**, poll the relevant CNFTS queue, if a file is there, put it in a file called **NOS.dayfile** in the current directory.

NAME

cnpoll - poll a CSIRONET CNFTS queue.

SYNOPSIS

cnpoll [options] file

DESCRIPTION

cnpoll polls a CNFTS queue exactly once. Any file that is waiting will be transferred to the local UNIX system as file. If file is omitted, standard output is assumed.

OPTIONS

-q*qname* — Name of a CSIRONET CNFTS queue. With each UNIX system which has **CNFX**, is associated a logical CNFTS queue mnemonic (usually of 3, but may be up to 7 characters). This mnemonic usually will have the form *nnn*FX where *nnn* is a logical CSIRONET node mnemonic associated with the local UNIX system.

-z*cnlog* — *cnlog* is a UNIX file in which a CSIRONET login sequence may be found. If the filename *cnlog* is omitted, the user will be prompted for elements of the login sequence.

EXAMPLES

cnpoll

Look for a file in the default CNFTS queue. If there is a file in the queue, transfer it to standard output (on the local UNIX system).

cnpoll NOS dayfile

Look for a file in the default CNFTS queue. If there is a file there, transfer it to the local UNIX host as a file called **NOS.dayfile** in the current directory.

cnpoll -q999FX >> bond.c

Poll the queue 999FX on the CSIRONET host specified in the *cnlog* sequence. If a file is waiting, append it to the local UNIX file **bond.c** in the current directory.

NAME

cnlogin - log in to CSIRONET as an interactive terminal

SYNOPSIS

cnlogin [**-z***cnlog*]

DESCRIPTION

cnlogin has 2 modes of operation:

C-mode This is the default. It should be used for CSIRONET login activities, and for connections to remote CSIRONET hosts.

U-mode This should be used once a connection has been made, via CSIRONET, to a remote unix host.

Once connection has been established, all characters to and from the local UNIX device are dealt with transparently (in RAW mode). No UNIX metacharacters will be recognised locally until the device has been disconnected from CSIRONET. To disconnect from a remote host, enter <dle>T or the logout command for that host. The connection to CSIRONET remains until cntrl-y is entered. For details of how to use CSIRONET, refer to Edition 3.0 of the CSIRONET Users' Manual, particularly Vol.2: 'Network Users Manual'.
In C-mode, lines are delimited by a <CR><LF> sequence for transmission over CSIRONET. Bells, carriage returns, linefeeds and <dle> sequences are all interpreted in the CSIRONET vernacular. To change (back) to C-mode, enter a <dle>C.

In U-mode, lines are delimited by a single <CR> for transmission over CSIRONET to the remote host (almost certainly another UNIX host). To change to U-mode enter a <dle>U. This should not be done until all login formalities to the remote UNIX system have been completed. Once in U-mode, all ASCII characters with a value of less than 040 (octal) are transmitted immediately they are received by the CSIRONET gateway software (except for <dle> itself, which is handled separately).

If the **cnlogin** command has no accompanying **-z** parameter, the user is simply connected to CSIRONET. If the **-z** parameter is present, a login sequence will be prepared and sent to CSIRONET before the user's terminal is connected to CSIRONET.

OPTIONS

-z*cnlog* *cnlog* is a UNIX file in which a CSIRONET login sequence may be found. If the filename *cnlog* is omitted, the user will be prompted for elements of the login sequence.

Folding Regular Polyhedra

Andrew Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This report describes a system for constructing a solid object from a specification of its planar net. The system can handle nets of polygons which may overlap. As an example, the nine regular polyhedra and the Archimedean polyhedra are constructed from their nets. A film has been made showing the construction process and some completed solids.§

Introduction

Many three dimensional objects are fabricated from planar shapes by folding along straight lines and then bonding the edges that coincide. The problems in describing the planar layout and assembly of these objects concern fields as diverse as sheet metalwork and geometric model-making. Since I am more adept with scissors and paper than with a soldering iron, this report will focus on the latter although all the techniques apply equally well to the former. In particular, the focus is on *polyhedra* which can be considered as three-dimensional shells made up of planar polygonal faces.

The end product of this work is a database of polyhedra, each of which is described as a planar net and as a three dimensional object. The polyhedra include the nine regular polyhedra, thirteen Archimedean polyhedra, thirteen Archimedean duals, and 92 other convex polyhedra described by Johnson[3], hereafter referred to as Johnson solids. The Johnson solids, combined with the Archimedean polyhedra and five of the regular polyhedra, constitute all the possible convex polyhedra with regular faces*.

Regular Polyhedra

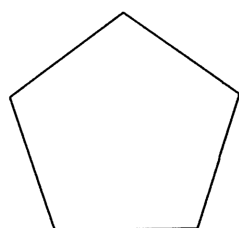
Before discussing regular polyhedra, we shall define a regular polygon as a polygon with a finite number of equal sides and equal vertex angles. According to taste, the sides may or may not be allowed to intersect. Figure 1 shows the two regular five-sided polygons, the pentagon and the pentagram.

We can define a regular polyhedron analogously to a regular polygon; a finite number of congruent regular polygon faces and equal dihedral angles (the angle formed by two faces meeting along an edge) not equal to π radians. A more complete and rigorous treatment can be found in [1].

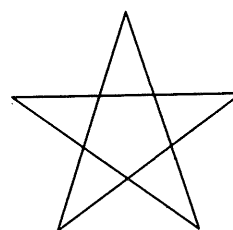
Nine solids meet this definition: the first five are the Platonic solids, which were known before the time of Euclid, the tetrahedron, cube, octahedron, dodecahedron and icosahedron. The other four are known as the Kepler-Poinsot solids and have intersecting faces. The small and great stellated dodecahedrons were discovered by Kepler (1571-1630) and have pentagram (star) faces. The great dodecahedron and great icosahedron were found

§this paper (and film) was originally presented at the Usenix Graphics Conference at Monterey, California in December 1985.

*excluding the two infinite classes described below (prisms and antiprisms).



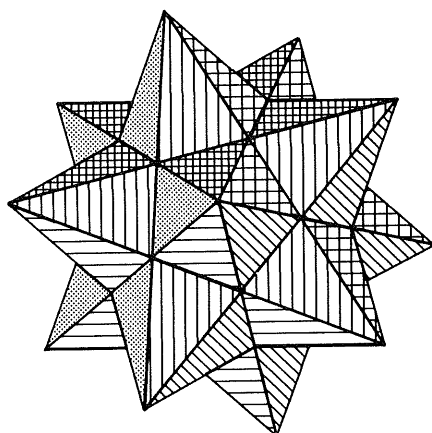
pentagon



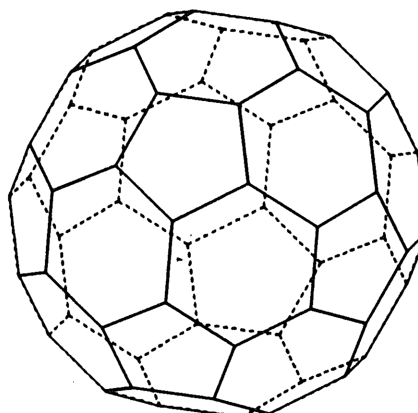
pentagram

Figure 1

by Poinsot (1777-1859) and have pentagonal and triangular faces respectively. The small stellated dodecahedron (shown in Figure 2) and the great dodecahedron are peculiar as they do not satisfy the Euler theorem $F + V = E + 2$ in its normal form.



small stellated dodecahedron



truncated icosahedron

Figure 2

Uniform Polyhedra

Another way to define a regular polyhedron is that its vertex figures are congruent regular polygons. The vertex figure is a polygon formed by joining the midpoints of all the edges meeting at that vertex. Other classes of less regular polyhedra can be generated by relaxing this definition. If the faces are all regular and the vertex figures congruent but not regular, then we get the (facially-regular) Archimedean solids. In this case, the dihedral angles are only equal between congruent pairs of faces. If the vertex figures are regular but not all congruent and all the dihedral angles are equal, then we get the (vertically-regular) Archimedean duals. For historical reasons, the Archimedean duals are largely ignored in discussions of uniform polyhedra.

The Archimedean solids can be divided into three classes. The first class is an infinite set of prisms formed by taking two regular congruent parallel faces and joining corresponding edges with squares. The second class is an infinite set of antiprisms. To construct the antiprism of a given prism, consistently divide each of the rectangular side faces into two triangles by a diagonal and then rotate one of the two regular congruent faces* in the direction such that all the triangular side faces are congruent. The third class is a set of thirteen solids

*through $\frac{\pi}{n}$ where n is the number of sides in the rotated face.

ranging from 8 to 92 faces. One of these, the truncated icosahedron†, is shown in Figure 2.

The dual of an Archimedean solid *s* has its faces associated with the vertices of *s*. The shape of the face is taken by constructing the dual of the vertex figure at any vertex. The dual polygon is formed by drawing the circle containing all the vertices of the vertex figure and drawing the tangents to this circle at each of the vertices. These tangents are the sides of the dual polygon.

The Database

The database is built in three steps. The first and most tedious step is to describe the planar net of the polyhedron. The second determines the dihedral angle between the faces. The third step combines the information from the first two steps and generates a three dimensional description of the polyhedron.

The Net Database

The net or development of a solid is a set of polygonal faces and a description of which edges are joined together. For polyhedra which are not convex, we need to indicate reverse folds. Thus, each edge connection has a sign showing the direction of the fold. The nets for the cube and dodecahedron are shown in Figure 3.

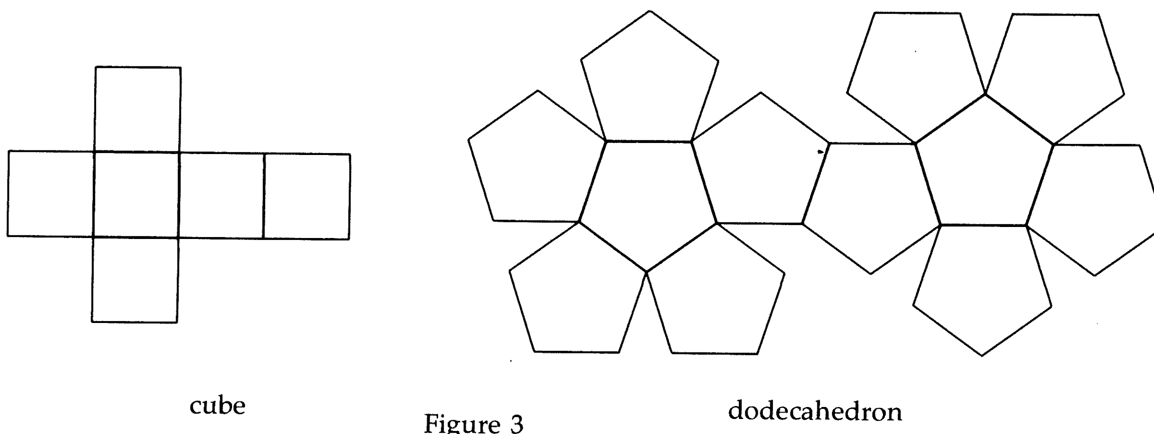


Figure 3

The net description forms an ASCII file. The descriptions for the above nets are:

```
solid "cube"
0:{4}~45
1:{4} 2:{4} 3{4} 4:{4} 5:{4}
<0.0 2.2> <0.1 3.3> <0.2 4.0>
<0.3 1.1> <5.0 4.2>
```

```
solid "dodecahedron"
0:{5}~180
1:{5} 2:{5} 3:{5} 4:{5} 5:{5} 6:{5}
7:{5} 8:{5} 9:{5} 10:{5} 11:{5}

<0.0 4.2> <0.1 5.2> <0.2 1.2>
<0.3 2.2> <0.4 3.2> <5.4 6.4>
<11.0 10.2> <11.1 6.2> <11.2 7.2>
<11.3 8.2> <11.4 9.2>
```

Solids are described by a label, a set of face descriptions and a set of edge connections. Faces are shapes (typically regular *n*-gons denoted by $\{n\}$) followed by an optional rotation in degrees*. Connections are denoted by two edge specifications. An edge is defined by $f.e$ where f is the face and e is the edge on that face. Edge e connects vertices e and $(e+1)\bmod n$

†Yes, it really is a soccer ball! It is also the shape of a new molecule C_{60} (*Nature*, **318**, 162-163, 1985).
*one rotation is useful for orienting the net. Other rotations can be useful as assertions as incorrect rotations are flagged.

where n is the number of sides of the face. Vertices are numbered counter-clockwise with zero at the north pole (before rotation). Reverse folds are denoted thus $<f.e \wedge g.h>$. Faces which are not regular n-gons are specified by their side lengths and vertex angles and denoted by an alphanumeric identifier.

The Angler

The angler program determines dihedral angles by attempting to fold a planar net into a convex polyhedron. Non-convex polyhedra have to be done by hand. Of course, for the regular and Archimedean solids considered here, there are often simple and elegant constructions for the dihedral angles. However, the folder is designed for the general case. A top level view of the algorithm is:

```
form perimeter
while perimeter not empty
{
        pick a vertex v
        form a solid vertex at v
        update perimeter
}
```

The perimeter is kept as a linked list of edges in counter-clockwise order. Given a vertex, the folder assumes that the two edges on the perimeter which meet at that vertex will coincide. Using the methods described below, it calculates the dihedral angles and rotates the faces (and any other attached faces). The perimeter is then checked and edges that coincide are deleted. Note that this may cause the perimeter to become a number of isolated edge lists.

A vertex is selected according to the heuristic of finding the vertex with the minimum angle between the two free edges. This heuristic may fail as sometimes a missing face will come from another part of the net during folding. For example, the nets for the snub cube and snub dodecahedron have large gaps that are filled by faces from another part of the net. This only becomes apparent after partial assembly. If the angler detects an error or inconsistency, it backtracks and selects another vertex.

Vertex selection for the Johnson solids is done differently. If n faces meet at a vertex, we say that vertex is of order n. If we are determining the dihedral angles between the faces meeting at a vertex of order n, there are at most $n-3$ degrees of freedom. Thus, vertices of order 3 have a unique solution. Because of the symmetries of the Archimedean solids and their duals, all the order 4 and 5 vertices have unique solutions as well. The Johnson solids do not have these symmetries. However, in most cases, by solving the order 3 vertices first, we can reduce the degrees of freedom at higher order vertices to zero and thus solve the entire solid.

In general, the angler checks itself after every vertex it forms and in case of errors, it dumps out the current structure and the perimeter lists of unjoined edges. This catches both bad vertex selection and impossible nets, that is, nets that do not form a solid.

Determining the Dihedral Angle

If all the faces are regular polygons, every vertex is of order 3, 4 or 5; examples are shown in Figure 4. Because the sum of the angles at the vertex must be less than 2π and the smallest angle in any regular polygon is $\pi/3$, the number of faces must be less than 6.

This is not true of the Archimedean duals as their faces are not regular polygons and in fact the hexakis icosahedron (the dual of the great rhombicosahedron) has vertices of order ten. However, as all the dihedral angles for the Archimedean duals are equal, we can use the vertex of mimimum degree. By inspection, the maximum degree vertex is 5 (in the pentakis dodecahedron, the dual of the truncated dodecahedron).

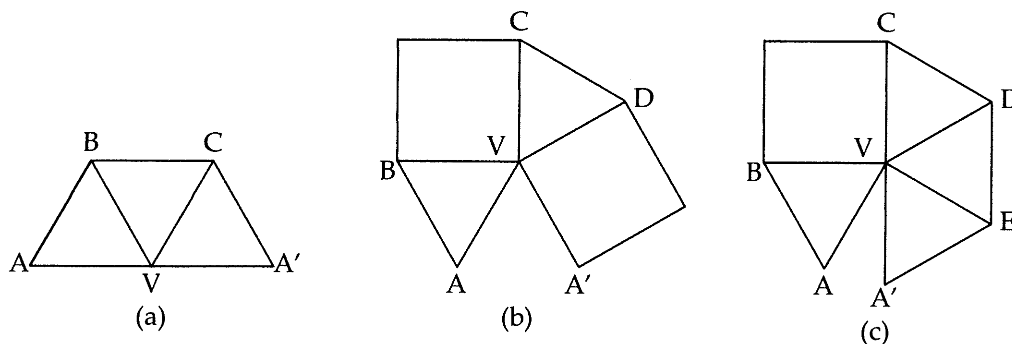


Figure 4

Vertices of order 3 are straightforward to solve. In the example in Figure 4(a), the face **VAB** is rotated about the edge **VB** and **VA′C** about **VC** until the vertices **A** and **A′** coincide.

Vertices of order 4 form a quadrilateral pyramid with one degree of freedom. As discussed above, for the Archimedean polyhedra and their duals, symmetries reduce the number of possible cases to the following few. In Figure 4(b), the apex will be **V** and the (rectangular) base **ABCD**. The base will always be one of four shapes. These shapes are shown in Figure 5(a) and 5(b). Assuming all the edges are of length 1, the parameter z is one of two values, $\sqrt{2}$ or ϕ, where ϕ is the golden ratio ($\approx$1.618). The two dihedral angles can be found easily by spherical trigonometry.

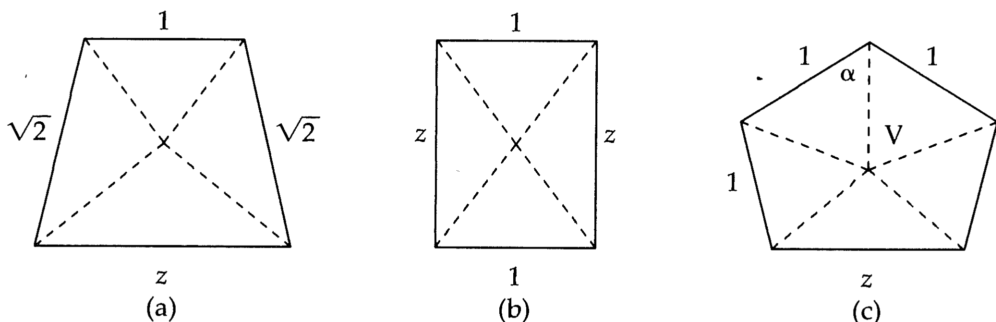


Figure 5

Vertices of order 5 are treated analogously to vertices of order 4. In the example of Figure 4(c), the pyramid has apex **V** and base **ABCDE**. The shape of the base of the pyramid is shown in Figure 5(c). The half-angle α can be found by solving the cubic $\sin\alpha - \sin 3\alpha = \frac{z}{2}$.

The Folder

The folder program treats the polyhedron as a set of hinged shells. Initially, every face is a shell. At every hinge, one of the two shells is rotated through the specified angle. Checks are made for coinciding edges and all shells with coincident edges are merged into one shell. At the end, there should be only one shell.

Current Status

The database has complete entries for the regular, Archimedean and Archimedean dual polyhedra. Nets have been entered for all of the Johnson solids and some of these have dihedral angles and have been folded. The numerical error in the coordinates (stored on a VAX as double precision) is smaller than 10^{-14}.

Future work goes in two directions: determining an algorithmic way to find the dihedral angles for awkward Johnson polyhedra and presenting the data for each

polyhedron symbolically. For example, specifying the dihedral angle for the dodecahedron as $\pi - \tan^{-1}2$ rather than the standard 116° 34′.

Conclusions

It is possible to construct a variety of solids using only their planar nets and with no knowledge about the final structure other than convexity. In addition, many solids (such as the regular stellated polyhedra) with concave vertices can be handled by means of reverse folds.

Indeed, constructing the solids by folding rather than by analytic methods has some advantages, as the mapping between faces in the planar and solid representations is directly derived. For example, given a geographical database, it is possible to project a world map onto any of the convex regular polyhedra. Each face is tangential to the inter-sphere and using a gnomonic projection, we can obtain nets that fold into polyhedral world globes. Such a net for the truncated icosahedron is shown in Figure 6.

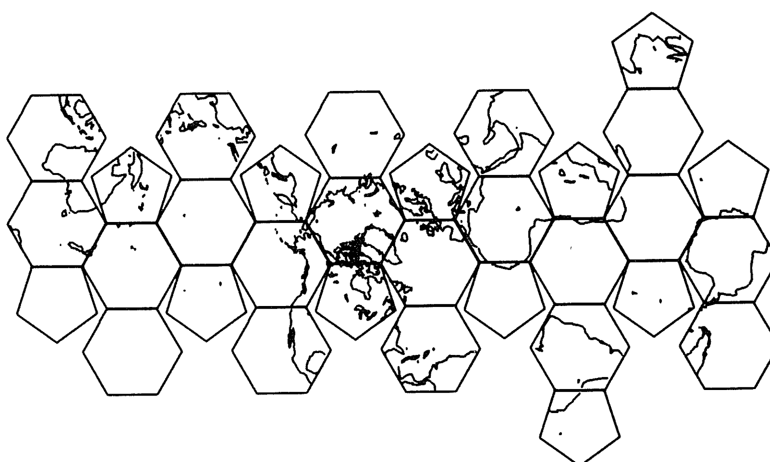


Figure 6

Acknowledgements

My interest in polyhedra and models stems directly from a fine book on geometric models by Cundy and Rollett[2]. Doug McIlroy provided encouragement and the map software. Tom Duff wrote the polygon renderer used in making the film. Peter Weinberger provided his inspiring visage.

References

[1] Coxeter, H.S.M., *Regular Polytopes* (2nd Ed), Macmillan, New York, 1963.

[2] Cundy, H.M. and Rollett, A.P., *Mathematical Models* (2nd Ed), Oxford University Press, Oxford, 1961.

[3] Johnson, N.W., Convex Polyhedra with Regular Faces, *Canadian Journal of Mathematics*, **XVIII**, 1966.

Initial experiences with a Gould UTX/32 System in a University Environment

Bob Buckley
Computing Lecturer

School of Mathematics and Physics,
Macquarie University,
North Ryde,
NSW, 2113.
ACSnet: bob@mqcomp

ABSTRACT

This article describes our first semester with a Gould PN6080 and UTX/32 (a version of 4.2BSD). The computer is used for teaching and research. Most of the student programming is done in Pascal and FORTRAN-77. Major student assignments were about compilers and GKS graphics.

Introduction.

In 1985 the Computing Discipline at Macquarie University was able to purchase its first computer. The computer was intended to relieve the load put on the central facilities by our advanced students. We needed a computer capable of supporting 30-40 simultaneous users. A computing laboratory was planned with about 25 terminals, including 5 graphics terminals.

A Gould PN6080 and UTX/32 was priced attractively and the system is suitable for most of our research activities (particularly numerical analysis and graphics). Languages were to include Pascal, FORTRAN-77, MODULA-2, COBOL and APL. Other software included GKS graphics and a relational database. Five APC-IIIs were purchased as graphics terminals. Most of the hardware was delivered on schedule. Delays in getting operational were caused by late delivery of software and the University's lack of resources.

The Hardware.

The processing engine is based on bit-slice technology. The CPU executes most instructions in the 150ns cycle time. The instruction set is reminiscent of the IBM360. The PN6080 is a dual processor system. Each processor has a

floating point accelerator. We have 4Mb of memory, two 340Mb winchesters, a 1600bpi streaming tape drive and an Ethernet interface.

We have 48 terminals ports: 26 in the laboratory, 5 modems and the rest are connections to offices or other computers. Communication with terminals is via two 'intelligent' controllers. The first communicates between the 26Mb/sec SelBUS and the 1.5Mb/sec MultiPurpose Bus. The second drives the RS232 ports from the MultiPurpose Bus. We had problems with terminal communications.

The hardware has several deficiencies. Memory management protection modes allow general read access to the kernel and virtual (and real) memory is limited to 16Mb. Data is completely aligned (even double words need to be double-word aligned). The lack of a real stack makes porting Berkeley Pascal, the Amsterdam Compiler Kit and some other software a bigger job than one would like.

Since delivery, we've had a major processor upgrade which has improved performance and system reliability. Initially, the system was very unstable. Reliability was so poor that an undergraduate assignment had to be dropped. The combination of faulty processors and premature software inflicted a crisis on an overworked staff. Students lost confidence in the system. Gould did all that was reasonable to improve things, and recent experience suggests that their efforts yielded a much improved system.

The Operating System.

UTX/32 is a version of 4.2BSD adapted for dual processors. Performance is good (after 3 software updates). UTX/32 has a few System V features. System calls in this category are *fcntl*, *lockf*, *ulimit*, *uname* and *ustat*.

There are several annoyances remaining in the system. The system doesn't auto-reboot. Device drivers can crash the system by panicing (UTX/32 is the first place we've seen device drivers with calls to 'panic' - we hope this innovation will be seen for what it is and eliminated). System security is a problem (eg. *clists* can be examined by any process). The printer spoolers still cause problems.

Applications.

The C compiler performs reasonably. We've had very few portability problems with C programs. We'd like a faster compiler. Object code performance is acceptable, especially considering how poorly C maps onto the architecture.

The debugging environment isn't good. *Adb*(1) disagrees with *as*(1) about instruction mnemonics. A 'source code

debugger', *dbx*(1), works with C programs.

The Fortran-77 compiler is typical f77: compiles slowly, produces mediocre code, etc. It compiles slightly slower than VMS Fortran on a VAX-11/780.

LNS Pascal was supplied (without source code). This compiler translates Pascal to C, then uses the native C compiler to produce executable code. The compiler has its own set of features: Standard and UCSD Pascal are subsets of LNS Pascal (ie. it has *units* and *strings*), it has C features (like *continue*, *break* and *return* statements). It doesn't tell the C compiler about the source code from which it was derived, making use of debuggers difficult (especially for students). Each version of LNS Pascal has its own set of bugs (eg. files can't be used as arguments). At the moment, we have two versions in use.

LNS Pascal isn't suited to *make*. Separate compilation may produce attribute files which are modified only when their content changes. This is difficult to express in a *Makefile*. It may be done by telling *make* how to create attribute files but not showing their dependance on Pascal source.

Seventy students were given a project involving modification of a simple compiler. The compiler was written using LLAMA (and Pascal) and produces Gould assembler. Table 1 gives a measure of Pascal performance on the Gould.

command	time (secs) (cpu+sys)	source lines
llama comp.ll	17.5+10.5	276
pc -c tree.p	5.2+ 2.4	147
pc -c code.p	27.3+ 3.3	527
pc -c comp.p	61.6+ 4.9	1268
pc *.o	3.8+ 3.7	
make comp	116.4+32.2	950
pc llama.p	207.9+15.3	4000
pc -O llama.p	260.7+20.6	4000

Table 1.

The first section gives some idea of performance for the students work. The compiler is split into several modules: comp.ll, tree.p and code.p. The file, *comp.p*, is output from *llama comp.ll*. LLAMA compilation is given for comparison.

Gould supplied CEEGEN's C version of GKS. This is a level 2B implementation (includes interactive support). Our students are not expected to know C so this wasn't directly

usable. To use it from Fortran-77 or Pascal, interfacing routines were required. Rather than write and document over 200 routines, we only tackled the 90 we expected students to use. But GKS and f77 each have routines called *malloc* in their libraries. We didn't have source code so we used a special program to convert all occurrences of 'malloc' to 'Malloc' in the GKS libraries.

GKS documentation is a problem. Reference books have slight variations in the GKS interface. The CEEGEN documentation is a single hardcopy manual. We still have to produce acceptable online manuals.

We developed our own software for APC-III graphics (being unable to purchase a suitable package). The software emulates a mixture of VT52 (we sometimes talk to VMS) and tek4010. For simplicity, graphics is done via the MS-DOS GRAPHIC.SYS and is slow, but is (just) acceptable for teaching. We haven't tackled graphics input yet, so half the CEEGEN demonstration programs aren't working at our site.

Student use of GKS graphics was testing for the system. The combination of f77 compiles, large linking jobs and graphics programs made the system sluggish (but not as slow as IGL on VAX780/VMS). Executable programs using GKS are large; we could have used a 'shared library' mechanism to good advantage had it been available. This would save disc space and could improve performance.

Several other pieces of software had a workout. ACSnet, PROLOG and Ditroff were ported and all work hard. A Pascal hosted pseudo-parallel processing package was ported to the Gould and used for process oriented simulations. Gould were unable to supply MODULA-2 so we accepted an ADA (90% subset) in its place. In the near future we expect to be using ADA, APL, COBOL and database for teaching.

The standard system is augmented by user supplied software (called the D4 tape). Our experience has been that this software isn't very good. Initially, LISP and INGRES didn't run and still have problems.

Documentation

UTX/32 online documentation is confused. It is set up for a mixture of BSD and ATT manual macros - we spent some time sorting this out. Some documentation is supplied in post-nroff form. This is annoying and tedious.

Eqn and *troff* are missing. *Nroff* is a mess: half the device tables cause core dumps. Many UNIX sites have to sort this out for themselves; with Gould, things are no different. Fortunately, we have source code.

We had to produce a document describing instructions and the assembler, for the students. We need a better processor manual. Gould manuals are often longwinded and difficult to read. Detailed information about the hardware is hard to obtain.

Conclusions

The processors are good value for money: they are fast (especially doing floating point processing). The peripherals are OK but Gould should put more effort into the terminal interface.

Most of our problems have been with software. Local support lacks experience of Gould, UNIX Systems and the support process. Local expertise is usually below our own level. There isn't enough experience of 3rd party software yet and Gould seem unwilling to press the suppliers. In most respects, Gould matches our experience of other hardware suppliers.

Suitable software for undergraduates is still needed. Compiler performance needs work. A better Pascal is needed (new compilers are *coming*). Berkeley Pascal and the Amsterdam Compiler Kit resisted our initial attempts to port them (the latter runs and will cross compile code for almost anything).

Acknowledgements

Our small but dedicated technical staff, Michael Homsey, Kevin Dawson and Peter Dumbrell, kept the system running through the semester. They tolerated heavy workloads, unreasonable demands, heated words and performed far beyond the 'call of duty'. Credit for keeping the system running through the semester belongs to them.

The local agents for Gould worked vigorously to meet our requirements. The students showed considerable tolerance. They had more patience than one has a right to expect.

An Introduction to the UNIFY Data Base Program

Roy R. Rankin

COMPUTERLINK DESIGN
66 Concord Rd, Concord NSW

UNIFY* is a relational based data system which runs under the UNIX** operating system. This paper will look at some of the capabilities of UNIFY using an example to demonstrate them.

1. Introduction

In order to demonstrate some of the capabilities of the UNIFY data base program, an example will be presented. The present example comes from a company that buys and sells used excavator equipment, and needed to keep track of their customers and their stock. There are many companies with similar needs.

It was decided to split the information into two different but linked types of records. The first type of record contains information about the customer and was given the name *customer*. This record includes data such as company name, address, contact, phone, and telex. The second type of record contains information about the equipment to be sold, wanted to buy, and for future reference has been sold. This second record type was given the name *equipmnt* (record names cannot exceed 8 characters), and carries details about the equipment. These details include whether the equipment status is buy, sell, or user, details about the equipment, and a pointer to a customer record.

This paper will describe some of the steps required to setup such an application. It is hoped that this will give some insight into the power and flexibility of the suite of programs known as UNIFY.

2. Entering a Schema

The first step in setting up an application is to design and enter the schema. The schema tells UNIFY what records exist, how many records of each type are expected, and sets the characteristics of the fields thus defining the structure of the data base.

A record is a collection of fields which is stored in the data base as a single unit. Each record definition in the schema is given a unique name and the number of expected records. For those familiar with the C programming language, there is a strong resemblance between records and arrays of structures as well as between fields and variables.

A field has a name, type, length, and long name. The name is used to reference the field in the system design. A field can have 1 of 7

---

* trademark of UNIFY Corporation

** trademark of Bell Laboratory

types: NUMERIC, FLOAT, STRING, DATE, TIME, AMOUNT, and COMB. With the statement that NUMERIC is for integer, and AMOUNT is for dollars and cents, all the types are clear except for COMB. COMB, or combined type is similar to a union in C. The length is the number of characters or digits for the field. The long name is a name which is used by SQL to identify fields during queries, and by the default screen form generator as labels.

One field in each record definition is designated as a primary key. The primary key is hashed. This allows search by primary key to be very fast, but also means then when entering data the primary key must be unique. The primary key field is indicated on a schema listings by a preceding asterisk. The following is a simplified list of the schema used for the excavator company which was entered into UNIFY using its interactive schema generator.

Schema Listing

RECORD/FIELD		REF	TYPE	LEN	LONG NAME
customer	1000				customer
*id			COMB		cus_idkey
idn			STRING	6	cus_idn
name			STRING	30	cus_name
add1			STRING	30	cus_add1
add2			STRING	30	cus_add2
add3			STRING	25	cus_add3
pc			NUMERIC	4	cus_pc
cont			STRING	35	cus_cont
phone			STRING	15	cus_phone
telex			STRING	12	cus_telex
equipmnt	10000				equipmnt
*eid			NUMERIC	7	eqp_eid
ecus		id	COMB		eqp_ecus
type			STRING	15	eqp_type
make			STRING	15	eqp_make
model			STRING	15	eqp_model
year			STRING	15	eqp_year
location			STRING	20	eqp_location
cost			AMOUNT	7	eqp_cost
sell			AMOUNT	7	eqp_sell
disp			STRING	4	eqp_disp
edate			DATE		eqp_edate
node	1				node
*nkey			STRING	1	n_key
equipseq			NUMERIC	7	n_seq

3. Enter Screen

Once the schema has been entered, an *enter screen* can be created. The enter screen is used for interactive data entry and enquiry. There are several ways to accomplish this task. The easiest is to use the default screen form generator. By this method the schema record name is given to the default screen generator, and the screen is generated. The default screen generator uses the long name field entered in the schema for labels.

A second method of generating an enter screen is to use a full screen editor known as PAINT. PAINT has a fairly simple, but powerful, set of commands which allows you to move around the screen and place captions and fields where desired, thus allowing you to layout the screen as desired.

The enter screen, no matter how created, is used by a program called ENTER to allow data to be added, modified, deleted, or searched. In some cases, we would like ENTER to perform functions it does not normally perform. If this is the case, ENTER can easily be customised.

4. Customising ENTER

For the present case, we would like to customise the entry program for two fields. The first field is the *eid* field which is the equipment primary key for the equipmnt record and thus must be unique. On data entry, we would like ENTER to generate this field automatically to give a unique equipment id number. The form which was selected for this id number is that it is a seven digit number. The two most significant digits are the year, ie 85, 86, etc. The next three digits are the day of the year, and the least two significant numbers are determined to make the id number unique. Note that if more than 100 items are entered in a single day, the id number will show the wrong day. This behaviour was deemed to be acceptable. In order to implement the automatic generation of the equipment id number(eid), a short C function was written and a record called *node* was created. Node has a field equipseq which is the last eid field generated. Although node is not required, it improves the efficiency when a number of records are added in the same day. The function operates in the following way. If a record is not being added or the field has already been entered, the function exits to default processing. Otherwise, it calls time() and ctime() and generates a key, and then reads the equipseq field of record node. If equipseq is greater than the key, it uses equipseq as the key. A search is then done with the key, and if a record is found, the key is incremented until no record is found. Equipseq is then updated with the new key, and the new key is sent to be processed as if it had been typed in.

The second field customised was the date field. A second C function is written to put in the current date if it had not already been entered by the operator. The logic of this subroutine is as follows. If a record is not being added or the field has already been entered, return. Otherwise, build the current date in the proper format using time() and

ctime() and then send the key to be processed as if it had been typed in.

In order to get ENTER to use the custom functions which have been written, several structures were created which indicate the record and the field on which we wish to execute our custom functions. The details of these simple structures are beyond the scope of this paper. Finally we compile the functions and the structures using a special C compiler front end called *ucc* and then create an archive of the object modules. Then we load our functions into ENTER using a shell script called *enter.ld*. Once this is complete we can used our customised ENTER program by just calling the appropriate enter screens.

5. SQL query/DML Language

With the ENTER program and our enter screens, we can already interactively perform all the data base functions of adding, modifying, deleting, or searching for data. There are some cases where we would like to perform the above functions on a batch level such as for generating reports. This is where the SQL is useful. SQL, which stands for Structured Query Language, was designed by IBM to be a powerful, flexible, and easy to use query language. SQL also uses a Data Manipulation Language, DML, to allow batch addition, modification of deletion of data base records. SQL/DML can either be run from the UNIFY program or outside it.

For the present case we are using a fairly simple SQL script to generate reports. The script is as follows:

```
lines 0
select *
from equipmnt, customer
where equipmnt.cus_idn = customer.cus_idn /
```

In SQL the "/" terminates the script starting execution. Normally SQL prints out a heading at the top of every screen full of output. "lines 0" tells SQL not to put out any headings. The asterisk means all fields in the given records, and thus the script says to print all fields from record type equipment and customer. The *where* clause is to print only the customer record associated with each equipmnt record. This script produces a list on the standard output of all the fields of equipmnt and customer separated by a '|' with each record on a separate line. This output could be sent to sed, awk, or any formatter program desired.

6. Report Processor

UNIFY also contains a report processor called RPT which can be used to format the output from SQL. RPT gives the programmer a high degree of flexibility over the placement and format of fields, headings, footings, titles, and pagination. To demonstrate some of the capabilities of RPT, a script for generating labels from the customer records will be

examined. This script presents only some of the capabilities of RPT. But first, the SQL script for this report is as follows:

```
lines 0
select * from customer/
```

The RPT script is as follows:

```
input
        customer.cus_idn,
        customer.cus_name,
        customer.cus_add1,
        customer.cus_add2,
        customer.cus_add3,
        customer.cus_pc,
        customer.cus_cont,
        cus_phone [string 15],
        customer.cus_telex
sort cus_pc, cus_name
detail
    if cus_add1 > 30[ ] then
    begin
        need 8
        print cus_name
        print cus_add1
        print cus_add2
        print cus_add3 no newline
        if cus_pc > 0
        then
                print cus_pc using '####'
        else
                print ' '
        skip
        if cus_cont > 35[ ] then
                print 'ATTN:' in col 5, cus_cont
        else
                skip
        skip 2
    end
end
```

This script has three main sections, input, sort, and detail. The input section tells RPT what variables are being input and the variable type. When the input is specified as customer.xxxx, RPT looks into the data base and finds the format for the field xxxx in record type customer. Alternatively the format can be specified as was done for cus_phone.

The second section of the script is the sort command. This sorts the data first by postal code, cus_pc, and then by name, cus_name, causing the output to be first in postal code order and then alphabetically by name.

The main section of the script is the detail command.
The detail command is executed for every line in the input, and is where the outputting is done. This part of the script checks if the cus_add1 field is other than 30 blanks, and if so prints the entry. The "if cus_pc > 0 then" is to prevent 0 being printed if no postal code was entered. The "using '####'" provides formating information.
In this case, cus_pc is printed as a number right justified in a field four characters wide. Two other constructs in the script are "in col 5" which starts the printing of the field in column 5, and the skip command which skips the number of lines specified with one as the default.
The following is a single entry from the output of the script.

```
  ComputerLink Design
  66 Concord Rd
  Concord, NSW
                          2137

      ATTN: Roy R. Rankin
```

7. Conclusion

This paper has gone through some of the steps required to setup an application under the UNIFY data base program in order to provide an introduction of some of its capabilities. Many capabilities have not been shown by these examples. In particular, the menu driven nature of the user interface has not been shown as well as the security features provided by UNIFY. Nevertheless, it is hoped that this paper increases understanding of a very significant piece of software which runs under the UNIX operating system.

fsh - A Functional UNIX Command Interpreter

Chris McDonald
Department of Computer Science
The University of Western Australia

1. INTRODUCTION

Both the UNIX operating system and functional programming systems (FPs) provide the ability to compose existing programs to form new ones. This ability was one of the primary goals of the designers of each system [Ritchie][Backus]. The widely accepted UNIX command interpreters, or shells - the Bourne shell (*sh*), C-shell (*csh*) and Korn shell (*ksh*) - enable the user to combine existing programs into new commands using the two mechanisms of pipes and control of flow within each shell. Both of these have close correspondence with the facilities provided in functional programming of composition and conditional evaluation. The recognition of this has lead to a small number of functional shells in which UNIX programs are viewed as functions and combining two UNIX programs is functional composition [Matthews][Shultis].

Unfortunately there is unnecessary complexity in existing command interpreters. Shells support a number of different command mechanisms - variables, aliases, shell scripts (command files) and, a more recent addition, functions. Each has a different syntax for definition and removal from the operating environment, and access to any arguments.

A powerful command interpreter may be constructed which considers every object as an expression requiring evaluation. Command sequences may be consistently defined and applied using the functional approach - function definition and application. Such an interpreter is termed a *functional* interpreter. Input and output redirection, and error and signal handling, are also tasks to be handled by an interpreter, and it is possible to handle these functionally - by defining expressions to perform these tasks. All these features lead to a syntactically and semantically consistent command processing environment.

One approach to creating a functional command interpreter has been to add function definitions to an existing shell, as has been done in *ksh*. The *ksh* additions do not offer true functional benefits. The added facilities are already available through the use of shell scripts. Higher order functions are not fully supported. Control facilities are imperative, as in precursor shells, rather than applicative as should be the case for true functions. Unfortunately, none of these facilities may be made to follow the functional paradigm without significant changes to the implementation.

The command syntax of existing functional command interpreters can often deter users. To date, functional command interpreters have used the syntax of Backus' FP and the advantages of expressiveness and command composition in an interpreter have not been appreciated. [Stabile] discusses the use of Backus' FP as a command language and comments particularly upon its 'notational austerity'. Although using a more formal system, the user is presented with a system that appears at a much lower level of complexity than the one he is

trying to control.

This paper demonstrates the advantages that a functional command interpreter offers over a non-functional one - consistency in command definition and evaluation, control of flow and support for higher order functions. A functional command interpreter is introduced, by way of examples, which overcomes the syntactic deterents of precursor functional interpreters.

2. A FUNCTIONAL SHELL

In this section we introduce a *f*unctional *sh*ell, *fsh*. *fsh* has been designed to support, and hence enforce, a functional approach to command procedures. Programs and functions are treated consistently as objects requiring evaluation. The implementation runs under the UNIX operating system.

fsh has a syntax similar to the abstract syntax used by [Henderson] in defining the semantics of his Lispkit. The syntax is considered by the author to be easier to learn than that existing programmable command interpreters. Syntax changes have been made to Henderson's syntax which provide filename expansion, input/output redirection and pipes.

fsh provides the types integer, character, string, boolean, functions and nested lists of these types. This set permits easier command programming than available in the existing UNIX shells, which perform all operations and argument passing using strings. Rather than supplying an enormity of operators for these types and a correspondingly difficult syntax, operators become overloaded - their semantic actions dependent on the types of their arguments. The basic operators on these types are arithmetic, relational and list and string construction and decomposition.

The syntax and operators of *fsh* will be informally introduced by example. A complete syntax is given in Appendix I.

The following command script uses all the above types. Function *nextfile* receives string and integer arguments to determine the next available version of an indicated filename. Here : is the string concatenation operator, which appends the suffix of a file's version number to the filename. Function *itoa* converts the version number (an integer) to a string and *exists* is a boolean function indicating the existence of a file (described later).

```
def nextfile(name version) =
        let filenm = name : '.' : itoa(version)
        in
        if exists(filenm) then nextfile(name version+1)
        else filenm;
```

A function parameter may be referred to by its name. This provides a distinct advantage over positional parameters (the $1, $2, etc) and the use of the *shift* operator in *sh*, *csh* and *ksh*. Arguments to command scripts may be passed as individual atoms or lists, providing a facility for scripts which accept a variable number of arguments. When arguments are passed to UNIX programs, they lose their type information and list "shape" by being flattened to string

representations of their values.

Many UNIX programs can recursively perform their function on a directory, utilizing the tree structure of the UNIX file system, by providing a "do recursively" flag (often "-r"). However, this ability is not consistent across all commands, for example one may recursively delete files ("rm -r") but not recursively protect them (there is no "chmod -r", [Bell]). Rather than add recursive application to every program requiring it (unnecessarily increasing program sizes), a generalized command procedure would accept a program, arguments and a subtree of the file system over which to be applied. Function *treeapply* presents such a command script written in *fsh*.

```
def treeapply(com args fs) =
    if null fs then nil
    else let f = hd fs in
         if isfile(f) then com(args f).treeapply(com args tl fs)
         else treeapply(com args glob(f:"/*")).treeapply(com args tl fs);
```

The intrinsic function *glob* expands its string argument and returns a list of all matching file names (as strings) or **nil** if there is no match. This is equivalent to standard UNIX filename expansion which returns matching filenames as arguments to a program. To make the syntax of filename expansion simpler, *fsh* also treats all *characters* within curly brackets as a candidate for expansion. These characters are not an expression needing evaluation (they are constant), but are passed directly to the *glob* function. The command

```
treeapply(chmod 600 {*});
```

will now recursively protect all files in a directory, and

```
treeapply(lpr "-Plp" {*});
```

will recursively print them.

Functional languages with no side-effects provide inherent parallelism. Arguments to functions and subexpressions provide opportunity for a functional interpreter to evaluate expressions in parallel. Unfortunately the side-effects introduced by programs destroy this inherent parallelism. Programs modify the file system, hence a function that accesses the file system may return different results if executed twice. It may be necessary to execute a program twice to achieve a desired result. The result is that what appears to the interpreter as a common subexpression may need to be evaluated twice.

Under the UNIX operating system, all devices are an integral part of the file system, and so any input/output performed by a program alters the file system. Programs consistently return an indication of their success using an integer exit-status, with success usually indicated by the value zero. Hence, the result of executing a program is a pair (a *cons* in Lisp), the first indicating the success of executing the program, and the second the changes made to the file system. It is, of course, impractical for every command to return a new file system and so the modified file system is represented on a physical

device.

Command processing environments use this exit-status as a boolean value. Using this idea, many simple and useful *fsh* functions may be written using the UNIX program *test*. Some of these have been seen already.

```
def isfile(name)   = test("-f" name);
def isdir(name)    = test("-d" name);
def exists(name)   = test("-s" name);
def readable(name) = test("-r" name);
def writable(name) = test("-w" name);
```

Using these functions and the following higher order function *map*, a function may be written to remove all core dump files in a file system. The function *rm* is a standard UNIX program to remove files.

```
def map(f l)    = if null l then nil
                  else f(hd l) . map(f tl l);
def rmcore(fs) = if null fs then nil
                  else let f = hd fs in
                       if f="core" and isfile(f) then [rm(f)]
                       else map(rmcore glob(f:"/*")).rmcore(tl fs);
```

The result returned by this function will be a nested list of integers representing the exit-status of each *rm* program.

Function *backup* is an example of a command script to recursively backup files and directories, maintaining successive versions of each. The keyword **lambda** defines a λ expression local to function *backup* [Henderson]. *cp* is a standard UNIX program to copy files.

```
def backup(from to) =
        if isfile(from) then
             if isdir(to) then cp(from nextfile(to:'/':from 1)
             else cp(from nextfile(to))
        else if isdir(from) then
             if isdir(to) then map(lambda(x) backup(x to) glob(from:"/*"))
             else error("cannot cp directory to file" to)
        else error("cannot cp" from);
```

The extended set of types and operators demands a syntax that is more difficult to use than that of existing UNIX shells, but permits and enforces a uniform treatment of UNIX programs and functions. Higher order functions involving both command scripts and programs may be easily constructed. The development of command scripts is identical to programming in a truly functional language. Using higher level functions, such as *map* given above, powerful one line commands may be written. For example, to change all filenames having the suffix ".1986" to filenames ending in ".86", the command

```
map(lambda(x) mv(x:".1986" x:".86") map(stripsuffix(".1986") {*.1986}));
```

may be used. Alternatively, a general command to change filename suffixes may

be written as

```
def change(old new) =
    map(lambda(x) mv(x:old x:new) map(stripsuffix(old) glob("*":old)));
```

3. EXECUTION

fsh commences by parsing any function definitions found in the file *.fshrc* in the user's home directory and any files indicated on the command line. The function definitions found here are those the user desires to establish a command environment of commonly used functions. Typically included is the "lookup path" for the location of programs and function definitions :

```
def path = [ "." "/bin" "/usr/bin" "/u/lib/fsh" ];
```

which defines a function returning a list of directory names.

fsh then enters a "read-parse-evaluate-print" loop. Input expressions are parsed using a parser generated with *yacc*. A parse tree is created with operators or subexpressions at the nodes and atoms at the leaves. This tree is traversed by applying each operator to its arguments. If a function (or UNIX program) is to be evaluated, its definition is first sought in the global environment, then in a file of the same name in a directory in the search path (each of the directory names returned when the *path* function is evaluated). The definition of a UNIX program is equivalent to the binary image of the program on the disk, and "evaluation" of this program, akin to its execution.

As an example, the user may define the prompt as a function accepting an integer parameter.

```
def prompt(n) = "\n" : itoa(n) : "-> ";
```

When *fsh* attempts to print the prompt, the *prompt* function is evaluated in the global environment. If it is not found, the *path* function is evaluated to obtain the directory list to locate a file containing the *prompt* function. In evaluating the prompt, the function *itoa* (converting an integer to a string) is required, and again the *path* function is used to locate the *itoa* function. A program or function's definition is only loaded into the global environment if it is found in a file of the same name (as is the UNIX convention for executable files). Using a stack of current input files, this procedure may continue until some operating system limit is reached.

path and *prompt* are examples of functions that the command interpreter evaluates during normal evaluation of the user's functions or programs. Others include *cdpath* (for changing the current working directory) and *debug* (the level of function debugging reported). To indicate whether or not timing of commands is required a function, *time*, may be defined which returns true or false.

4. BUILT-IN COMMANDS AS FUNCTIONS

Historically, UNIX utilities such as *echo* and *test* existed as programs which were spawned by the shell and their result (exit-status) monitored. For reasons of efficiency these programs are now built into shells, increasing their size and complexity. These commands are often called *built-in* commands.

fsh only incorporates those commands as built-in functions which cannot meaningfully exist as seperate programs. A good example is the command to change the current working directory. The current directory is an attribute of each process and cannot be directly passed back to the calling process. Spawning a new process to change directory would leave the interpreter in the original directory. Hence, changing directories must be performed by the interpreter itself. For this reason there is no change directory program under UNIX and *fsh* has a built-in function to change directories. In contrast, the program *test* may exist as a seperate program and is not built into *fsh*.

For example, the change directory command is defined as :

```
def cd(directory) = builtin("cd" directory);
```

and may be evaluated with

```
cd("/user/chris/fsh");
```

All builtin functions return a result. The *cd* function returns either **T** or **F** indicating whether or not it was possible to change directories. The function *glob* (described earlier) is also a built-in function which returns a possibly empty list of filenames.

Built-in functions also enable the user to access many values associated with the user, his terminal and default parameters for commands. Under UNIX this information is contained in an *environment*, a collection os strings of the form *name=value*. The *name* describes the attribute contained in *value*. A representative environment is :

```
HOME=/user/chris
USER=chris
TERM=HE
PATH=/bin:/usr/ucb:/usr/bin:/usr/local:/usr/games
```

This information is inherited from a calling process and is accessed with the UNIX call *getenv*. *fsh* provides access to the environment strings with the builtin function *getenv*. Frequently used functions are :

```
def home         = builtin("getenv" "HOME");      # home directory
def user         = builtin("getenv" "USER");      # user's name
def term         = builtin("getenv" "TERM");      # terminal type
def path         = builtin("getenv" "PATH");      # command lookup path
def getenv(name) = builtin("getenv" name);
```

The environment values may be accessed by calling each function. Each function returns the string *value* if *name* is defined and **nil** otherwise.

The use of the *getenv* function is in contrast to the use of *shell variables* or *shell parameters* in *sh* and *csh*. In these shells the shell variables are referred to by preceding their name with a '$', for example $home and $user. *fsh* removes the need for shell variables by providing access to the environment values with a builtin function. This results in a simpler, more consistent syntax.

Built-in functions are not implemented as intrinsic functions in the interpreter. For example the function *cd* is not builtin, but may call the function *builtin* with the string "*cd*" as the first argument. This enables the user to redefine the action of any built-in function for every expression using local definitions.

5. INPUT/OUTPUT REDIRECTION

Input and output (i/o) necessarily introduce side effects in any functional programming environment. Execution of UNIX programs provides each program with a standard input channel, either from a pipe, file or (by default) the keyboard, and a standard output channel, either to a pipe, file or the terminal. I/o is part of the operating environment of a UNIX program. This idea may be extended by considering i/o as part of the environment of any expression being evaluated.

I/o redirection is treated as a binary operator of expressions, the first, the expression whose input or output is being redirected and the second, an expression returning a filename. The result returned by the i/o redirection operator is a boolean value indicating the success of all redirection for an expression (see also Error and Signal Processing).

I/o redirection is evaluated in the environment of the current expression, permitting conditional redirection. The *nroffit* command will redirect its output to successive versions if the indicated output is a directory.

```
def nroffit(input output) =
    nroff(input) -> if isdir(output) then output:'/':nextfile(output,1)
                    else output;
```

If redirection is explicitly indicated, it is added to the current i/o environment. By maintaining i/o as part of an expression's environment, i/o may be inherited by any expression from its outer environment as are function definitions and parameters. I/o redirection for a particular expression may be viewed similarly to a local function definition for any expression. By default, i/o is taken from the standard input and output of the interpreter.

6. ERROR AND SIGNAL PROCESSING WITH FUNCTIONS

During the parsing of an expression, all syntax errors are reported by way of type and location. If a syntactically incorrect function definition is being read from a file on which the user may edit, a copy of the syntax errors are placed in the file and the user's *editor* function, typically

```
def editor = vi;
```

is evaluated using *path*, and if possible, applied to the file containing the function definition. If the file is modified, it is re-parsed and evaluation continues.

Run-time errors are of two types - those from which no meaningful recovery is possible, and those from which the user may recover by supplying an alternative action or result. The first type, such as the interpreter's inability to spawn a process, or attempting to apply an expression that that is not a function are trapped and an error message printed.

If an error of the second type occurs, such as division by zero or an invalid input/output redirection, the user defined function *error* is evaluated in the *current* environment enabling the user to recover from the error. As an example, the following functions divide one list through by another,

```
def divlist(l n) = if null l then nil
                   else (if hd n=0 then maxint()
                         else hd l/ hd n) . divlist(tl l tl n);

def divlist(l n) = if null l then nil
                   else (hd l/ hd n . divlist(tl l tl n))

                   where error(type val) = if type=2 then maxint()
                                           else error(type val);
```

The first function provides a thorough check against a division by zero, at the expense of a test that may never evaluate to true. The second uses the *error* to trap the division by zero as an exception, saving the test. Note that the local definition of *error* is not recursive (**where** is used rather than **whererec**) and if an error arises that is not division by zero (here represented by 2), the error can propagate to the outer (global) environment. By having a global "catchall" *error* function, the user can both trap all errors and provide their own error messages. For example, the *error* function below reports two explicit errors.

```
def error(errno val) =
    let print(str) = builtin("error" str val)
    in
    if errno=2 then print("Attempted division by 0")
    if errno=3 then print("Unable to redirect output")
    else            print("Error");
```

A *signal* or interrupt may be received at any time by a command interpreter and so cannot predictably return a meaningful result to the current expression

being evaluated. However its occurrence must be noted and an action performed upon receipt of a signal. All signals received by *fsh* can either be trapped or ignored through the user defined function *trap*, shown below. The action taken upon receipt of a signal is to evaluate the *trap* function in the current environment and the result returned to the "top-level" environment (where it is printed). The special value **nil** is used to indicate that a signal is to be ignored. As in the case of error handling, the use of non-recursive local definitions of *trap* enables signals to be propagated from each expression level.

```
def trap(signal) =
    let print(str) = builtin("trap" signal str)
    in

    if signal=2 then print("interrupt")
    else if signal=3 then nil
    else                 print("trapped " : itoa(signal));
```

7. IMPLEMENTATION

fsh is written in the C programming language and runs under the 4.2BSD UNIX operating system on a Digital VAX computer. Other than for memory requirements, *fsh* could be transported to a smaller UNIX environment as no 4.2BSD specific features have been used.

The interpreter performs all calculations in a static heap of 240k and uses a dynamically allocated stack typically requiring a further 30k. A stack is used to represent the input/output environment, limited by the number of files a single process may have open. Strings are stored uniqely in a hash table.

Garbage collection is performed using Knuth's modified version of the Schorr-Waite algorithm [Schorr][Knuth], all accessible data in the global environment, run-time stacks and the input/output stacks is marked and all unmarked storage collected. Garbage collection is performed when the heap is exhausted or if there is sufficiently high memory usage before the *prompt* function is evaluated. This frequency does not cause a noticeable degradation in speed under normal loads. Real time garbage collection has been examined [Dijkstra][Wadler]. Most real-time algorithms require two equally sized data spaces. Considering the large amount of memory already required by the interpreter, this memory overhead would be excessive and has not been included in *fsh*.

fsh, when used as a command programming language requires only one process to evaluate command scripts (functions). Functional composition may be evaluated within one process rather than one process for each shellscript as required when piping between processes in conventional shells. UNIX programs and pipes between UNIX programs still result in processes being spawned, their exits being monitored.

Although not used here, *lazy evaluation* [Henderson] could be used in a command interpreter. This would permit the composition of two *fsh* functions to behave similarly to two programs connected with a pipe. Pipes under UNIX permit the passing of possibly infinite character streams between two processes. The

process producing the character stream executes until its normal termination or until the receiving process terminates. If the latter occurs, the pipe connecting the processes is termed "broken". The writing process learns that the reading process has terminated by attempting to write on this "broken pipe". This often results in the premature termination of the writing process. With lazy evaluation possibly infinite data structures may be passed between two functions. The reading function requests input from the writing function. This *demand driven* concept of a pipe would result in each function only performing as much work as required and would guarantee a cleaner termination than the "broken pipes" under existing interpreters.

8. CONCLUSION

The UNIX file system and process structure encourage the use of functions to control an interactive environment. This paper has described the functionality available in a command interpreter under the UNIX operating system. A functional shell, *fsh*, has been introduced by way of example which encourages a functional consideration of commands and provides a functional treatment of command composition, input/output redirection and signal and error handling. *fsh* provides the user with control over his interactive environment with a simple and consistent command syntax.

References

[Backus] Backus, J.
Can Functional Programming be Liberated From the von-Neumann Style?
A Functional Style and its Algebra of Programs.
CACM, Vol 21, No 8.
August 1978, pp613-641.

[Baden] Baden, S B.
Berkeley FP - Experiences with a Functional Programming Language
Proc IEEE COMPCON, 1983.
pp274-277.

[Bell] Bell Telephone Laboratories.
The UNIX Programmer's Manual
Murray Hill, New Jersey.
1983.

[Dijkstra] Dijkstra EW, Lamport L, Martin AJ,
Scholten CS and Steffens EFM.
On the fly garbage collection
CACM Vol 21, No 11
Nov 1978, pp966-971

[Henderson] Henderson, P.
Functional Programming : Application and Implementation
Prentice-Hall International Computer Science Series
ed : C.A.R Hoare
London, 1980.

[Kamath] Kamath, Y.
FP-Shell, A Functional Programming Environment for UNIX
M.Sc Thesis, University of South Carolina

[Knuth] Knuth, D.E.
The Art of Computer Programming Vol I, Fundamental Algorithms
Addison-Wesley, Reading, Mass.

[Matthews] Matthews, M.
Functional Aspects of UNIX
IEEE SOUTHEASTCON '83 PROCEEDINGS
pp91-94.

[Raoult] Raoult, JC.
Properties of a Notation for Combining Functions
JACM Vol 30, No 3.
July 1983, pp595-611.

[Ritchie] Ritchie, D.
The UNIX Time-Sharing System
Bell Systems Technical Journal
Vol 57, No 6, 1978.

[Schorr] Schorr, W.
An Efficient Machine-Independent Procedure for Garbage Collection on Various List Structures
CACM Vol 10, No 8.
1967, pp 501-506.

[Shultis] Shultis, J.
A Functional Shell
SIGPLAN Notices, Vol 18 No 6, 1983.
also : Proc. Symp. on Prog. Lang. Issues in Software Systems, pp202-211.

[Stabile] Stabile, L.A.
FP and its uses as a Command Language
Proc IEEE COMPCON, 1980.
pp301-306.

[Wadler] Wadler, PL.
Analysis of an Aalgorithm for Real Time Garbage Collection
CACM Vol 19, No 9.
pp 491-500.

APPENDIX I : THE SYNTAX OF *fsh*.

```
command     = (defn | expr) ';' .
defn        = def identifier '(' formal_args ')' '=' expr.
params      = {identifier}.

expr        = identifier
            | constant
            | '(' expr ')'
            | lambda '(' formal_args ')' expr          ! lambda expressions
            | \ '(' formal_args ')' expr
            | expr '(' {expr} ')'
            | unary_op expr
            | expr binary_op expr
            | expr input-redirect
            | expr output-redirect
            | expr input-redirect output-redirect
            | if expr then expr else expr
            | (let | letrec) bindings in expr
            | expr (where | whererec) bindings

constant    = integer | character | string
            | T | F | nil .
list        = '[' {constant} | {list} ']'
            | '{' string '}' .                         ! filename expansion

unary_op    = hd | tl                                  ! list access
            | atom | null
            | int | char | string                      ! type checking
            | not .

binary_op   = '+' | '-' | '*' | '/' | '%'              ! arithmetic
            | '.' | '..' | ':'                         ! list construction
            | '<' | '<=' | '=' | '<>' | '>=' | '>'     ! relational
            | and | or                                 ! logical
            | '==' .                                   ! pipe between processes

input-redirect    = '<-' expr .                        ! input from
output-redirect   = '->' expr                          ! output to
                  | '->>' expr .                       ! output appended to

formalargs = identifier | formalargs
bindings    = binding | bindings ',' binding .
binding     = identifier
            | identifier '(' params ')' '=' expr .
```

The Perth AUUG Meeting: A Visual Perspective.

or

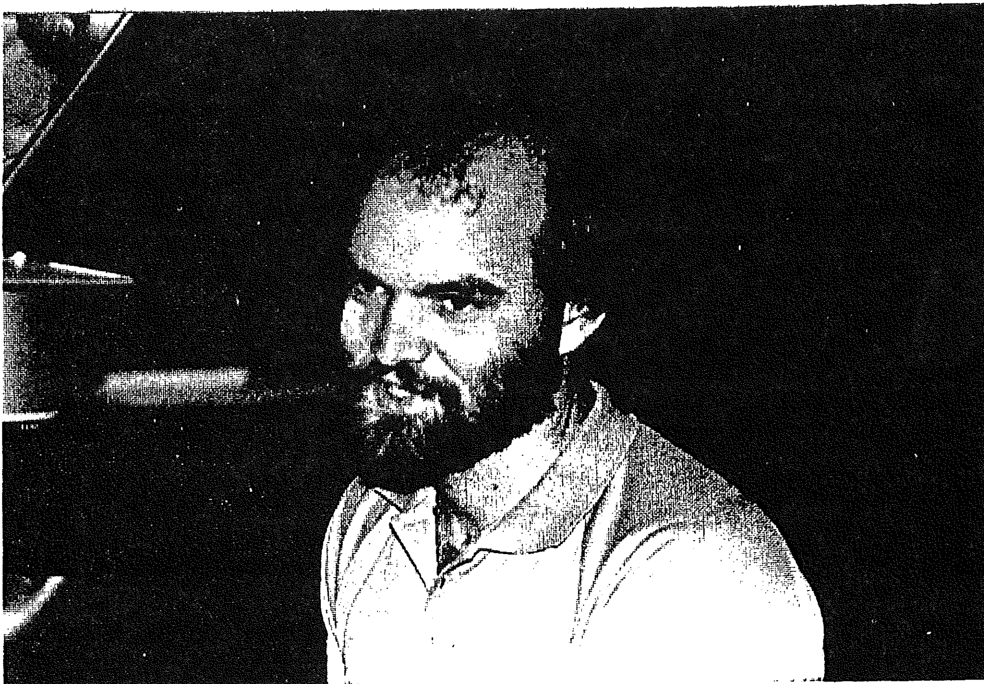
How I Pestered Everybody With a Camera,
and
How They Came Out.

Well, there I was in Perth, going to this meeting, and faced with the usual prospect of being dragooned into work merely because I was unfortunate enough to be the AUUG Secretary. "Ha," I said to my roommate Chris Maltby, "Ha," I said, "I'll fool them, I'll bother all of the speakers by taking photos at crucial moments of their talks."

Chris (the Treasurer) replied "Just don't do it to me, or else I'll ...". I won't tell you what he would do. Suffice it to say that the picture of Chris is not shown here.

However, lots of others are.

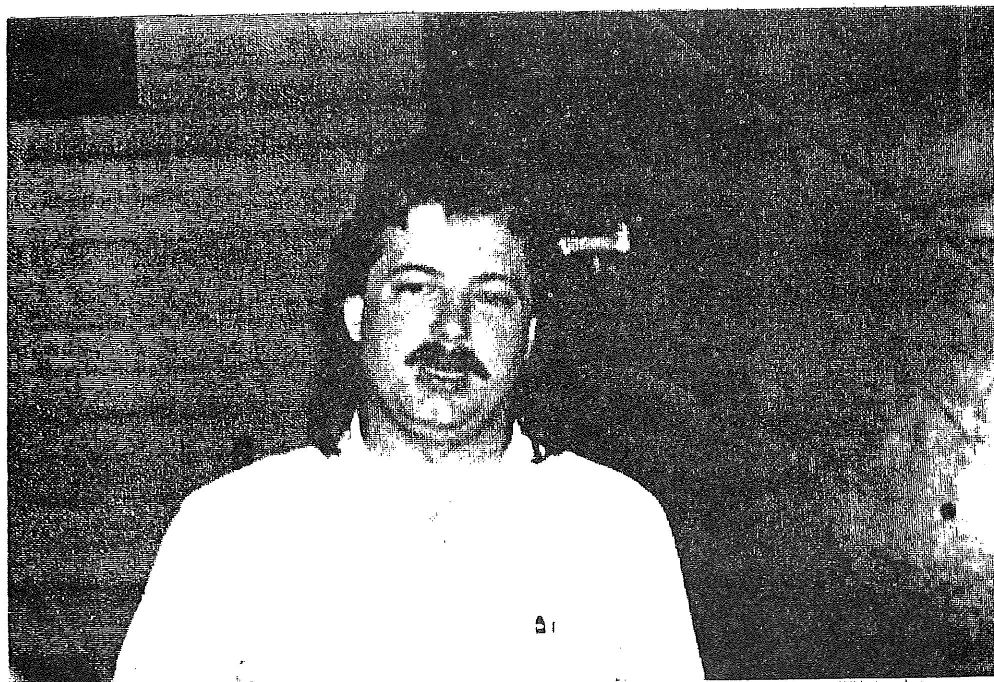
Taking pride of place, because organising a meeting is even harder than being Secretary of AUUG, so I really sympathise with him, and because he did a whacking good job, is **Glenn Huxtable**. He also gave an interesting talk about setting fire to prominent public buildings, or something like that.



This was a meeting chock-a-block with overseas speakers. The first of these was **Kirk McKusick**, who was really from the University of California at Berkeley, although there was some confusion over this in the program.

Since this was one of the first talks, the photo was not very good, being taken by my inexperienced self. I made up for this later, getting a reasonable shot of Kirk while he thought I was getting a shot of his friend Eric. I also got a *very* good one of Kirk at a skinny dip on Rottnest Island, the negative of which is for sale to the highest bidder.

AUUG got a bonus when we invited Kirk. He had enough miles on a free travel program to get a free ticket for someone else, so when we invited him, we paid for his ticket, and some airline paid for **Eric Allman**.

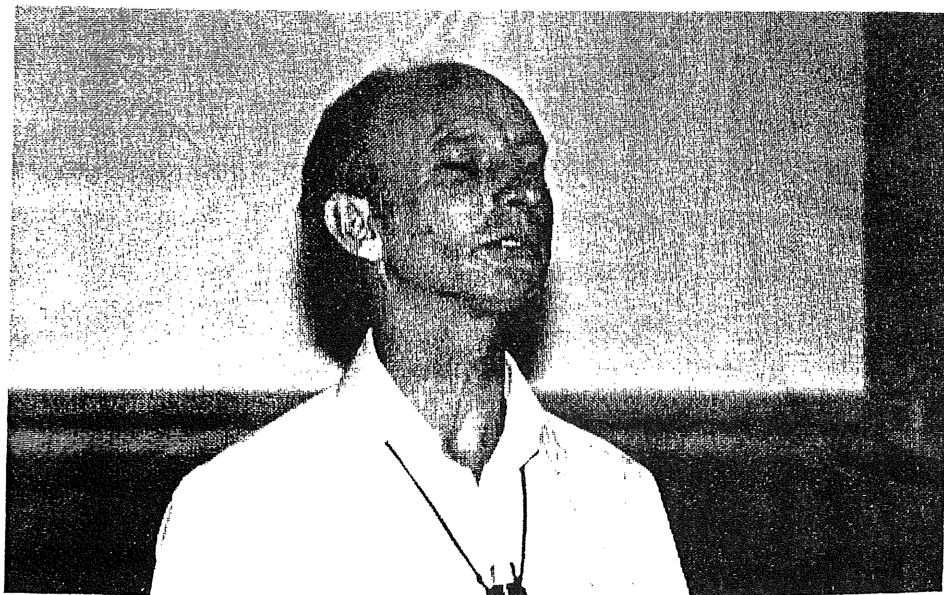


Eric, seen here during his talk, works for Britton-Lee, makers of database back-ends. He had a great T-shirt, saying "Get your bits out of our Back End". I want one.

We got in touch with Kirk and Eric through **Robert Elz**, that villain of the melodrama from the 1984 meeting. Robert is *rabidly* antiphotography, so Kirk and Eric helped us get the next photo by snatching the program away from his face just at the right moment.

Captions are desired for this photo, just to make life even harder for Robert.

The other (other) keynote speaker was **Roger Hicks**, representing the NZUSUGI (New Zealand UNIX Systems User's Group Inc.?). He had good t-shirts for *sale*, which just goes to show the terrible commercialisation of the NZ group. His talk was a real eye opener for many of the listeners, and we have a lot to learn from this young and progressive group. Not a flattering photo Roger, I'm sorry.



The next nearest to a keynote speaker was **Andrew Hume** (or Huge) who had come all the way from the US to be with his native Australians and their Comet. His talk was quite interesting, being about folding regular polyhedra, and showed that with enough backing from Bell Labs anything can make a good presentation.

Here, I will present the locals of the meeting. These three were much in evidence around the meeting, with the first two giving talks, and all three helping with organisation. There were a host of others who helped, but these were the only ones with decent photos. Indeed, the last was the only photo given with cooperation.

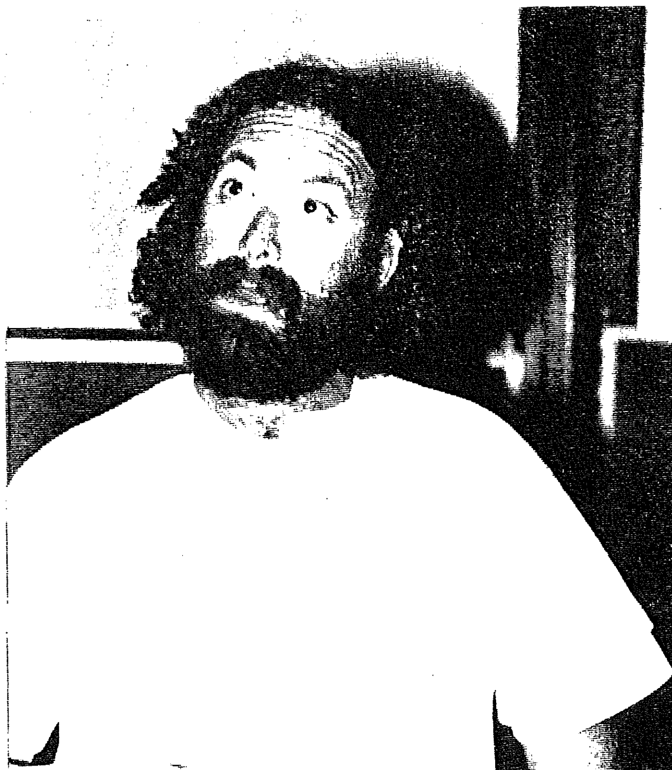
With great embarassment I admit that I am writing this much later than intended, and the photos were shuffled by well meaning lookers, so I have lost the names of all but the first, **Chris McDonald**, and I may have got that one wrong too, and I can't get on to Glenn to check, and I'm sorry, god I'm sorry, I'm just so ...

LATE NEWS! STOP PRESS! (I told you this was a last minute effort.) Glenn has responded to my descriptions and I now have more names. I like to think the next one is **Mark Ellison**, if he isn't he should be, and Glenn is very sorry...

This next one is of **Anita Graham**, who is actually with the West Australian Regional Computer Centre, but who was around and helpful all of the time.

Now come the die hards. These are the people that you see at all of the meetings, who will try to come even if it kills them, and who quite often talk. We need more of these. And just to further rankle **Robert Elz**, I'll put him first...

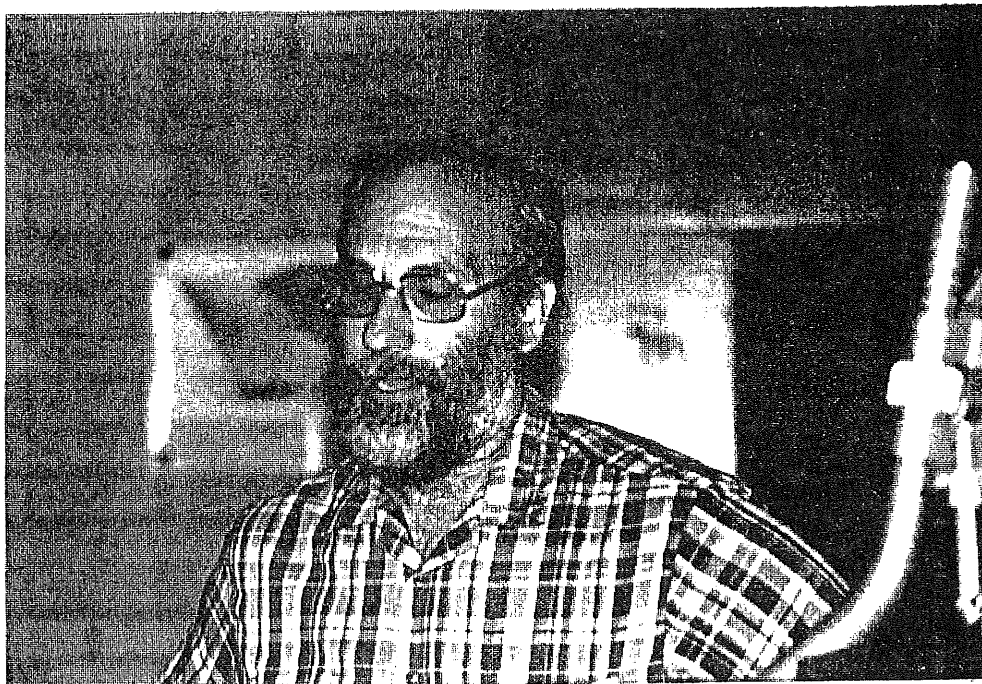
Bob Buckley, as always an erudite and well informed speaker, as well as funny. He was lying down for the talk, which is why the photo is long and skinny, not short and wide.



Steven Landers, who works with Esso, and proves that commercial applications are more than viable under UNIX.

Jeremy Firth was actually making his first journey (to my knowledge) to an AUUG meeting, from CSIRONET in Hobart.

Roy Rankine is an old hand, and came all the way from Sydney. He has just started with a new company, whose name I have forgotten.



Then, of course, the inimitable Chairman of AUUG, **John Lions**, put in his bit, but he gets too much publicity as it is, so I left him till later.

Well, that is it for the photos. There were of course more, including me and Chris Maltby, but of course I won't put them in. The photos are part of the AUUG archives, and can be made available en masse if desired.

Greg Rose.

Preliminary Notice and First Call for Papers.
Australian Unix systems User Group.
1986 Winter Meeting.
Australian National University, Canberra.
September, 1-2, 1986.

Introduction

The 1986 Winter Meeting of AUUG will be held at the Department of Computer Science at the Australian National University, Canberra, on Monday and Tuesday, 1 & 2, September, 1986.

Local arrangements are under the direction of Peter Wishart ['pjw@anucsd.oz'] at the Department of Computer Science, Australian National University, Canberra, ACT 2601.

Call for Papers

Papers on subjects related to the Unix system, or Unix-like systems are now called for. Papers relating to new developments, new applications, or new insights or research into the UNIX system are particularly requested, but tutorial and survey papers will also be very welcome.

The program committee consists of Chris Campbell (chairman), Ross Nealon and Piers Lauder. Abstracts should be sent to the Chairman or one of the committee members as soon as possible, preferably no later than Friday, 23 May.

Completed papers are needed for reviewing by Friday, 20 June. It is requested that final versions be ready for *troff* formatting and publication by no later than Tuesday, August 5.

Abstracts and full papers may be sent by ordinary mail to:
Chris Campbell,
c/- Olivetti Australia Pty Ltd,
140 William Street,
Sydney 2011.
[(02) 358-2655 x 466, or 'chris@olisyd']
or by ACSnetwork mail to Piers Lauder
[(02) 692-2824, or 'piers@basser.oz']
Ross Nealon is at the University of Wollongong
[(042) 27-0802, or 'ross@uowcsa']

The conference is one of the best way of sharing your UNIX

experience with others. Think seriously about presenting a paper at this conference.

It would be appreciated if you indicated your intention to the program committee as early as possible.

Hardware Display

There will be a display of appropriate hardware and software in conjunction with the conference, and vendors/manufacturers are invited to demonstrate their products. For more information on participating in this display, contact:

Peter Wishart
Department of Computer Science,
Australian National University,
Canberra City, ACT 2601.
[(062) 49-3850, or 'pjw@anucsd.oz']

Registration

The registration fee will be $50, with a $10 discount for AUUG members, and a further $10 discount for early registration. To qualify for the early registration discount, your registration must be received and paid for by not later than Monday, 18 August.

Each speaker will receive a free ticket to the conference dinner, and will be entitled to a complete remission of fees if a final version of the paper in publishable format (2-4000 words) is received by August 5.

What else ?

Details of accommodation and transportation will be announced in the next newsletter.

Further information will be posted to aus.auug in the near future, including registration forms. AUUG members will receive registration info via snail mail in due course. Non members, not participating in netnews, may write to Peter Wishart.

Australian UNIX* systems User Group
(AUUG)

Membership Application

I, ______________________________ do hereby apply for ordinary($50)/student($30)** membership of the Australian UNIX systems User Group and do agree to abide by the rules of the association especially with respect to non-disclosure of confidential and restricted licensed information. I understand that the membership fee entitles me to receive the Australian UNIX systems User Group Newsletter and I enclose payment of $_______ herewith.

Signed ______________________________ Date ______________

===

Name ____

Mailing address for AUUG information ______________________________

____

____

Telephone number (including area code) ______________________________

UNIX Network address ______________________________

I agree to my name and address being made available to software/hardware vendors YES ☐ NO ☐

===

Student Member Certification

I certify that ______________________________ is a full-time student at

____

Expected date of graduation ______________________________

Faculty signature ______________________________ Date ______________

===

Office use only 10/85

---

* UNIX is a trademark of AT&T Bell Laboratories

** Delete one

Australian UNIX* systems User Group Newsletter (AUUGN)

Subscription Application

I wish to subscribe to the Australian UNIX systems User Group Newsletter and enclose payment of $_________ herewith for the items indicated below.

Signed ____ Date ________________________________

===

☐	One years subscription (6 issues) available on microfiche or paper	$30.00
☐	Back issues of Volume 1 (6 issues) available only on microfiche	$24.00
☐	Back issues of Volume 2 (6 issues) available only on microfiche	$24.00
☐	Back issues of Volume 3 (6 issues) available only on microfiche	$24.00
☐	Back issues of Volume 4 (6 issues) available on microfiche, some paper copies	$24.00
☐	Back issues of Volume 5 (6 issues) available on microfiche, some paper copies	$24.00
☐	Subscribers outside Australia must pay more per volume to cover surface mail costs	$10.00
☐	Subscribers outside Australia must pay more per volume to cover air mail costs	$30.00

===

Name ____

Mailing address ____

____

____

Telephone number (including area code) ______________________________________

UNIX Network address ____

	YES	NO
I agree to my name and address being made available to software/hardware vendors	☐	☐

===

Office use only 10/85

* UNIX is a trademark of AT&T Bell Laboratories