AD-A055 152 ARIZONA UNIV TUCSON DEPT OF CHEMISTRY F/G 9/2
HISS: AN APPROACH TO HIERARCHICAL SYSTEM SHARING FOR LABORATORY--ETC(U)
APR 78 J B PHILLIPS, M F BURKE, G S WILSON N00014-75-C-0512
UNCLASSIFIED TR-9 NL

1 OF 1
AD
A055 152

END
DATE
FILMED
7-78
DDC

13

OFFICE OF NAVAL RESEARCH
Contract N00014-75-C-0512
Task No. NR 051-518
Technical Report No. 9

HISS: AN APPROACH TO HIERARCHICAL SYSTEM SHARING FOR LABORATORY COMPUTERS

J. B. Phillips, M. F. Burke and G. S. Wilson

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

Submitted for Publication in

Computers in Chemistry
April 1978

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-9 (Number 9)	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): HISS: AN APPROACH TO HIERARCHICAL SYSTEM SHARING FOR LABORATORY COMPUTERS.	5. TYPE OF REPORT & PERIOD COVERED: Technical Report 2-1-74 10-31-77
	6. PERFORMING ORG. REPORT NUMBER: Technical Report No. 9
7. AUTHOR(s): J. B. Phillips, M. F. Burke, G. S. Wilson	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0512
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Chemistry, University of Arizona, Tucson, AZ 85721	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR051-518
11. CONTROLLING OFFICE NAME AND ADDRESS: Material Sciences Division, Office of Naval Research, Arlington, VA 22217	12. REPORT DATE: 11 Apr 78
	13. NUMBER OF PAGES: 34 P.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research (Code 472), Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computers; software; operating systems; threaded code; structured programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An operating system for a laboratory computer network has been implemented utilizing the principles of hierarchically structured software and hardware. The system was developed using a threaded code technique which is shown to provide a greater degree of expandability and maintainability than is found with operating systems developed with less structured programming techniques. The continuous evolution of computing needs in a research

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014 6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (continued)

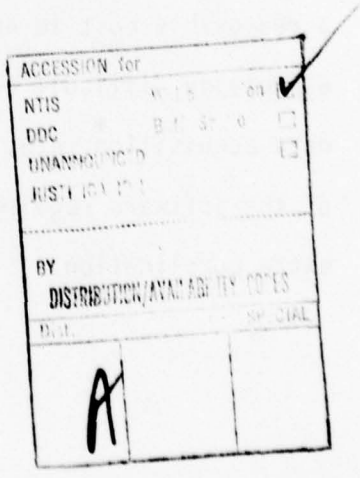
environment requires the ability of user-computer communication at a high level. This operating system provides an environment which facilitates the development of special purpose languages at a very high level by means of the threaded code.

The principles of the design of the system are discussed and contrasted with other reported approaches to laboratory computing. Examples which demonstrate the multilevel nature of the design and the advantages of this approach are described.

ABSTRACT

An operating system for a laboratory computer network has been implemented utilizing the principles of hierarchically structured software and hardware. The system was developed using a threaded code technique which is shown to provide a greater degree of expandability and maintainability than is found with operating systems developed with less structured programming techniques. The continuous evolution of computing needs in a research environment requires the availability of user-computer communication at a high level. This operating system provides an environment which facilitates the development of special purpose languages at a very high level by means of the threaded code.

The principles of the design of the system are discussed and contrasted with other reported approaches to laboratory computing. Examples which demonstrate the multilevel nature of the design and the advantages of this approach are described.



INTRODUCTION

In a typical university chemistry research environment, computer system support is required for a variety of research projects in areas such as chromatography, electrochemistry, and spectroscopy, among others. The type of computer support required varies greatly from experiment to experiment, but usually includes direct instrument control, data acquisition and storage, and computation of results. Instrument control may involve the use of fixed programs for a given class of experiments or it may require sophisticated interaction with the user. Data acquisition requirements can vary from low speed, high volume data to very high speed, relatively low volume data. Some experiments may require small amounts of real-time computations, while others, such as simulations, may run for hours or days. At any given time, approximately ten people may be employed in up to ten different research projects requiring some kind of computer support.

The fundamental problem in laboratory computing is making computer services available to the experimenter when and where they are needed at a reasonable cost in equipment and user time. This environment is an especially difficult one because of its very wide range of computation, data acquisition rate, and data volume problems. The constant evolution of the software required by the ever changing nature of research adds an extra complication.

Most of the specific problems of laboratory computing can be described in terms of communication or information flow requirements. Data acquisition obviously involves a flow of information from instrument to computer. Similarly, instrument control, data reduction, and presentation of results can be thought of as information flow problems. The need to change software through programming is also an information flow problem because the user must communicate his ideas to the computer.

The purposes of a laboratory computer operating system are to provide for orderly information flow among the hardware and software components involved in an experiment and to communicate with the user of the system. Specific services are provided to aid in moving data and results from experiment to user and commands from user to experiment. Other services are provided to aid in application software development. The effect of these sets of system services is to create an environment in which experiments may be more conveniently performed.

The environment provided by the system software is much more structured than that provided by the bare minicomputer hardware. This structure along with the specific services provided by the system simplifies the writing of application programs by reducing the number of decisions a programmer must make and the amount of detail included in each program. However, a more specifically structured environment is necessarily less general than the hardware environment. An operating system improves applicability to those kinds of information flow problems most likely to be encountered by reducing generality. A system designed to a wider

range of applications requires greater generality and, therefore, is more difficult to implement and less satisfactory for any one application. Thus, there are some very good turn-key systems designed for specific applications, but few general purpose research laboratory computer systems. The ideal laboratory computer system should provide a flexible environment as an overall structure plus some means of creating special purpose environments for specific classes of problems.

In a well-structured environment, there is at least some hope that independently designed programs can communicate with each other and that programmers can understand each others work. The structure should impose an automatic discipline on the system users, guiding them into compatible program and data constructions. This is especially important in a university setting where the users most often are students who have not yet developed much self-discipline in programming and will not be around long enough to maintain their own work. However, because of the requirements of flexibility in research projects, this discipline must be imposed without excessive restrictions on generality. It is better to provide standard means for solving problems rather than restricting the kinds of problems which can be solved.

This paper describes an innovative approach to an operating system which is capable of providing the low level generality needed to communicate with a variety of specific instruments as well as providing the high level of communication required by a variety of continuously evolving experiments.

Hierarchically Structured Hardware

The only economical way to provide the required variety of computer services in this research laboratory environment is through the use of a hierarchy of computers. For efficiency, expensive peripheral equipment and sophisticated software are shared, as much as possible, in one central system which is oriented toward the support of software development and data reduction computation. Such a medium to large scale central system cannot, however, be made sufficiently flexible and general to also support simultaneous data acquisition and experiment control under a wide range of conditions. Small, dedicated systems can better handle the data acquisition and experiment control problems, but, without a large investment in peripheral equipment and system software, they are not able to supply adequate software development and computation services. Dividing computer support services into two classes, operating in two different environments, results in better service in both environments.

Our approach to this problem has been to develop a computer system which provides for a division of labor between one central computer and several (currently two) peripheral computers. The hardware structure for this system, which is known as the Hierarchical Interactive Sharing System (HISS), is shown in Figure 1. The peripheral computers can devote their full attention to the instruments attached to them while the central computer provides the data storage capacity, computing power, and extensive peripheral equipment to effectively extract information from the data. Also, during the software preparation phase, the central computer's

more extensive memory and peripherals greatly improve the programmer's efficiency.

If support services are divided between local instrument control computers and a central computer, then some form of communication must be provided for the transfer of data, commands, and programs. This communication may be through user carried messages, common peripherals such as paper or magnetic tape, direct data lines, shared peripherals such as a single disc drive for two computers, or even shared main memory. The various communication possibilities have a wide range of data transfer rates, complexities, and costs.

The HISS system uses multiplexed direct parallel data line communication between the central computer and each peripheral computer. This technique offers sufficient data transfer speed for our experiments, allows the central and local computers to be located in different laboratories, and is not too difficult or expensive to build. The multiplexer electronics are illustrated by a simplified schematic in Figure 2. Under central computer software control any one of up to 15 peripheral

computer stations may be selected by the multiplexer. Information may then be transferred, in parallel, in either direction at rates of up to 300,000 16-bit words per second. The actual rate depends on the length of the connecting cable and the speed of the peripheral computer.

Software Requirements

Operating system software is required to support this hierarchical hardware organization and the application programs using it. This system must provide efficient communications between the central computer and peripheral computers so that system services can be effectively used by experiments while the flexibility of the peripheral computers is maintained.

The following goals were set for the design of this laboratory computer system:

1. Provide real-time data acquisition for a variety of unrelated experiments.
2. Provide mass storage for data and programming.
3. Provide interactive control of experiments.
4. Provide sophisticated data massaging during experiments.
5. Provide relatively fast computational speed.
6. Provide for appropriate maintainability of system software.
7. Provide for future expandability of system software.
8. Provide both the flexibility necessary for a research environment and sufficient structure to lessen author dependence of user software.
9. Provide for efficient chemist-computer communication at a level consistent with the problem and the chemists background.

A computer system may be put together and operated in a great many different ways. At the time this system was being planned, the only

viable choices for the hardware were minicomputers from various manufacturers. More recently, smaller and less expensive microprocessors have been developed. These are logical candidates for peripheral computers in a laboratory computer network. Experiences with the HISS system should be applicable to designs using them as well as to the minicomputers actually employed.

Software for microprocessor based laboratory systems is still poorly developed because their very low hardware cost discourages spending very much on system software. They have been used principally in the research laboratory as built in controllers for instrumentation rather than as user programmable computers. As parts of a hierarchical system, however, they could become an important addition to general purpose laboratory computing systems.

Software for minicomputer based laboratory systems is more readily available than it is for microprocessor systems, but it is still not satisfactory for hierarchically structured hardware or the kinds of experiments performed in this laboratory. The central computer operated initially under a fairly simple disc based operating system and each peripheral computer operated under an even simpler core based operating system. None of these operating systems are very well designed for a laboratory computer environment. They are good examples of older, larger scale computer system designs reduced in size and capabilities and put into a small laboratory computer. Communication between computers occurs through common I/O devices such as paper tape, a slow, bulky, and error

prone medium. Consequently, very little communication between computers occurred.

Most vendors of laboratory computers have some kind of real-time executive (RTE) operating system designed to enhance instrument control capabilities of their computers. The Hewlett-Packard RTE system, which is typical of this class, is a much improved version of their disc operating system. The added multiprogramming capabilities allow the computer to service instruments in a real-time mode while executing another program in whatever time is left over. This system is intended to be used for directly controlling several instruments simultaneously by time sharing one central processing unit. There may, however, be considerable system overhead and delay in responding to requests from attached instruments, especially if more than one of them can produce data at a fairly high rate.

Time sharing of one processor for several independent operations must result in increased operating system overhead and slower response. Of even more importance, however, is the resulting software complexity. When central processors were very expensive, there was some economic justification for sharing one processor among several tasks. The expense of writing and maintaining extremely complex software could be recovered by making one processor do the job of several. This reason for multiprogramming systems is no longer very important and the disadvantages of extremely complex software are becoming more apparent. Such systems are difficult to maintain even by the original authors. Ordinary users

have little hope of being able to make any changes at all in them to accommodate new kinds of hardware or experiments.

Communicating with the Laboratory Computer

Software development, either the creation of new programs or the modification of old ones, requires some form of communication between programmer and computer. Having efficient communication is especially important in a chemical research environment because of the evolving nature of the software and the often limited programming experience of the users.

In some laboratory computer systems, for example BASIC language systems, the operating system and the programming language used to communicate with it are practically indistinguishable. In others, there may be distinct divisions between system and languages. But, since in all cases the programming languages available are a major factor in defining the environment in which users work, we will consider them as parts of the operating system. The programming languages available with small laboratory computer systems are usually quite limited both in number and capabilities.

The operating systems themselves are usually written in a low level assembly language compounding system software maintenance problems. Assembly language may also be used for application programs and usually must be used for those few programs directly concerned with instrument interfacing. For most programs, however, the excessive generality of the language results in unnecessary program complexity because

of the large number of coding details required. To make any changes in a program whenever computing needs change, a user must become familiar with all these details.

Higher level languages help improve programmer to computer communication. Because they are more specific, higher level languages require less detail to express those kinds of algorithms for which they are designed. The only higher level languages commonly used with small laboratory computers are FORTRAN and BASIC, both of which were designed for other purposes and do not completely satisfy the needs of a research laboratory environment.

Systems built around the BASIC language have the advantages of being interactive and relatively easy to learn (Wilkins and Klopfenstein, 1972). These two properties are especially important in a research laboratory environment because of the continually changing software and the lack of programming experience of some users. As long as experiments remain fairly slow and simple, BASIC can provide an excellent means of communicating with the laboratory computer.

BASIC has two fundamental limitations, however. First, it is an interpretive language, and, therefore, too slow to keep up with some high speed instruments or to perform extensive computations, such as simulations. Second, it is an easy to learn and use language only if the programs are very small. Its lack of modular structuring is a serious deficiency in a research environment. The programmer cannot write a series of modules which may be combined to make programs, but, instead, must write each program, no matter how complex, as a unit.

FORTRAN-based laboratory computer systems avoid the more serious limitations of BASIC language systems at some expense in increased system software complexity and reduced interactive capabilities. Most importantly, FORTRAN systems allow the user to build programs from modular subprograms written in either FORTRAN or assembly language. A complex program may be broken into smaller, more managable subprograms with only limited and well-defined communication between them. And, if the subprograms are made general enough, they can be gathered into libraries for use by other programmers working on future projects.

Despite its advantages over BASIC, FORTRAN remains a poorly structured language. For problems limited to input, calculation, and output, it is adequate. But, if many decisions are required or complex data structures are involved, then programs become excessively convoluted and confusing. Subprogram structures are less efficient and harder to use than they are in many other languages.

There are quite a number of less well-known (to chemists) systems and languages which could be applied in the chemical laboratory. A very elegant example is the UNIX system which was written for the PDP-11 at Bell laboratories (Ritchie and Thompson, 1974). This system has been very successful because it provides a simple, yet powerful, environment for users and their programs. Its strongest point seems to be consistency in I/O and file structures allowing programs to very effectively communicate with each other and with users.

FORTH (Moore, 1974) provides a limited laboratory computer operating system and a high level programming language based on hierarchical structuring and is available for a number of different computers, including the HP 2100 series. At the time this system was designed, FORTH was limited to dedicated computer-instrument systems. It was, therefore, unacceptable for use in a central computer, although potentially useful in peripheral computers. Recently, it has been extended to include multiprogramming capabilities similar to an RTE operating system (Rather and Moore, 1976). This extension, while useful in some applications, suffers the same limitations as other RTE type systems.

Communicating with a laboratory computer using FORTH is much easier than with the programming languages available on most systems. The criticisms of it as a programming language are more of a technical nature rather than of the principles upon which it is based. Because of its poor syntax, it tends to be hard to read and does not encourage the use of comments. A lack of localization structuring makes in vulnerable to foolish mistakes by uninformed users. In some situations, it is inefficient and slow due to excessive overhead processing.

The Threaded Programming Technique

The HISS operating system software is implemented using a threaded programming technique (Bell, 1973; Dewer, 1975). A threaded program consists of a string of indices indirectly pointing to subprograms which, as illustrated by Figure 3, may be either machine code service routines or other threaded programs. A threaded program itself does not contain any

executable code. It just specifies the order in which subthreaded programs and service routines are to be executed. Ultimately, all computation takes place in machine code service routines which may be either core resident or mixed with threaded programs.

The essential feature of this threaded programming technique is its very extensive use of subprogram structures. The structuring of programs using modular subprograms is generally agreed to be desirable (Wright,1976), and most programming languages provide some means of accomplishing it. For example, the FORTRAN language includes both a CALL statement for explicitly transferring control to a subprogram and a function expression for implicitly transferring control during arithmetic calculations. This threaded programming technique takes the subprogram idea to its logical conclusion, making all program statements, except those in machine code service routines, into implicit subprogram calls.

It is not practical to use this extreme subprogram structuring with any of the common programming languages. First, transferring control to a subprogram is usually much too slow. For example, in the FORTRAN system provided with the HP 2100, subprogram calls require hundreds of microseconds. It is assumed in the operating system and FORTRAN compiler designs that subprogram calls are relatively rare in comparison with arithmetic operations and, therefore, are of little importance. Second, the syntax of common languages permit, but do not encourage, the use of subprogram structures. In the FORTRAN language, each subprogram must be

explicitly defined by the programmer, is listed by the compiler on a separate page, and can be called only through limited kinds of syntax. Again, the language design assumes that subprogram calls are relatively rare and, therefore, that the readability of subprogram structures is unimportant. Subprogram structures in BASIC are even less efficient and readable.

FORTH (Moore, 1974; Rather and Moore, 1976) and a similar laboratory computer operating system, CONVERS (Tilden and Denton, 1978), both make use of the threaded programming technique. In FORTH, a threaded program is a string of absolute addresses; while in CONVERS it is a string of subroutine call instructions. Both of these structures are less efficient in memory space than the indices used in HISS. FORTH has a very small fixed kernel consisting of little more than the threaded program interpreter and a couple of I/O drivers. Everything else is compiled from FORTH source code when the system is loaded. CONVERS includes a large number of fixed machine code service routines along with the interpreter and I/O drivers and is compiled by a standard assembler. HISS includes even more fixed machine code service routines to provide an interface with a more sophisticated multiprogramming monitor. CONVERS is the simplest to implement and is the fastest in execution on the microcomputers for which it was designed. FORTH, however, is at least as fast for most minicomputers and is less machine dependent. HISS requires a microprogrammable machine for efficient implementation and so, at the lower levels, is very machine dependent. The high level threaded program syntax, however, is machine independent as are both FORTH and CONVERS.

An interpreter implemented in microcode for the HP 2100 executes threaded programs by transferring control from service routine to service routine in the sequence specified by the threaded program. The interpreter is similar to the one described by Bell (Bell, 1973) and works in the following way:

Step 1. Increment PC on top of control stack.
Step 2. Fetch S from PCth address of memory.
Step 3(a). If S denotes a service routine, execute it.
Step 3(b). If not, push the address of S to control stack.
Step 4. Go to Step 1.

A control stack provides a place to save nested threaded program return addresses. During each cycle of the interpreter, the address of the current threaded program instruction is present on top of the control stack. All that is required to begin interpretation of a threaded program is to push its address onto the control stack. The return address to the calling threaded program is automatically saved on the control stack during subprogram execution. Upon completion of a threaded program, a special service routine, NEXT, which is also implemented in microcode, is executed to delete the address on top of the control stack returning control to the calling threaded program.

This threaded program interpreter is quite simple and fast, requiring only an average of 8 microseconds per interpreter cycle. This is nearly as fast as the most limited assembly language subroutine call and is much faster than the hundreds of microseconds required for a typical FORTRAN subroutine call. The interpreter and associated service routines

use 241_8 of the 400_8 instruction words available in the writable control store module. The rest of this space is available for other uses including user defined instructions. A more complete discussion of this threaded program implementation has been described previously (Phillips, et al, 1978).

The high speed of subprogram transfers provided by the threaded program interpreter would be lost without corresponding efficiency in passing parameters to and returning results from subprograms. The most efficient means of communication between subprograms, through a data stack, is used in the HISS system. This data stack, although implemented in a fashion similar to the control stack, is distinct from and independent of it. Any kind of data may be stored on the data stack, but the control stack is limited to subprogram return addresses and a few related program flow items.

Most of the stack oriented computers currently in use, for example, the HP 3000, use a single stack for both data and subprogram control information (Bulman, 1977). Each subprogram call and return requires a complex series of operations to allocate stack memory and communicate parameters or results. This single stack architecture is appropriate for languages in which subprogram calls are relatively rare, but it cannot be used for a threaded program system.

Hierarchically Structured Software

The threaded programming technique, when combined with a suitable compiler, allows the development of highly structured hierarchical programs which, in addition to being smaller in size than equivalent

conventionally structured programs, are also easier to write and understand. Each threaded program is defined in terms of more basic or general purpose instructions which are, in turn, programs defined from even more basic instructions. At the bottom of this hierarchical structure are microcode and assembly language service routines which only rarely need to be changed. Above them are a series of levels of threaded program building blocks defining all of the operations needed for a particular kind of application. At the highest level are specific applications programs. The language available for implementing a threaded program becomes more specific as higher levels are added to the hierarchical structure. It is also the higher levels which most often have to be modified in a research environment. This hierarchical structure separates the implementation details of a program from the overall logic making the program easier to understand and modify.

Some example threaded program procedures are given in Figure 4. These procedures are part of the intermediate level coding for a chromatography simulation language or system (Phillips and Burke, 1978). They are written in terms of instructions implemented as lower level threaded programs and service routines such as HITRATE and R.EXP. These lower level instructions are much more specific than the machine's general purpose assembly language would be. Consequently, the example threaded programs are much shorter and easier to understand than they would be if implemented in any general language. The first two procedures in Figure 4, HITSURF and STUCK, are, in turn, used as instructions in the third procedure, ADSORB. By the addition of these two threaded programs procedures, the

language becomes even more specific and following threaded programs can be written on an even higher level.

The lowest level service routines, such as R.EXP, an exponential random number generator, are unlikely to ever need any changes. The code for these routines is compiled and stored separately to keep it from cluttering the higher level threaded programs with excessive detail. Users working at this intermediate level in the simulation language hierarchy would be interested only in what an instruction does and not how it does it. These intermediate level threaded programs would, in turn, be uninteresting implementation details for a user manipulating models of chromatographic processes.

Most of the logic of the operating system is implemented with threaded code in a fashion similar to the chromatography simulation example illustrated by Figure 5. Included among these routines are executive call interpreters, a file manager, and interactive user interface. Threaded code is resident in a virtual memory with currently active pages in core and inactive pages kept in a support file on disc. This allows for future expansion of system functions without increasing the amount of core memory occupied since additional threaded programs remain on disc until actually used and then automatically displace other unused programs from core.

The minimum amount of virtual memory support core for efficient system operation is 65 pages of 32 words each. Core resident assembly

language routines and system communication tables require about 7,000 words of memory. Thus, the total HISS system requires about 9,000 words of core memory leaving 23,000 words for user programs or data.

The operating system software is organized in a hierarchical structure as shown in Figure 5. Control flows from higher to lower levels in this structure; that is, routines at a higher level can direct the activity of routines at lower levels, but lower level routines can only return information or make requests of higher level routines. The user interface to the computer system is through the system supervisor at the highest level. Peripheral computer software is at a lower level than the communication network manager and, therefore, peripheral computer programs can make requests but cannot control the central computer system.

All of the system software, except for a small network communications driver, resides in the central computer. Thus, the peripheral computers are dedicated to applications programs, while the central computer provides operation system services sharing its resources among the users. An applications program in a peripheral computer needing some system service makes a request through its communications network driver. The central computer software then answers this request by taking control of the peripheral computer, interpreting its request, and either performing the requested service itself or, in the case of data transfers, telling the peripheral computer how to perform the required operations. Only the central computer software is capable of interpreting system requests, whether from a user at a terminal, a program in the central computer, or

an applications program in a peripheral computer. In all cases, the central computer plays the role of master and the peripheral computers act as slaves under its control so long as the communications network driver is present in their memory. The peripheral slave computers can make requests and the central master computer will carry them out if it understands them and accepts them as appropriate.

Conclusion

The hierarchical organization used extensively in the design of this system has proved to be a very useful approach for both hardware and software components of laboratory computer systems. It is essential to use definite structures in the design of anything as complex as an operating system. The hierarchical structure is one of the most general possible allowing it to be used in many different places in the system. It is also an ideal way of dividing complex problems into smaller and simpler pieces with the usual result being more general solutions and software is easier to understand, maintain, and extend.

Most of the goals originally set for the system have been met. The peripheral computers provide very good real-time data acquisition. The resulting data can be transmitted at high speed to the central computer and stored on disc. Interactive control of experiments is provided by user programs in the peripheral computers. The system helps only indirectly in this by rapidly moving data out, but at least it does not limit the user program as many systems do, since it is located in another processor. Sophisticated data massaging during experiments is

available in the central computer. Fast computation is available as demonstrated by a chromatography simulation program (Phillips and Burke, 1978). The system software is extremely modular making it much more maintainable than most operating systems. This modularity, plus the hierarchical structure provided by the threaded programming technique, should make future expandability of the system easier and provide hardware independence for much of the system programming. The flexibility necessary in a research environment is provided by giving complete control of the peripheral computers to the user programs resident in them during performance of experiments. This is the most important place in the system to have extremely good flexibility. Other user programs are more structured by the system supplied services.

The HISS threaded programming technique, one of the most useful results of this project, could be improved in two ways. First, the present threaded code compiler is based on a general purpose macro interpreter. This approach allowed extensive experimentation in the syntax of threaded programs in the early stages of this project and resulted in a more readable and easier to use language. Replacing this compiler could improve the system's interactive capabilities. Second, the threaded program technique should be made available in peripheral computers. This would require that microprogramming capabilities be added to them.

As a prototype operating system for laboratory computers, this design includes several features which are likely to become more important as newer and less expensive hardware becomes available. The continuing

decrease in prices of processors and memory is changing the economics of laboratory computing in favor of distributed systems. And, as the hardware designs change, so must the software systems change.

Real-time multiprocessing systems in which a single processor is shared among several concurrent tasks are becoming much less attractive for laboratory computing. By assigning separate, less powerful, processors to each task, much can be saved in software complexity with little cost in hardware. The HISS system uses fairly expensive minicomputers to do this because they were what was available during its design, but the principle of hierarchical processors applies just as well to the newer and less expensive microprocessors. Having a peripheral computer whose only responsibility is to service a particular instrument greatly reduces software cost and complexity while improving service to that instrument. Expensive peripheral equipment can be shared by attaching it to one central computer.

The HISS system provides capabilities useful in a chemical research laboratory which were not previously available and is an advance in the design of laboratory computer systems. Its hierarchical design is a very good approach to building such an operating system and makes it much more maintainable and extendable by its users. Systems like this one are likely to become more common and more important as the need for and uses of laboratory computers increases. This approach to software design can be expected to influence future computer hardware resulting in laboratory computer systems better able to assist in chemical research.

Detailed listings and other system documentation will be made available to interested readers.

ACKNOWLEDGEMENT

The authors express their gratitude to M. Bonner Denton for the development of the communication network hardware used in this study. Also, thanks are due to A. Swartz, C. Stauffer and M. Parker for their contributions to the programming.

This work was supported in part by the Office of Naval Research.

References

Bell, J. R. (1973), "Threaded Code," Comm. ACM 16, 370-374.

Bulman, D. M. (1977), "Stack Computers: An Introduction," Computer 10(5), 18-28.

Dewer, R. B. K. (1975), "Indirect Threaded Code," Comm. ACM 18, 330-333.

Moore, C. H. (1974), "FORTH: A New Way to Program a Mini-Computer," Astron. Astrophys. Suppl. 15, 497-522.

Phillips, J. B. and Burke, M. F. (1978), "Digital Simulation of Chromatographic Processes," manuscript in preparation.

Phillips, J. B., Burke, M. F. and Wilson, G. S. (1978), "Threaded Code for Laboratory Computers," Software Prac. and Exper. 8, 222-333.

Rather, E. D. and Moore, C. H. (1976), "The FORTH Approach to Operating Systems," Proc. ACM, 233-240.

Ritchie, D. M. and Thompson, K. (1974), "The UNIX Time-Sharing System," Comm. ACM 17, 365-370.

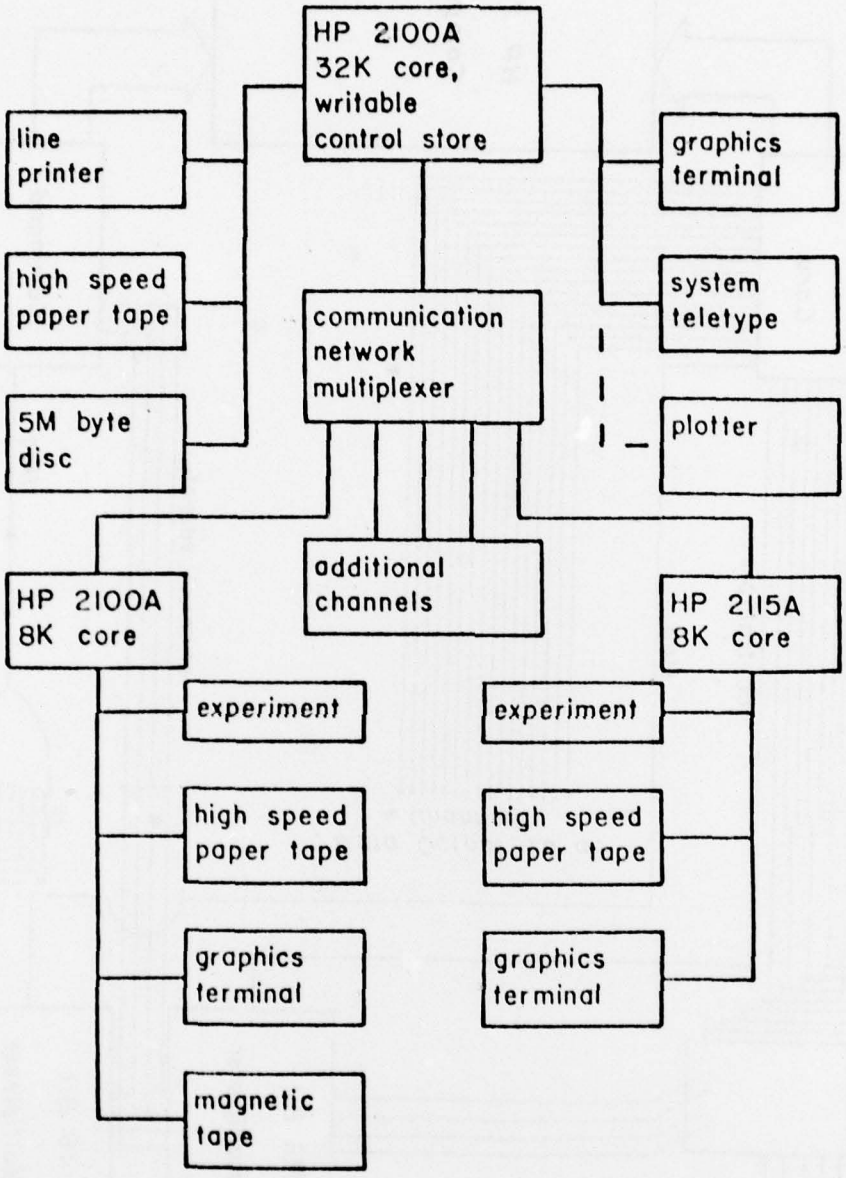
Tilden, S. and Denton, M. B. (1978), "CONVERS-an Interpretative Compiler," Byte (manuscript submitted).

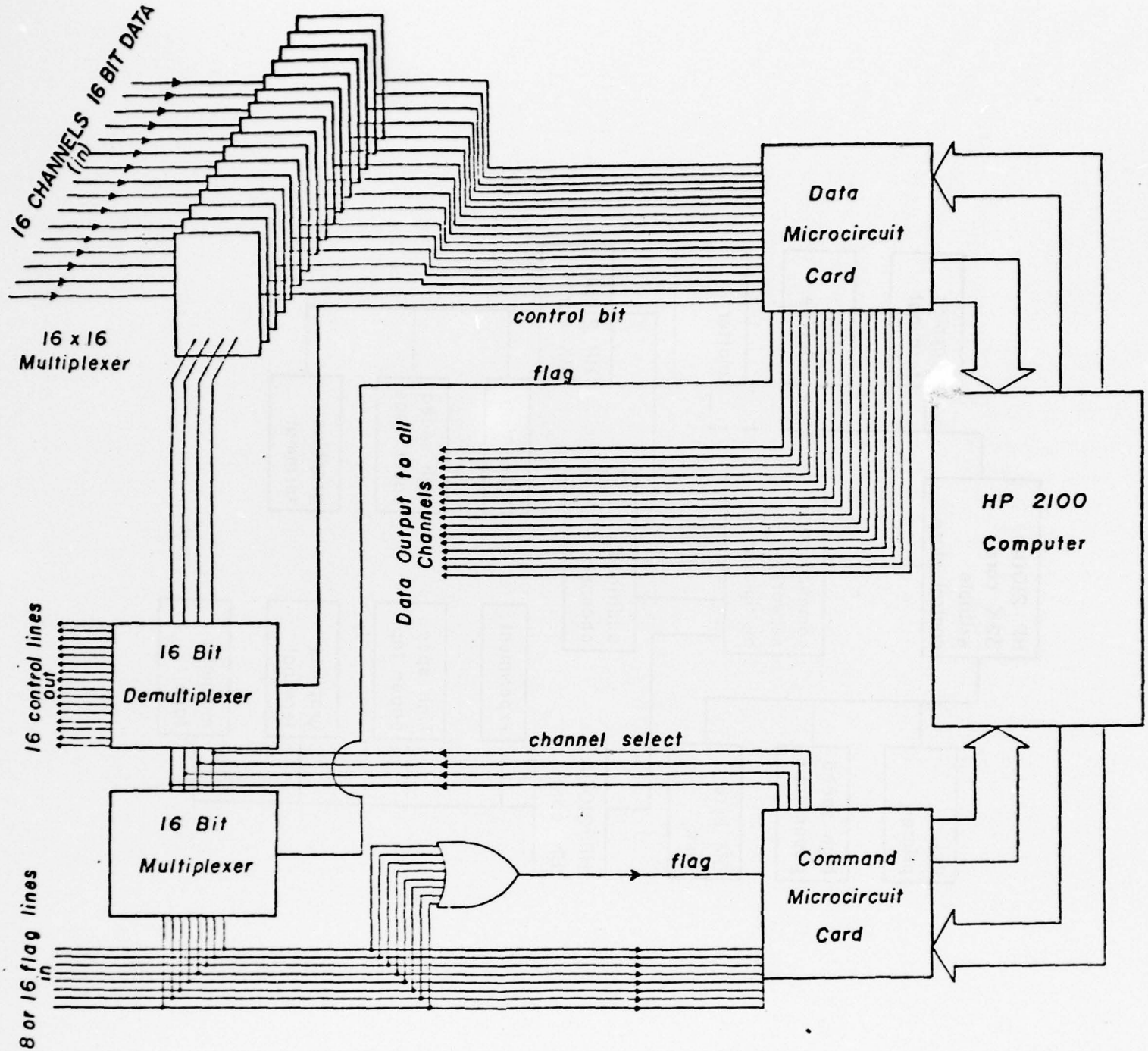
Wilkins, C. L. and Klopfenstein, C. E. (1972), "Laboratory Computing with Real-Time BASIC, Part I: Background," Chemtech., 560-568.

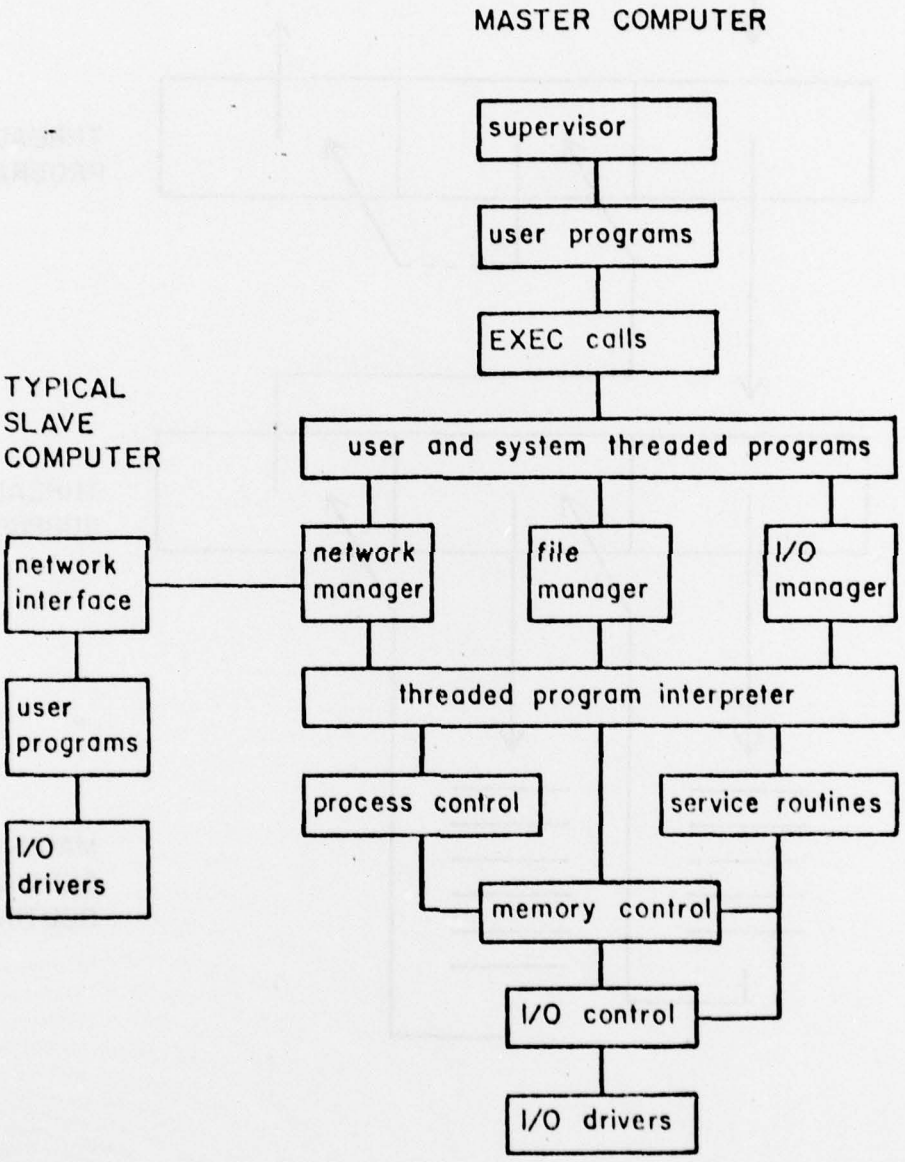
Wright, S. (1976), "Invocation--the Key to Program Structure," Comm. ACM, 361-364.

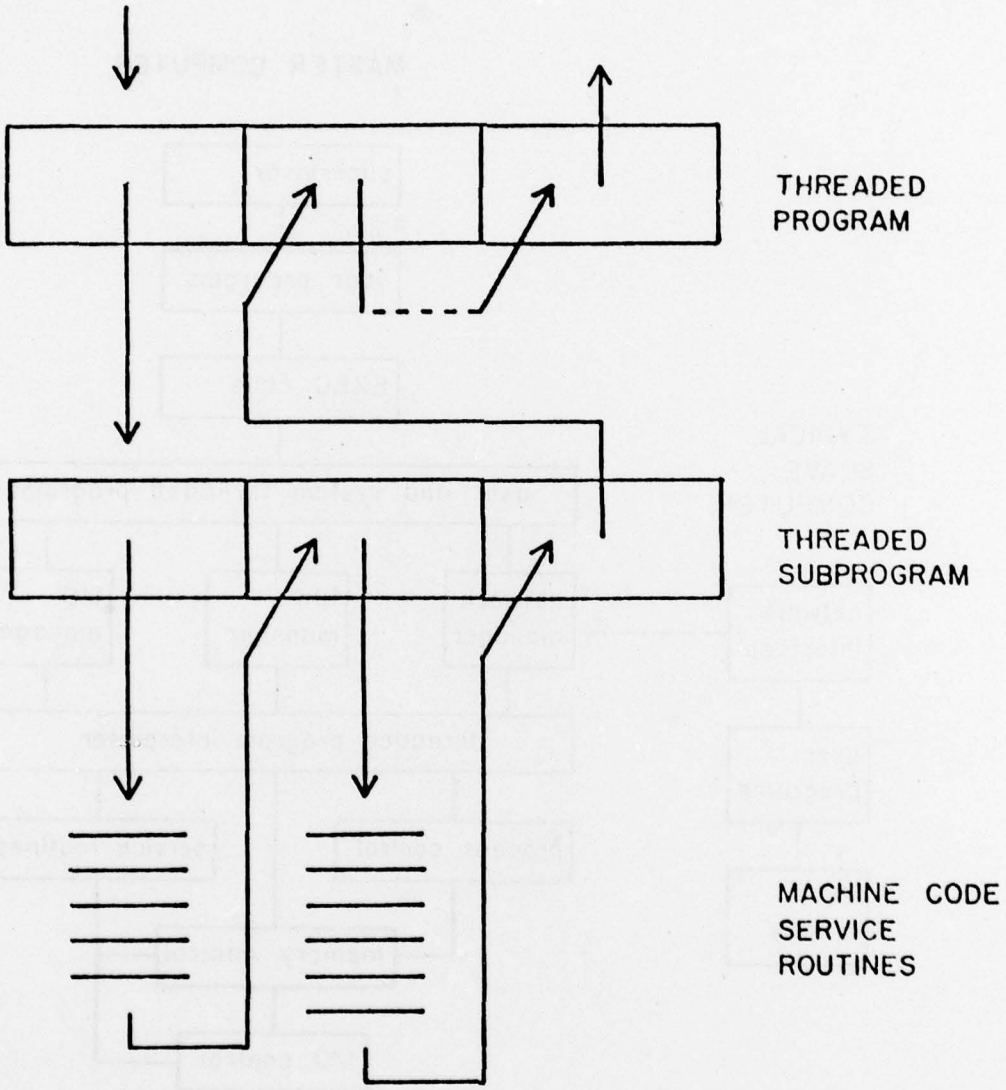
Figure Captions

1. HISS Hardware Organization
2. Simplified Schematic of the Communication Network Multiplexer
3. Structure of a Threaded Program
4. Example Threaded Programs
5. HISS Software Organization


HP 2100A
32K core,
writable
control store
line
printer
high speed
paper tape
5M byte
disc
graphics
terminal
system
teletype
communication
network
multiplexer
plotter
additional
channels
HP 2100A
8K core
HP 2115A
8K core
experiment
experiment
high speed
paper tape
high speed
paper tape
graphics
terminal
graphics
terminal
magnetic
tape


16 CHANNELS 16 BIT DATA (in)
16 x 16 Multiplexer
Data Microcircuit Card
control bit
flag
Data Output to all Channels
HP 2100 Computer
16 control lines out
16 Bit Demultiplexer
channel select
16 Bit Multiplexer
flag
Command Microcircuit Card
8 or 16 flag lines in


MASTER COMPUTER
supervisor
user programs
EXEC calls
TYPICAL
SLAVE
COMPUTER
user and system threaded programs
network
interface
network
manager
file
manager
I/O
manager
user
programs
threaded program interpreter
process control
service routines
I/O
drivers
memory control
I/O control
I/O drivers


THREADED
PROGRAM
THREADED
SUBPROGRAM
MACHINE CODE
SERVICE
ROUTINES

```
*
* HITSURF
*
* PROCEDURE TO MOVE MOLECULE TO ITS NEXT SURFACE ENCOUNTER
* THE DATA STACK CONTAINS THE CURRENT COLUMN POSITION
*
LOCAL PROC HITSURF BEGIN
HITRATE       * AVERAGE SURFACE ENCOUNTER RATE
R.EXP         * GENERATE EXPONENTIAL RANDOM NUMBER
I.+           * ADD TO COLUMN POSITION
NEXT
END
*
* STUCK
*
* PROCEDURE TO MAKE AN ADSORPTION OR NOT DECISION
* THE DATA STACK CONTAINS CURRENT COLUMN POSITION
*
LOCAL PROC STUCK BEGIN
DUP           * GET COPY OF MOLECULE'S COLUMN POSITION
TIME          * MOLECULE'S CURRENT TIME COORDINATE
DENSITY (     * GET MOLECULE DENSITY AT THIS TIME & PLACE
F.(SCALE /)   * DIVIDE BY NUMBER OF SITES PER MOLECULE
R.>U          * IS UNIFORM RANDOM NUMBER > MOLECULE DENSITY?
NEXT
END
*
* ADSORB
*
* NON LINEAR ADSORPTION PROCEDURE
*
LOCAL PROC ADSORB BEGIN
COLUMN        * PUT CURRENT COLUMN POSITION ON DATA STACK
REPEAT HITSURF * ENCOUNTER THE SURFACE
UNTIL STUCK   * TEST DENSITY & MAYBE BE ADSORBED
! COLUMN      * UPDATE MOLECULE'S COLUMN POSITION
NEXT
END
```