1981

Evaluation of interrupt handling routines with a logic state analyser

P. J. McKerrow
University of Wollongong, phillip@uow.edu.au

EVALUATION OF INTERRUPT HANDLING ROUTINES WITH A LOGIC STATE ANALYSER

Phillip John McKerrow

Department of Computing Science,
The University of Wollongong,
Wollongong, N.S.W. 2500
Australia.

ABSTRACT

Use of a logic state analyser as a hardware monitoring tool is described. The interrupt handling routines of the UNIX* operating system, running on a Perkin-Elmer 7/32, were measured before and after major code modification. Clock handling is used as an example. Performance improvements included a reduction in the execution time of the common interrupt handler from 654 to 55 micro-seconds and a consequent reduction in clock handling time from 842 to 347 micro-seconds. Measurement methods and performance enhancements are discussed.

Acknowledgements

This work is supported by the Department of Science and Technology, Australian Research Grants Committee. Juris Reinfelds, Richard Miller and Ross Nealon, all of the University of Wollongong, have made significant contributions.

Keywords: Logic State Analyser, Performance Evaluation, Unix Operating System, Common Interrupt Handler, Clock Interrupt Handling, Portable Code, Multi-Tasking, Real-Time, Execution Time, Program Path, Hardware Monitoring Tool.

* UNIX is a trademark of Bell Laboratories

1. INTRODUCTION

As digital logic circuits increased in complexity, traditional fault-finding tools became inadequate. Oscilloscopes are ideal for monitoring one or two high speed signals; similarly, logic probes for low speed digital signals or for catching the occasional high speed pulse. Logic timing analysers were developed to assist in the fault-finding of complex, high-speed, logic circuits. This tool monitors many signal lines, e.g. computer bus. Readings are synchronised with an external clock; typically bus synchronisation and processor clocks. Thus the data read by the analyser is identical to the data latched into the device at the receiving end of the bus. Some logic timing analysers have been enhanced to indicate if bus activity has occurred between measurements.

Logic timing analysers met with success as engineering fault-finding tools. It was not long before they were being used to monitor program flow. From this the logic state analyser (figure 1) was developed. Manufacturers [1] claiming it to be a valuable tool for monitoring software, code optimisation and performance analysis.

Performance evaluation research at the University of Wollongong has included using a logic state analyser as a hardware monitoring tool. This paper discusses the analysis of the interrupt handling routines of the UNIX* operating system using a logic state analyser.

2. RESEARCH GOALS

System performance is studied using a number of measurement and modelling techniques including queueing models, simulation models and work-load models. Queueing network [2, 3] models are currently popular because of their mathematical tractability, intuitiveness and success in the field. The data required to develop and use these models is obtained using software and hardware monitoring tools. Some computer manufacturers are providing software monitoring tools as part of their systems [4]. Implementing software tools on an existing system requires a thorough knowledge of the operating system and a willingness to modify it. The greatest disadvantage of software monitoring is that it uses system resources interfering with performance.

Hardware monitoring does not interfere with the software being studied. Traditional hardware tools have provided limited information and if not designed into the hardware can be difficult to implement. Logic state analysers are the latest development in the stored-program class of hardware tools. They contain some "intelligence" and have great flexibility.

---

* UNIX is a trademark of Bell Laboratories

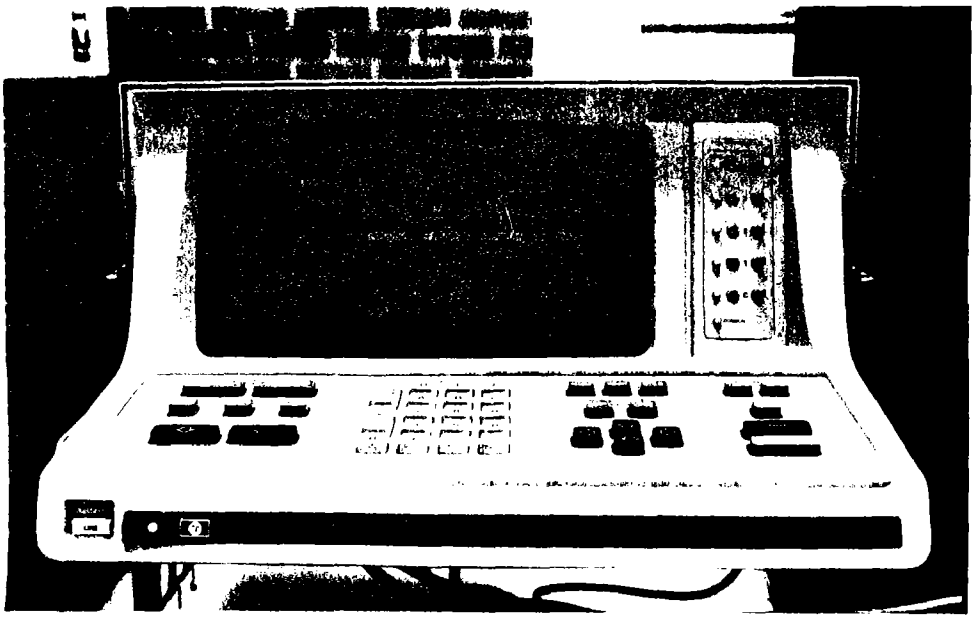


Figure 1. HP 1610A Logic State Analyser

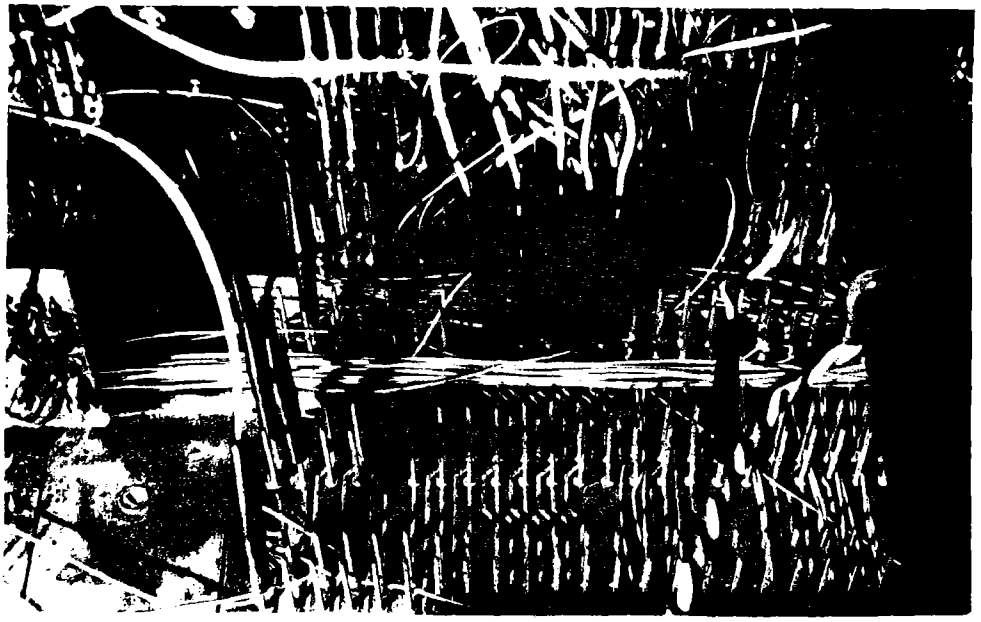


Figure 2. Connection of the analyser to the Computers Back Plane

Our interest is in the development of a hardware monitoring tool which will act like a "system microscope". It should be easily attached to any data flow path in the system in order to collect user selected information - data patterns, program flow, data transfer. Subsequently, the selected data will be analysed and performance indices calculated directly or from models. The specific goal of the work discussed in this paper is to evaluate the logic state analysers suitability for this task.

This goal needs to be seen in the context of wider performance evaluation goals:

(i) Performance evaluation of the UNIX operating system. UNIX was moved from Digital Equipment Corporation PDP-11 series of computers to Perkin-Elmer thirty-two bit series machines by Richard Miller [5,6,7] at the University of Wollongong. The Perkin-Elmer version is now available commercially, a world first in portability, motivating evaluation of its performance in an environment it was not designed for.

(ii) Development of an easy to use hardware/hybrid performance evaluation tool.

(iii) To develop our own understanding of performance evaluation theory and practice, and hopefully to make a small contribution to the field.

3. LOGIC STATE ANALYSER

The state analyser (figure 1) monitors thirty-two digital signals simultaneously with a maximum sampling rate of ten megahertz [8]. It can store sixty-four readings and display any sequential group of twenty-four of these. Each reading is synchronised with an external clock. The analyser can be set up to trigger on a specified sequence of data patterns. Once triggered it will record the occurrence of specified data patterns (states) until its memory is full.

The analyser is connected directly to the backplane of a Perkin-Elmer 7/32 processor (figure 2). Insulated connectors, supplied with it, push onto wire wrap pins making a convenient and safe connection. Connection of an analyser is outside the scope of the normal Perkin-Elmer hardware maintenance contract. The contract was renegotiated to overcome this problem. For some computers, Hewlett Packard sells cards that plug into the processor chassis thereby simplifying signal connection.

The Interdata 7/32 was the first of the Perkin-Elmer thirty-two bit minicomputer series; released in 1974. It uses 750 nano-second core memory and the fastest instructions take one micro-second, e.g. register to register add. Communication with memory occurs over a twenty bit address bus and a sixteen bit data bus.

Eight bits of the data bus, sufficient to read the operation code, the eighteen bit local address bus and some status lines (figure 3) are connected to the analyser. Perkin-Elmer 7/32 microcode differentiates between memory reads to fetch instructions and memory reads to fetch data. The signal used as a clock is produced as a result of the execution of an instruction fetch in the microcode. The clock is active low. On the rising edge of the clock both address and data buses are stable. The data bus contains the operation code fetched from the location specified by the address bus. Comparison of data bus information to a machine code listing establishes confidence in the analysers operation.

Communication between the user and the analyser is carried out using a series of interactive video displays which can be called and modified from the keyboard.

Input signals are split into logical groups with group selectable number base and logic polarity using the format display (figure 3). If the clock stops, e.g. operating system crash or loose connection, a slow clock warning is flashed. Signal activity in the preceding one hundred milli-seconds is indicated by a "!" symbol on the display below the label assignment providing an indication of bus activity or possible signal loss.

The trace display (figure 4,5) allows the user to specify:

(i) The sequence of states required for triggering and whether the trace should start, centre or end at the trigger point. Also the number of times that state is to occur, e.g. the tenth time through a loop.

(ii) The states to be traced. Normally when first looking at a routine all states are examined. As understanding of the routine increases it can be characterised by branches, loops and sequential sections of code. Often only the entry points of these sections need to be traced.

(iii) Measurement of time or number of states between readings.

(iv) Restarting of a trace trigger sequence if the states don't occur in the specified sequence.

These controls give the user considerable flexibility and power in measurement. A single path through a complex branching network (figure 6) or the program flow through multiply nested loops can be traced.

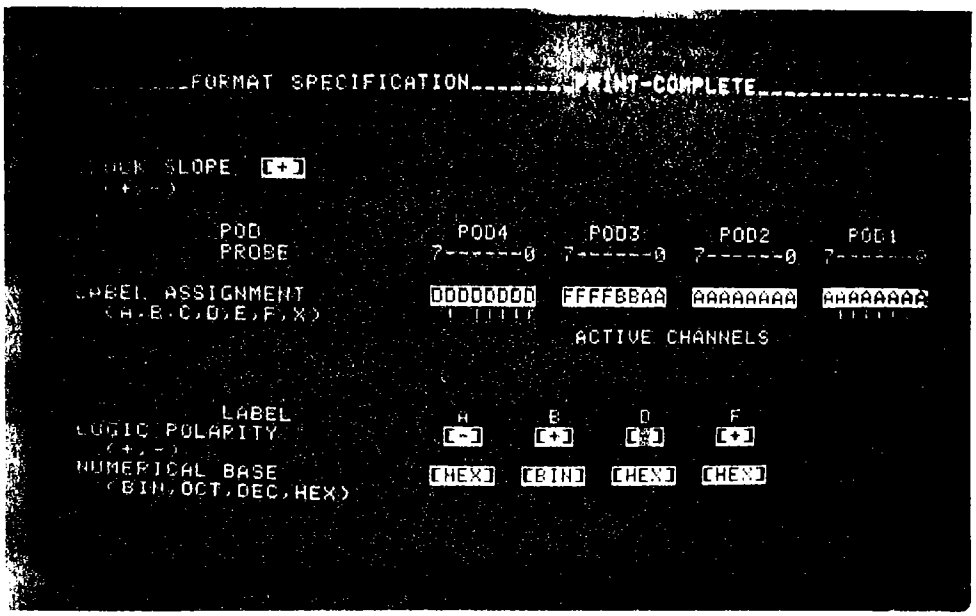


Figure 3. Format Display A - address bus
D - data bus
B - status - local/remote memory
- user/supervisor mode
! - signal activity

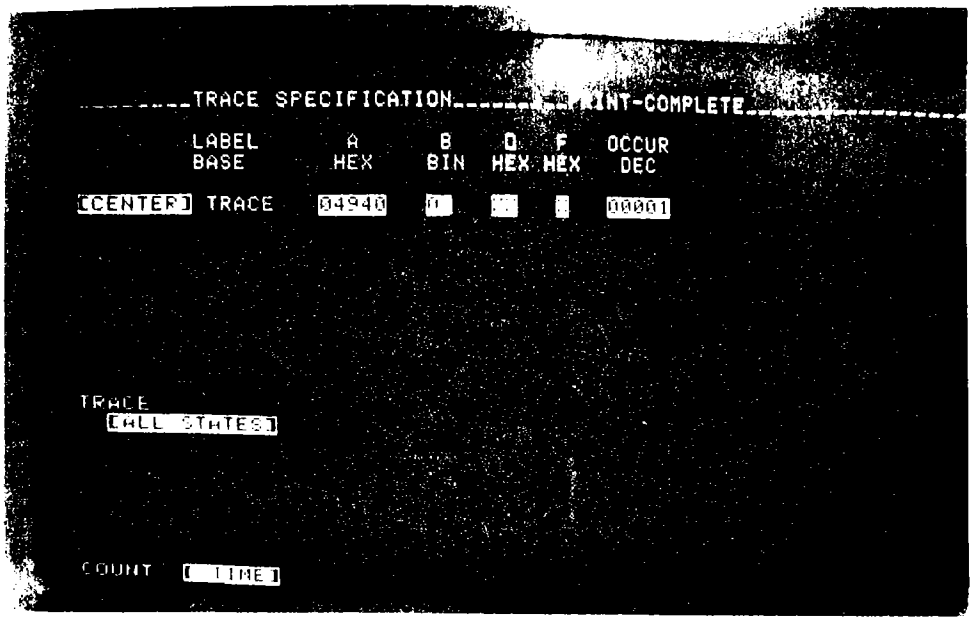


Figure 4. Trace Display set up to trigger on entry to clock routine and to trace all states before and after the trigger state. Time between states is to be recorded also.

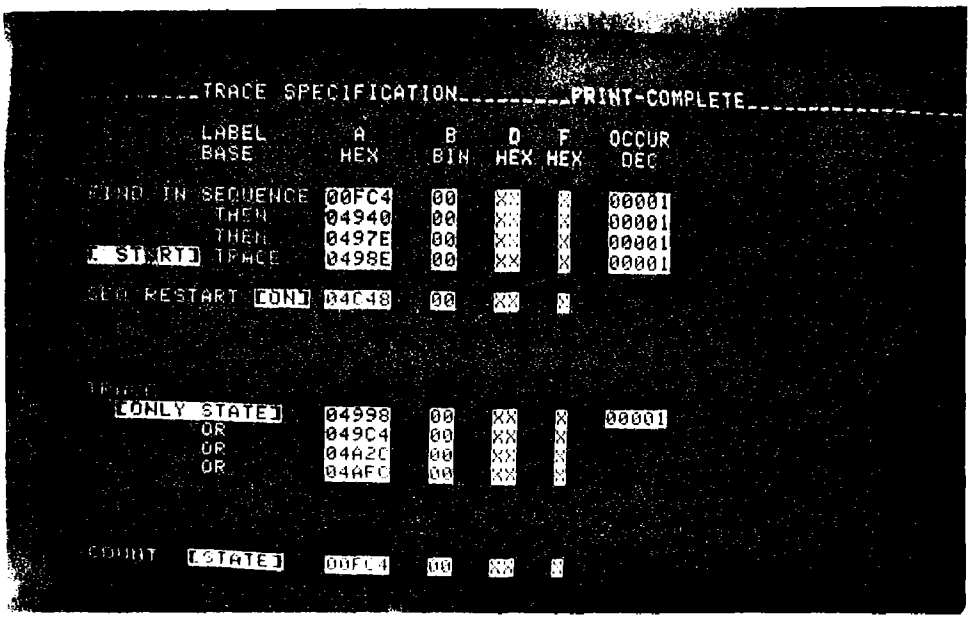


Figure 5. Trace Display set up to:-
(i) Trigger when specified path through clock routine is found.
(ii) Restart trigger sequence if routine exit is reached by another path.
(iii) Trace only states of interest.
(iv) Count the number of interrupts between states.

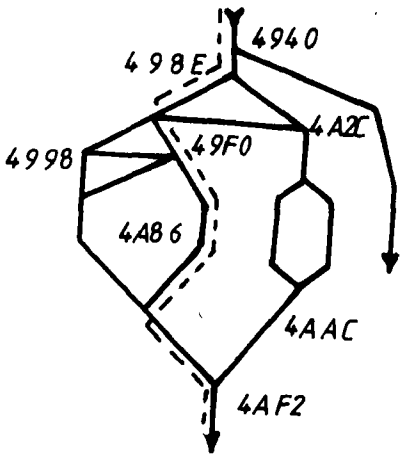


Figure 6. Complex network branching requires high level sequential trigger capability. e.g. Trace states from 4AF2 only when dashed path is executed.

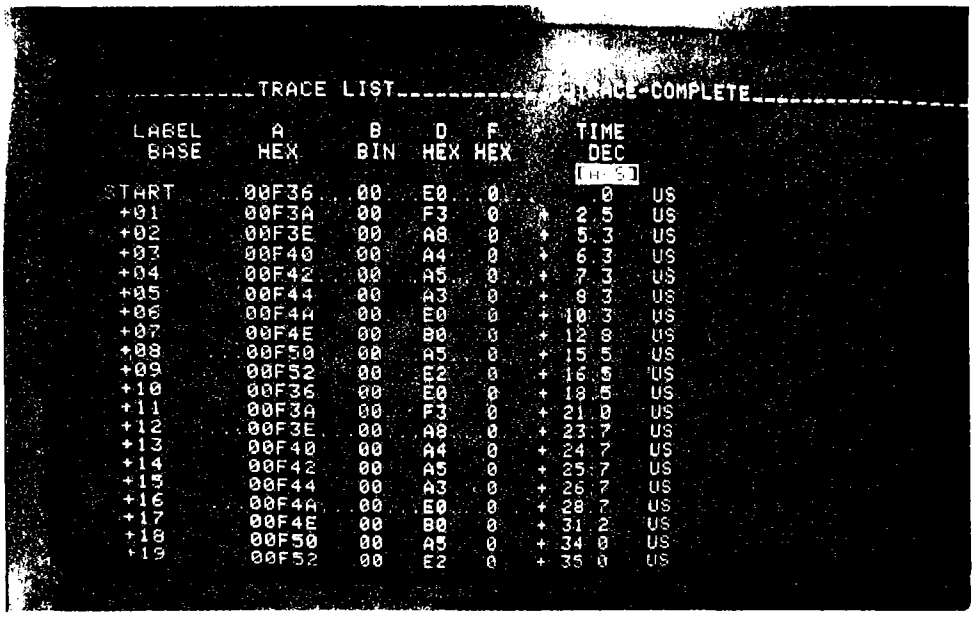


Figure 7. Display of traced states.

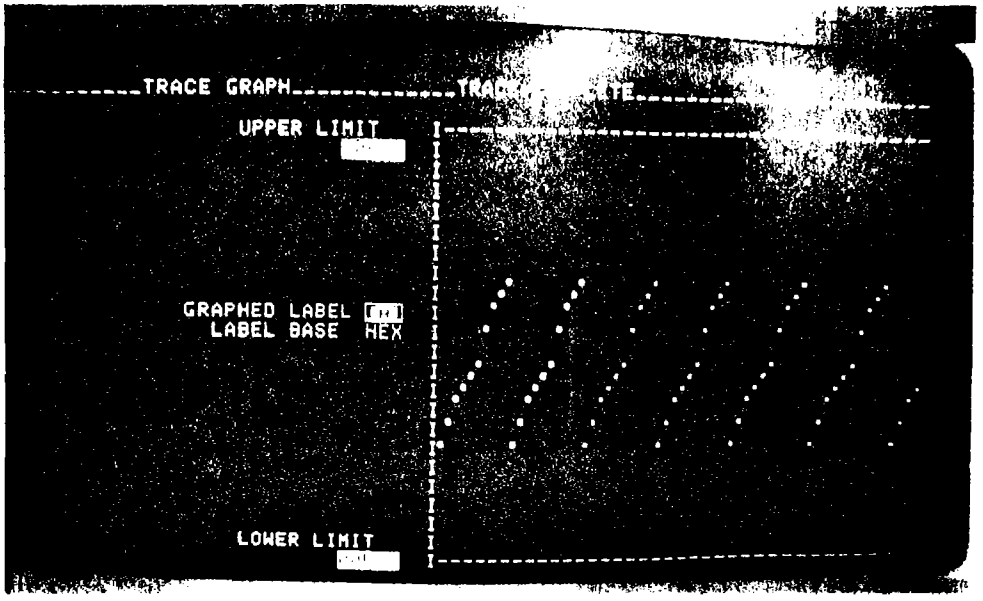


Figure 8. Graphical display of the addresses recorded in figure 7.

Measurement is initiated by the press of a key. When the measurement is complete the analyser automatically displays (figure 7) twenty-four of the sixty-four states. Other recorded states can be observed by rolling the display. The complete set of states for a particular label can be displayed graphically (figure 8) allowing rapid user analysis by pattern recognition.

4. APPROACH

Our initial approach was to measure and model the low level routines running in the computer, starting with a completely idle system. When no user programs are active the operating system is occupied solely with housekeeping functions, e.g. the time of day clock. From this we are gradually progressing to a fully loaded system.

The philosophy behind this approach is:

(i) These routines are relatively small and thus easy to modify.

(ii) Many of the routines constitute fixed overheads and thus have to be taken into account when measuring higher level behaviour.

(iii) We were interested in measuring the interrupt handling routines.

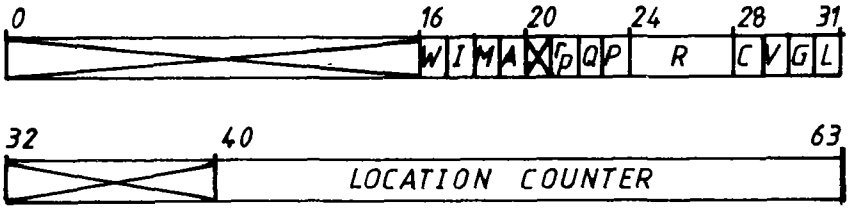
(iv) Only a small part of the operating system is written in assembler; the majority is written in the C programming language. To enable the transfer from PDP 11 to Perkin-Elmer the assembler code had to be rewritten making it an area of interest. The assembler code includes the common interrupt handling routines.

(v) The heart of any operating system is the scheduler and its associated interrupt handling routines. It was imperative that these be understood before more extensive measurements were made.

5. INTERRUPT HANDLING HARDWARE

A variety of interrupts, all of which can be inhibited by masks in the program status word (figure 9), occur within a Perkin-Elmer 7/32 system.

External devices interrupt the processor via the single level immediate interrupt. When the processor recognizes the request, the following sequence is initiated:



Bit 16	Wait state
Bit 17	Enable Immediate Interrupts
Bit 18	Enable Machine Malfunction Interrupts
Bit 19	Enable Arithmetic Fault Interrupts
Bit 21	Enable Relocation/Protection
Bit 22	Enable System Queue Service
Bit 23	Supervisor mode
Bit 24:27	Register set
Bit 28:31	Condition code

Figure 9. Program Status Word

(i) The current instruction is completed and the location counter updated to point to the next instruction.

(ii) The processor switches from the user register set to the supervisor register set. The 7/32 has two sets of sixteen thirty-two bit general purpose registers, selectable using a flag in the program status word. One is used by user programs, the other is reserved for supervisor functions.

(iii) The current program status word is saved in two of the supervisor registers.

(iv) The status portion of the program status word is set to:

a) disable all interrupts except machine malfunction

b) allow privileged instructions to be executed, and

c) disable memory relocation and protection.

(v) The interrupt is acknowledged. Device number and status are obtained from the interrupting device and placed in registers.

(vi) An address in the interrupt service pointer table is calculated from the device number. The start address of the interrupt service routine is obtained from this location and loaded into the location counter portion of the program status word.

(vii) Execution of the interrupt service routine is commenced.

6. COMMON INTERRUPT HANDLER

When UNIX was transferred to the Perkin-Elmer, the design of the original version was maintained. The assembler code was written to match the functions of the PDP 11 code as closely as possible. There is an entry in the interrupt service pointer table for each external interrupting device. The processor vectors to this address (figure 10) and commences execution. Software traps (supervisor calls) are handled in a similar manner.

The address of the interrupt handling program and the desired processor status is obtained. Then all interrupts and software traps branch to the common interrupt handling routine which performs functions common to all interrupt and trap handlers.

If the system was previously in user mode it is necessary to load the memory relocation registers to switch to the kernels address space. Then memory relocation and protection is re-enabled. Any information to be passed to the device interrupt handler is saved on the stack and the processor is switched to the user register set. User registers are saved and if a trap handler, not a device interrupt hander, is to be called interrupts are enabled.

The appropriate interrupt or trap handler is called to carry out the action necessary to service the interrupt or trap. On return from the handler, a check is made to see if a higher priority process may be waiting for service if the processor was executing a user process prior to the interrupt. Once a second the clock handler sets a flag to force this to happen. If so the process dispatcher is called to determine if a higher priority process is waiting and to switch to that process.

Then the common interrupt routine returns to the mode of operation prior to the interrupt or to a higher priority user process.

This code was studied with the state analyser. Execution times in micro-seconds of the different sections of code are shown on figure 10. Times in brackets are for a software monitor that can be added to the routine. This monitor measures system, user and idle time to the nearest millisecond using a precision internal clock. For every interrupt from a user process this tool costs 42 micro-seconds and 48 micro-seconds for an interrupt or trap from a kernel process. When a return is made to idle the monitor costs an extra 50 micro-seconds. Counters maintained by the tool can be accessed by accounting programs.

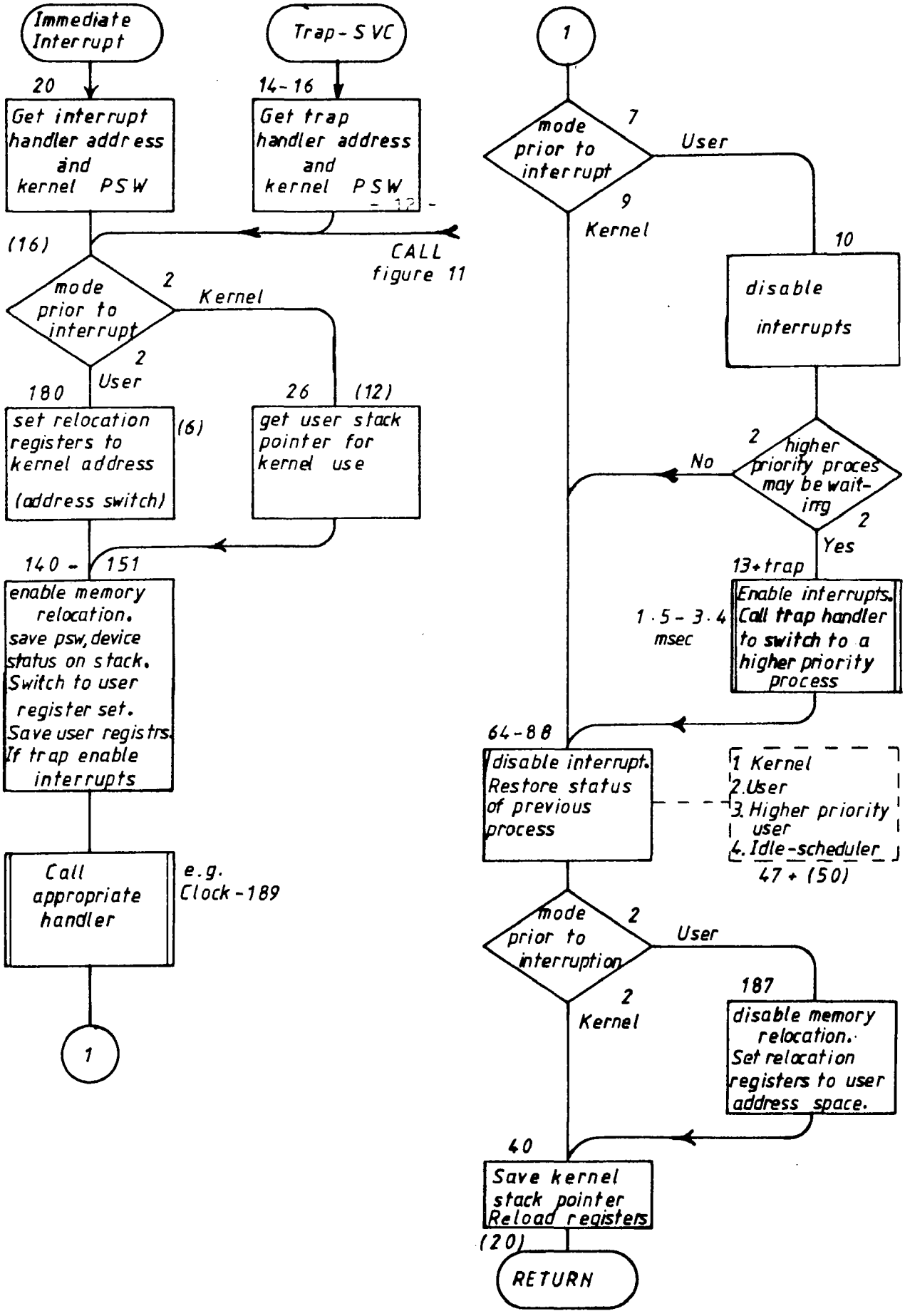


Figure 10. Common Interrupt Handling Routine.
Numbers above box's are the execution time in micro-seconds.
Numbers in brackets are the execution time of the software monitor that can be added.

Execution time for the common interrupt routine depends upon the processor state prior to the interrupt. The time taken by the high level interrupt and trap handlers varies from handler to handler. For two sections of the flow chart on figure 10 a time range is given. This is due to variations in the time taken for the effective address calculation in indexed instructions. Total time taken to execute the common interrupt handler code for various conditions is given in table 1.

If the machine is idle when interrupted, the interrupt is serviced and then the scheduler entered. Depending upon the action of the interrupt handler either a process is started or the system returns to idle. A return to idle via the scheduler takes 47 micro-seconds.

7. CLOCK INTERRUPT HANDLING

The purpose of the common interrupt routine described in the last section is to remove common actions, e.g. saving user registers, from the interrupt handling routines. Information required by the handler is placed on the stack. This defines a standard linkage and thus modularises the interrupt handling software. Adding a new interrupt handler is reasonably simple.

The regular nature of the clock interrupt, every ten milliseconds, makes the handler relatively easy to monitor. When the computer is idle only the clock interrupt handling routine uses any system resources. Two cleanup routines are started by the clock at regular intervals (0.5 and 5 minutes). Thus measuring and modelling the clock routine characterises an idle system.

The clock routine is written in the 'C' programming language. Assembler mnemonics produced by the C compiler are assembled to produce machine code. Thus an assembler listing of the C program is available. This was used in conjunction with symbolic debug to define program entry, branch and exit addresses for state analysis.

Table 1. Common Interrupt Handler Execution Time

Exception Type	Previous Mode	Return Mode	Basic Time Microseconds	Additional Time
Interrupt	Kernel	Kernel	303-336	handler
"	User	User	654-687	handler
"	User	Higher priority user	667-700	handler + switch
"	Idle	Idle	350	handler
Trap	Kernel	Kernel	297-332	handler
"	User	User	648-683	handler
"	User	Higher priority user	661-693	handler + switch

Table 2. Representative Times for Clock Handling in micro-seconds after Modifications

Clock tick during kernel process	341
Clock tick during user process	347
Clock tick during user process return to a higher priority process	3,151
1 second period, kernel process, no callouts	5,634
1 second period, user process, callouts pending	6,022
Clock tick during user process 1 callout-printer new line delay	1,475

General housekeeping functions are performed by the clock routine in the following order:

(i) Every ten millisecond clock tick:

(a) The contents of a memory location selected with the front panel switches are displayed on the front panel.

(b) A function can arrange to be called with a specified argument after a period of time, e.g. new line delay on a TTY driver. To do this an entry consisting of an incremental time, an argument and a function address is placed in the callout queue. The clock decrements the incremental time of the first entry in the queue. When it reaches zero the function is executed and the queue is updated.

(c) User or system time is updated for the current process.

(d) Time used by the current process is incremented for use in process priority calculations.

(ii) At the end of every one second period:

(a) Time of day measured in seconds, since the beginning of the year, is incremented.

(b) A flag is set for the process to give up the central processing unit to a higher priority process that may be waiting.

(c) Wakeup any processes that are waiting for a once second period to end.

(d) For all currently defined processes:

1. Increment their process time.

2. If any sleeping processes are ready to wake up modify their process table entry so that they are able to be started by the scheduler.

3. Reduce the priority of the current process and if it goes below a set level recalculate process priority.

(e) If the scheduler is waiting to rearrange things, e.g. as a result of swapping a program into memory from disc, wake it up.

The clock routine is a small routine but can take longer than the ten milli-seconds between clock ticks. So that interrupts are not lost the clock is able to interrupt itself while doing callouts and during the once-per-second processing. A flag

is set to ensure that these areas of code are not entered when processing the second interrupt.

Time taken to execute the routine depends upon the number of callouts and whether a one second period is up. The majority of the routine could be characterised by measuring individual sections and combining these as we did with the common interrupt routine. However, important information can be obtained using a simpler approach.

Set the state analyser to trigger on a clock interrupt, i.e. the address the interrupt vectors to. Trace the clock routine entry address; exit address and an address in each of the branches to be studied. If the interrupt handler entry and exit addresses are not known the common interrupt routine call and return addresses (figure 10) could be traced. Routine execution time is the time between these two states. Occurrence in the trace of branch addresses indicate the path taken through the routine.

When the system is idle there are no callouts, every one hundred clock ticks the one second code is executed and two processes are asleep. The time taken to service a clock tick is 541 to 548 micro-seconds consisting of common interrupt handling - 303 micro-seconds, clock handling - 188-195 micro-seconds and return to idle via the scheduler - 47 micro-seconds. If the software monitor is added an additional 92 micro-seconds is used. Thus servicing the clock takes a minimum of 5.41 percent of central processing time. Increasing the clock frequency to give a one millisecond tick would render the computer almost useless for any other processing. Reducing the frequency of the clock tick to one tick every one hundred milli-seconds would reduce clock handling to less than one percent of central processing time. Changing the clock frequency requires either modifying the line frequency clock hardware or using a less accurate quartz oscillator based precision interval clock.

Selection of clock interrupt rate is an important consideration in system designs. With faster computers this should be less of a problem.

When executing user programs servicing the clock takes even longer; 842 - 849 micro-seconds minimum. The common interrupt handler must switch from the user address space to the kernel address space before executing the clock handler and back again afterward. Loading the memory relocation registers in the memory access controller takes 174 micro-seconds. After completing the interrupt handling the routine returns directly to the user process.

Execution time and program path for handling interrupts to user processes can be measured in the following ways:

(i) Adding the times already measured for the different paths.

(ii) Measuring the routine in the same way as we did for the clock routine in an idle system.

(iii) When an interrupt occurs the central processor switches from user to supervisor mode by setting a flag in the program status word. This signal is available on the backplane and it was connected to the state analyser (figure 3). Total interrupt handling execution time is the time between the user mode instruction immediately prior to the interrupt and the next user mode instruction. Setting the analyser to trigger on the interrupt vector address and tracing only user mode instructions with the trace centred on the interrupt provides this information. Tracing address in branches again indicate program flow. To simplify initial measurement only one user process was running on the system; a "BASIC" program in an infinite loop.

A lot of useful information is provided by the preceding measurements but to model the clock routine completely two areas need further investigation - the callout section and the process table update section. Execution time for the callout section depends upon the number of callouts and the functions called. A function called to provide newline delay on a line printer takes 1,068 micro-seconds. Process table update execution time depends upon the number of current processes, the number of processes ready to wake up and the priority of the current process. If the state analyser could read memory locations these numbers could be determined during the trace simplifying measurements. At present, these numbers are determined either by simulating conditions or by using software tools to determine the current system state. Representative times for the clock routine are given in table 2.

8. MODIFYING THE COMMON INTERRUPT HANDLER

Unix was written as a multi-tasking operating system to support a number of programmers in an interactive environment. A common interrupt handling system is adequate for this application and has the advantages of modularity and ease of modification mentioned earlier. For real-time applications the response time of the common interrupt system is excessive; one of the reasons why Unix is not a real-time operating system. Real-time systems have to respond rapidly to interrupts either because of the frequency of the interrupts or the requirements of the interrupting device. This can be accomplished best by individual high speed assembler routines sacrificing modularity and ease of modification.

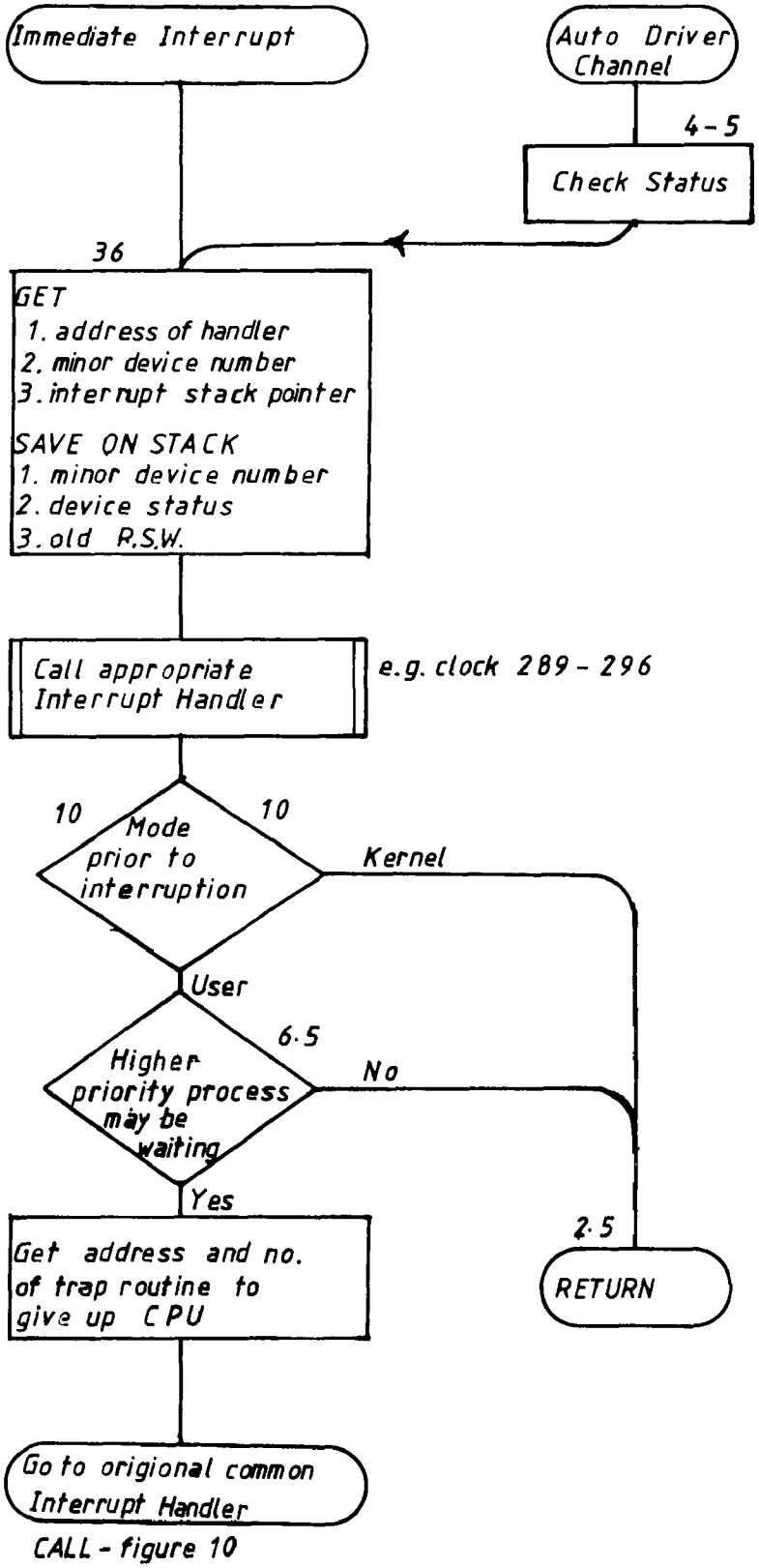


Figure 11. Modified Common Interrupt Handler

A Perkin-Elmer computer running Unix was linked to a Univac main-frame computer running Exec 8 for remote batch job submission. A synchronous RS232C link was used. Due to the cumbersome design of the Univac synchronous driver much of the information transmitted is protocol signals. Handshaking overhead effectively halves the data transfer rate.

To provide a usable link the common interrupt handling software was modified to reduce interrupt handling time in most situations (figure 11). This was achieved by making use of architectural features specific to the Perkin-Elmer range of thirty-two bit computers. Trap handling was not modified.

Time is saved in the following ways:

(i) An address switch is no longer done. Memory relocation is inhibited and program addresses are physical addresses.

(ii) A special stack was provided for the interrupt routines to use.

(iii) All interrupt processing is done using the supervisor register set. Thus there is no need to save and restore the user register set or to pass data from one set to the other via memory.

An additional architectural feature was utilised to make system reconfiguration easier. Interrupts no longer vector to separate locations to obtain handler information before branching to the common routine. Handler information is placed in tables. All immediate interrupts, except the auto drive channel, vector to one address. The physical device number has already been placed in a register by the processor. This is used to index into the tables to obtain the address of the handler and the minor device number.

In the case of an interrupt to a user process where the flag is set to indicate that a higher priority process may be waiting the new routine branches to the entry point of the original common interrupt routine (figure 10). Otherwise a return is made to the state prior to the interrupt; user process, kernel process or idle.

Execution time of the common interrupt routine for interrupts to user processes has been reduced from 654 to 55 micro-seconds. For interrupts to kernel processes it has been reduced from 303 to 49 micro-seconds. If a higher priority user process is waiting the execution time of common interrupt handling is 689 micro-seconds. Adding the software monitoring tool would increase the common interrupt routine execution time by 40 micro-seconds making it a very expensive tool.

Because memory relocation is switched off the interrupt handlers had to be modified also. The contents of the kernel

segmentation table is used to calculate the physical address of the user stack area from the virtual address. This address is used as a pointer to the user area structures. Clock handler time was increased from 189 to 292 micro-seconds as a result of this modification, negating some of the savings in the common interrupt handler. For a clock tick during a user process the clock service time has been reduced from 842 to 347 micro-seconds. Clock service time duing a kernel process has been reduced from 588 to 341 micro-seconds. Thus for the majority of interrupts processing time has been reduced.

Modifications to the common interrupt handler created some problems for state analysis. All interrupts vector to the same location making distinguishing one interrupt from another difficult. Interrupt specific information is contained in memory locations and registers. This problem was overcome by setting the analyser to trigger on the entry point of the interrupt handler rather than on the interrupt vector address.

9. CONCLUSION

State Analysis has proved to be a useful tool for measuring routine execution times and paths for performance evaluation. Routines can be examined in great detail - down to individual instruction level. Complete programs can be monitored by tracing program entry points, exit points and branch paths. Execution time of interrupt handlers can be measured by measuring the time spent in supervisor mode servicing the request. State analysis is a powerful and accurate measuring tool which does not interfere with the software it is monitoring.

Code written to be portable and easily modifiable often does not make use of specific features of the target architecture. Modifying code to use these architectural features improves performance at the cost of portability. This is shown by the reduction in common interrupt handling execution time from 654 to 55 micro-seconds. A surprising amount of CPU resources, 3.41 percent of CPU time minimum, are consumed by clock handling. Thus selection of clock interrupt rate is an important consideration in system design.

Some limitations have shown up in the state analyser for performance evaluation. These limitations and possible solutions are under investigation.

References:

1. Hewlett Packard, Seminar in Problem-Solving Concepts for Computer Systems, Part III, Complex Bus and Software Analysis Introduction, October 1978, pages 53-80.

2. P.J. Denning and J.P. Buzen, The Operational Analysis of Queueing Network Models, Computing Surveys, Volume 10, Number 3, September 1978, pages 255-261.

3. K.M. Chandy and C.H. Sauer, Approximate Solution of Queueing Models, IEEE Computer, Volume 13, Number 4, April 1980, pages 25-32.

4. S. Johnston, Development of a Software Monitor, Computer Performance, Volume 1, Number 2, September 1980, pages 61-80.

5. Richard Miller, UNIX - A Portable Operating System?, Operating Systems Review, Volume 12, Number 3, July 1978, pages 32-37.

6. S.C. Johnson and D.M. Ritchie, UNIX Time-Sharing: Portability of C Programs and the UNIX System, Bell System Technical Journal, Volume 57, Number 6, October 1978, pages 2021-2048.

7. B.W. Kernighan and J.R. Mashey, The Unix Programming Environment, IEEE Computer, Volume 24, Number 4, April 1981, pages 12-24.

8. Hewlett-Packard, Operating Guide 1610A Logic State Analyzer, February 1978.