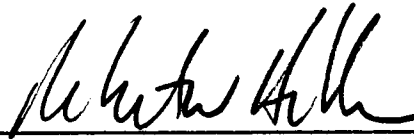


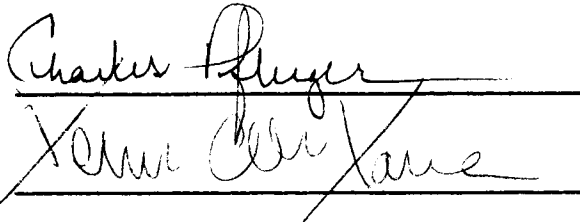
To the Graduate Council:

I am submitting herewith a thesis written by Kathryn Y. Sowder entitled "Specification for a Standard Interface Between Microcomputer Operating Systems and Applications Software." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

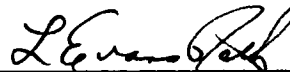


Robert W. Heller, Major Professor

We have read this thesis
and recommend its acceptance:



Accepted for the Council:



Vice Chancellor
Graduate Studies and Research

SPECIFICATION FOR A STANDARD INTERFACE BETWEEN
MICROCOMPUTER OPERATING SYSTEMS AND APPLICATIONS SOFTWARE

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Kathryn Y. Sowder

December 1981

3055503

ACKNOWLEDGEMENTS

The author wishes to thank Dr. Robert Heller for his suggestion of the topic and his guidance during its development. Thanks are also due to Dr. Charles Pfleeger and Dr. Kevin O'Kane for their assistance in critiquing the document and insuring its technical accuracy.

ABSTRACT

The purpose of this thesis is to define an interface between microcomputer-based operating systems and application programs in the areas of computer communications, computer graphics, real-time computing, language compilation, and data management. This necessitates determining the minimal functional requirements for application programs in each of these areas as well as determining how the application programs relate to one another. It is not the purpose to dictate to the designers of operating systems what the internal workings of their software should be, but only to specify a virtual operating system as it would appear to the application software with which it is to be interfaced. Such an interface, if made standard, would allow application software to be portable between different operating systems and microprocessors.

TABLE OF CONTENTS

SECTION	PAGE
I. INTRODUCTION	1
II. COMPUTER COMMUNICATIONS.	3
III. COMPUTER GRAPHICS.	9
IV. REAL-TIME COMPUTING.	16
V. LANGUAGE COMPILATION	23
VI. DATA MANAGEMENT.	28
VII. CONCLUSION	33
LIST OF REFERENCES	38
VITA	43

I. INTRODUCTION

A computer operating system is a collection of program modules which manage computer resources, such as processors, primary storage, secondary storage, I/O devices, and the file system. An operating system also serves as an interface between the user's program and the physical hardware. Microcomputer operating systems are used to increase computer productivity as are the operating systems of a large machine; however, because of the simpler design goals, the low budget market at which they are aimed, and limited primary storage, there are fundamental differences between operating systems designed for microcomputers and larger machines. A classical microcomputer is a single user machine having up to 64 KB of primary storage. It uses cassette tapes or floppy disks for secondary storage and is within the budget of hobbyists, primary and secondary educational institutions, and small businesses. This paper is aimed at this category of microcomputer, and not at the new 16 or 32 bit machines that are more expensive and would classically be referred to as minicomputers.

By reconstructing our thoughts on what program modules should be included in the operating system and also including minimum functions required for applications software, it would be possible to obtain more efficient

software production and lower software costs. Providing such support in operating systems and making the interface to it standard would prevent duplication of effort among applications software writers. They would not be required to use the functions provided, but, if chosen properly, these functions would provide a minimal system which could be used to create more sophisticated and complicated software packages. This would enable software houses to produce their products more cheaply and without commitment to only one operating system. Users would find their applications software purchases to be more portable and also cheaper.

The most commonly encountered application areas are the best candidates for merging with the operating system. The remainder of this paper will attempt to determine the minimal functional requirements for five areas requiring such support, determine if it is possible to place these functions in the operating system, and, if so, specify a standard set of system calls by which these functions would be accessed. The areas considered are computer communications, computer graphics, real-time computing, language compilation, and data management.

II. COMPUTER COMMUNICATIONS

The ability of one computer to communicate with another is highly desirable for both the largest and smallest machines. There are many specific reasons for interconnecting computers, but all fall within the general category of resource sharing or distributed computation. Resource sharing occurs when resources on one computer system are made available to or are shared with another system. Resource sharing activities include remote file access, intercomputer file transfer, and distributed database queries. Distributed computation is when cooperating programs or processes executed in different machines in a network communicate and exchange information in the performance of an overall higher task. Examples are real-time process control structures such as database computers and parallel processing.

Interconnected computers are referred to as a network. A network may be as simple as two microcomputers communicating or as complicated as several mainframes connected in a distributed topology. Obviously, there are different reasons for networking microcomputers and mainframes due to different requirements the networks must meet, as well as different economic factors. The objectives of this section are to determine what are minimum

requirements for microcomputer networking programs, determine if it is feasible to let the operating system perform these functions, and to specify an interface by which these functions included in the operating system should be accessed.

The most common reason for wanting to interconnect a microcomputer to another computer is to transfer files. As anyone can testify who has worked in an environment with several different computers running several different operating systems, it is amazing how difficult it can be to transfer a simple file from one system to another. The first case is when both computers have the same secondary storage medium and the directory structures and data formats on the media are identical. The solution to this problem is trivial. The first machine writes the file on tape, disk, or whatever the common media happens to be, and the second machine reads directly what is written. A second, more complicated, situation which is encountered frequently is having identical secondary storage media but different operating systems. In this case, there is a very good probability that the directory structures or the data formats are different. If the user is lucky, there may be with a utility program which reads the file written in the first format and writes it as a file written in the second format. Utilities such as this are very rare on microcomputers. Another possibility is to connect the microcomputers together and send the file as a serial

encoded binary stream via cable or telephone line. A third situation which also arises frequently is desiring to transfer a file from one system to another and having no common secondary storage media. In this case, computer to computer communication is the only practical possibility for copying the file if it is large.

File transfer may be desired between microcomputers or between a microcomputer and a large computer. In addition to simply desiring a program which is located on a large machine, there are other reasons to connect a microcomputer to a large machine. Often times a program will go through the initial debugging process on a simulator on a large machine in a multiprogramming environment. After no errors are detected on the simulator, the object code is downloaded to the microcomputer. On occasion it is desirable to transfer data from a microcomputer to a large machine. Because of the limited computing capability of microcomputers, data is often collected by the microcomputer and then transferred to a large machine for processing. Letting the large machine do the number crunching is a form of distributed processing and makes optimum use of all computer resources.

At this point we must examine the practical considerations in dealing with microcomputers and what constraints these place on a design. The first limitation is secondary storage. Software is generally distributed on a floppy disk. An 8 inch single sided, single density, soft

sectored disk can store approximately 256 KB. A 5 inch minifloppy, even when double density can store even less. These are the most common types of secondary storage media. It is true that single sided double density disks are being seen more often and even double sided, double density floppies are available, but it would be very unrealistic to limit a software package to such systems. A second problem is a limitation on the amount of primary storage available. The new 32 bit microprocessor systems have a larger addressing range, but most systems are limited to 64 KB. Overlaying is always a possibility, but it is desirable for all program modules to be resident in primary storage at once. Cost is a third problem. Microcomputers are inexpensive. It makes little sense to solve a problem by means of a solution which costs more than the microcomputer. For example, in the long-lines ARPA network which connects several mainframes in different parts of the United States, all communications to or from a host (mainframe) go through an interface message processor (IMP). An IMP is a minicomputer which serves as a switching element to interface a host into the communication network. IMPs also provide standard protocol for the network as well as a communication control subnet through which all mainframes communicate. Obviously, such a network design is not practical for microcomputers. The minicomputer costs many times more than the microcomputers. It would be bad economics to use support hardware that costs more than the

main machine. Another problem is how "smart" a communications program should be. Should it be programmed to do such things as enter account numbers and passwords? Many communications programs are designed to do such things; however, often problems result from such a design. For example, a program which logs-on automatically must be written to look for a very specific log-on prompt and send the response in a very specific format. If the log-on procedure is changed in anyway, the program will not work, and the user will be sitting there wondering why. Eventually, if it is available, the source code must be modified to fix the problem. It is much more sensible to let the user of the program provide the intelligence to cope with such changes. One last problem is how the communications program should deal with errors in transmission. It is feasible to have error detection and correction for transfers between microcomputers; however, when a large machine is involved, this is not the case. This, coupled with our design goal of minimal requirements, leads to the conclusion that error detection must be left to the human element of the system, the user. The user must verify if the transfer was successful either by visually examining the file or by determining if the transferred file executes properly. It is the user's responsibility to determine such things as whether to repeat the transfer when an error occurs.

In summary, the functions which the operating system

should provide and the interface to these functions are follows: the system call for a transfer will be `trans(destination file, source file)`. It will be the user's responsibility to perform such system dependent tasks as log-on. It is also the user's responsibility to verify that the transfer has been accomplished successfully and to respond to any abnormal situations which occur. The transfer may be terminated prior to completion by entering a control c, the end of text signal (ETX), from either the terminal associated with the originating or destination machine.

III. COMPUTER GRAPHICS

Computer graphics is an area which has experienced rapid growth in recent years. Until recently, the only microcomputer graphics systems available were very primitive. At this lowest level of the graphics hierarchy, the most significant bit of the byte containing an ASCII character is used to determine which part of the dot matrix in which character is embedded will be displayed. Individual pixels are not addressable. The programmer must determine for each line of the display where in that line a character should be placed and select a character or portion of a character to use in this position to create the desired picture. Clearly, The picture created is of very poor quality and there is no option for color. The only advantage to this method is that no additional memory is required.

As the cost of primary storage has plummeted, the sophistication of microcomputer graphics systems has risen due to the economic feasibility of fully addressible pixels. Graphic systems of this type are referred to as raster scan graphics. In raster scan graphics, the image is stored in a buffer in memory called a frame buffer. The frame buffer serves as a matrix where each element of the matrix represents the color of one pixel of the picture. The

greater the number of colors and shades of colors which are possible, the greater the number of bits per pixel which must be used to encode the information. Also, the greater the resolution, the greater the number of pixels required. Both the number of pixels and bits per pixel affect the amount of memory required. Today's typical low resolution raster scan color graphics systems require an 8 KB frame buffer. At yesterday's prices, this much memory would have been prohibitively expensive, but today it is within the budget of many small system owners.

A third echelon of small computer graphics is raster scan systems with analog input such as joysticks. These devices are used to position the cursor anywhere on the screen by simply moving a control lever. The visual feedback provided by the human eye allows the user to compensate for any nonlinearities in the joystick and to control the cursor with great precision.

The highest graphics echelon adds a graphics tablet to the system. A tablet provides an x-y coordinate system from which to enter data or commands quickly and easily. When the stylus or cross-hair touches the tablet, it senses its coordinate on the tablet. The coordinate may represent a point on a map or another item from which data is being input, or it may be a command to the computer via a menu.

Scientists, businessmen, educators, and artists, as well as hobbyists, are making use of microcomputer graphics. Part of the reason for this is that a picture conveys more

information, more quickly, than either a verbal or written explanation. This has been known for many years as shown in the old adage, "a picture is worth a thousand words", shows. Today's graphics tools can be used to create visual aids quickly and easily. In addition, people who enjoy fine arts can create pictures that are both beautiful and creative. Lastly, computer graphics can provide many hours of entertainment in the form of computer games.

Microcomputer graphics differs from large machine graphics in several ways. In the small computer world, storage scopes are almost never used. There are two reasons. First, storage scopes are expensive, and anything which is expensive is almost always excluded from microcomputer systems. Second, although the ability to store lines is a convenience in static displays, it makes animation impossible, and animation is very common in microcomputer graphics. The bright green flash which occurs when a storage scope is erased is also undesirable in animated programs.

A second way in which large machines and microcomputers differ is in their use of animation. Because microcomputer graphics are often developed for educational and recreational purposes, animation is often employed. As Saturday morning cartoons illustrate, animated drawings capture a child's attention. Plus, it is possible to illustrate many ideas much more effectively with movement than with static drawings. Programs developed for

recreational purposes are much more fun and challenging with animation. Because of the expense involved, large machines are generally used by the government and large businesses. Their graphics needs are very different from those of education or recreation. They use graphics for such things as attractive reports and visual aids; machine time is too expensive for fun and games.

Another way in which large and small machines differ is the degree of interactivity between the computer and the user. The majority of the time graphics programs for microcomputers are very interactive; however, on large machines, interactive programs are not as common.

Lastly, large machine graphics packages tend to have many more "bells and whistles" such as providing subroutine calls to draw special items. On small machines most of the time the users must write their own special purpose items.

With the price of memory so low and dropping even lower, it would be ridiculous to consider developing a graphics system which does not use a frame buffer. Raster graphics opens many avenues which are impossible with other types of systems. The user of a raster system is able to create animated pictures by repeatedly displaying and altering the buffer. Pictures of better quality are possible because of the addressibility of each individual pixel. The positive points of raster scan graphics systems far out-number the negative points.

Although graphics are done in black and white, a high

resolution color system with a black and white option is the best choice for microcomputer graphics. With a color system, both color and black and white drawings are possible. The package should also include provisions for some means of analog input such as joy sticks or paddles. A tablet is nice but should be considered an extra.

To develop any graphics package, certain primitives are required. The package must provide subroutines both to move the cursor and draw lines and points at absolute locations and locations expressed relative to the present position of the cursor. In addition, a subroutine call must be available to initialize the system by performing such tasks as clearing the frame buffer and display screen. Separate subroutines are also needed for clearing the frame buffer and display screen. A bell should be provided for audible feedback. Lastly, alphanumeric lettering is required. These are the primitives which are absolutely required in a graphics system for microcomputers.

A slightly more sophisticated package is recommended. Such a system allows the user to perform rotation and windowing transformations. These add considerably more power and versatility to the package. If these are not provided, serious developers of graphics software in any area will have to write their own.

Because graphics software is used with many different types of hardware, a standard interface between the graphics hardware and the applications software is especially

important. By providing standard system calls for the primitives required in any graphics software package and letting the problems generated by the variances in hardware be resolved in the device drivers and system generation procedures, powerful graphics software packages can be created more quickly and easily.

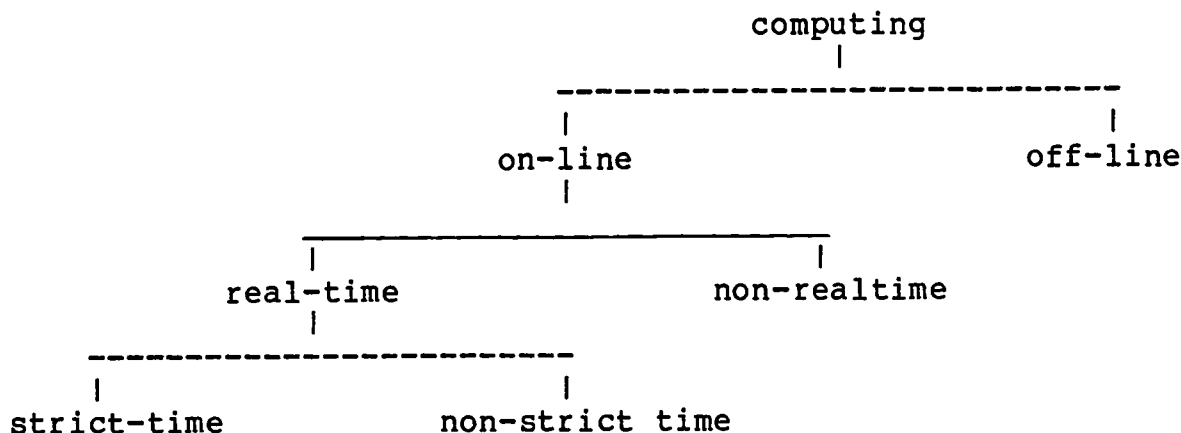
In summary, the graphics primitives which the operating system should provide and the interface to these functions are as follows: `movrel(x-displacement, y-displacement)` moves the cursor relative to its present position. `Movabs(x-coordinate, y-coordinate)` moves the cursor to an absolute location. `Pntrel(x-displacement, y-displacement)` draws a point relative to the current cursor position. `Pntabs(x-coordinate, y-coordinate)` draws a point at an absolute location. `Drwrel(x-displacement, y-displacement)` draws a line relative to the current cursor position. `Drwabs(x-coordinate, y-coordinate)` draws a line from the current cursor position to an absolute location. `Rotate(direction, degrees)` will display a rotated version of the object stored in the frame buffer. `Window(x-minimum, x-length, y-minimum, y-length)` will display the specified portion of the frame buffer. `Clrscr` will clear the screen. `Clrfrm` will clear the frame buffer. `Bell` will cause the bell to ring to provide audible feedback. `Init` will cause such functions as clearing the screen and frame buffer and positioning the cursor at the first element of the array to be performed. `Letter('string')` causes the specified string

of characters to be displayed beginning at the current cursor position.

IV. REAL-TIME COMPUTING

Real-time computing occurs when a computer's principal task is to monitor and/or control a time-critical process whose own logical sequence is asynchronous to that of the computer itself. The criticality of the process may be due to the rate of external data generation or because the response of the computer is one of the parameters of a closed-loop system. A simple test for determining if a system is real-time is to imagine the system clock gradually being slowed down; if at some stage this would cause a system break-down, the system is operating in real-time.

There are two classes of real-time, strict-time and nonstrict-time. Strict-time is when a process includes a direct or indirect reference to the value of time occurring in the external world. It includes such things as the digital filtering of speech signals and averaging the impulse responses of a system. Nonstrict-time has no reference to time. An example is pulse height analysis of signals from a nuclear radiation detector. Strict-time is generally the rule, and nonstrict-time is the exception. The relationships of the various classes of computing are shown in the following illustration.



Time is the most important factor in real-time computing and tends to determine the nature of it. When a real-time system is designed, there is a total preoccupation with time in both the hardware and the software. In most cases, the system must be fast. To facilitate this, complicated operations which occur frequently such as multiplication, division, and floating point operations are implemented in hardware. Software used to control the system must be efficient and is either totally written in assembler language, or, at the very least, the critical portions of the code which actually drive the I/O devices are written in assembler. I/O facilities are critical and unique for each system. They must be designed to be fast, reliable, and easy to use. The techniques used to process the signals must, on an average, be the best available for the particular problem.

The external devices which real-time systems monitor or control are many and varied. Because of this, there are few "off the shelf" real-time systems. The designers of each

system must determine what is required to service that device and then integrate unique hardware and software to meet these requirements.

All real-time systems fall within two general categories. The first category includes those systems which just collect data. No processing of the data is done. At a specified time or at a specified time interval, a sample is taken and nothing more. The second category includes those systems which collect data, process it, and use the results obtained to determine the future actions of the system.

A factor which is common to all real-time systems is the program interrupt. Program interrupts are asynchronous to program execution and are used to signal the processor that an external device needs attention immediately. When an interrupt occurs, the computer temporarily stops executing the current sequence of instructions and begins executing a new sequence of instructions, the interrupt service routine which services the device.

There are two classes of program interrupts used in real-time programming. These are nonmaskable interrupts and maskable interrupts. A nonmaskable interrupt is an interrupt which cannot be inhibited by the programmer. This type of interrupt is used for emergencies and lacks flexibility. Maskable interrupts, unlike nonmaskable interrupts, may be inhibited by the programmer. The priority is not as high as with the nonmaskable interrupt, but the sequence of events which occurs is identical for both.

When an interrupt occurs, hardware automatically saves the contents of the current program counter and program status word and places the address of the interrupt service routine, which may be obtained directly or from a vector table, in the program counter. Upon completing execution of the interrupt service routine, a return from interrupt instruction is executed, hardware restores the program counter and program status word, and execution is resumed as if the interrupt had not occurred.

Computation is required to process data which has been collected. An essential ingredient of any real-time system which is to perform any task more complicated than just collecting data is a mathematical library. The trigonometric functions sine, cosine, tangent, and their inverses must be available as well as exponentiation, natural and common logarithms, absolute values, and square roots. The capability must exist to transfer between fixed point and floating point representations of numbers. Hyperbolic functions are not used as frequently as the trigonometric functions and can be derived. Facilities for working with complex numbers must also be provided. Because 8 bit accuracy is not sufficient, 32 bit accuracy should be given for normal calculations and 64 bit accuracy for double precision calculations. It is assumed multiplication, division, and floating point operations will be implemented in hardware.

Because of the limitation on space and the wide variety of numerical techniques which are appropriate for analysis of certain categories of data but not others, routines for performing numerical analysis should not be provided. It should be the responsibility of the individual experimenter to determine the best numerical technique for use in a particular experiment and implement it in code.

Provisions for buffering of data must be made to compensate for differences in the rate at which the data is input and the rate at which the data is processed. The rates must still match on an average, but instantaneous matching is not required. Ideally, each I/O channel should have its own buffer area.

In the processing of signals, there are two basic domains in which a signal may be represented, the time-domain and the frequency-domain. In almost every situation involving signal processing the initial representation of the signal is as a continuously varying pattern. Mathmatically, such patterns are represented as continuous fuctions of time. For many operations, such as convolution and filtering, which are performed on a signal, it is desirable to model the signal mathematically as continuous functions of frequency. To perform this conversion from the time-domain representation to the frequency-domain representation, one applies the Fourier transform. The inverse Fourier transform is applied to return to the time-domain from the frequency-domain. In the

Laplace transform, the Fourier transform is extended to include complex frequencies. Work using complex frequencies is done to allow the study of the natural and forced responses of a system together, to overcome certain convergence problems so that certain functions such as the step function may be included, and to take advantage of the complex plane as a vehicle for analysis and synthesis of systems.

With digital processing of signals, representations of discrete signals must be used. The z-transform is a sampled version of the Laplace transform. The discrete Fourier transform is a version of the z-transform of a finite length sequence obtained by sampling along the unit circle. It has properties very similar to many of the useful properties of the z-transform and Fourier transform. It is used with discrete signals for computing spectrum estimates, correlation functions, and implementing digital filters.

A computing algorithm called the fast Fourier transform (FFT) is used to calculate the discrete Fourier transform of a signal on a computer. The algorithm is of order $N \log(N)$. Prior to the development of the FFT, the algorithm used was of order N^2 . Needless to say this is a significant savings in computer time. To return to the time-domain from the frequency-domain, the FFT is simply applied again. This algorithm has had a tremendous impact on the digital signal processing field and should definitely be included in the operating system for real-time support.

To summarize, the minimal real-time functions which the operating system should provide are as follows: all calculations will be done to 32 bit accuracy unless double precision is requested, in which case 64 bit accuracy will be given. Double precision is specified by stating double precision followed by the variable names in the program declarations. System calls will be provided for the sine, cosine, and tangent trigonometric functions as well as their inverses. System calls will also be provided for natural and common logarithms, absolute values, exponentials, square roots, fix, and float. Calculations with complex numbers will also be possible in the baseline system. These will require functions for taking the complex conjugate, the real portion, the imaginary portion, and the magnitude of a complex number. Because Fortran has been the programming language for scientific computation, all system calls will be expressed as they would be in standard Fortran. A good fast Fourier transform program will be included for converting to and from the frequency domain. The FFT program will be invoked by `fft(array,size)` where `array` is the 1-dimensional array to be transformed and `size` is the maximum dimension of this array.

V. LANGUAGE COMPILATION

Before a program can be executed by a computer, the program must be represented in machine language. If the program was not written in machine language, and very few are, it must be translated. Assembler language programs use mnemonics for the numeric codes that the machine understands. Because there is a one-to-one correspondence between assembly and machine language instructions, it is very easy to translate assembler programs into machine language. Interpreters and compilers are used to translate higher level languages into machine language.

In an interpreted language, each instruction corresponds to a machine language subroutine. Operation of the interpreter is as follows. The next source instruction is fetched. The statement counter is advanced. The statement is analyzed to determine what action is to be performed upon what operands, and the statement is executed by calling a machine language subroutine which performs the specified action. Interpretation generates no object module. Every time a statement is executed it must be interpreted again.

Compilers, unlike interpreters, generate an object module which may be saved and executed many times. Compiling and executing a program take more time than it takes to interpret an equivalent program; however, once the

object module has been created, executing it is much faster than interpretation. Storage requirements must also be considered in deciding whether to use an interpreter or a compiler. An interpreter is a smaller program than a compiler, but an object module requires less space than source code. Compilers are used more frequently than interpreters, but it is important to understand the tradeoffs involved in choosing one over the other. This paper is concerned with compilers because they provide more opportunity for operating system support.

Compilation is a complicated process because there is not a simple mapping from source code into object code. The entire process requires lexical analysis, syntactical analysis, intermediate code generation, and error recovery routines. If it is an optimizing compiler, optimization is performed on the intermediate code. Lexical analysis is the process of breaking the source code up into its smallest logical units and determining each one's type. These smallest logical units are called tokens. Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. Syntactic analysis of the tokens determines if a string of tokens complies with the grammatical rules of the language. This acceptance or rejection of a string as a member of the language is called recognition. Syntactic processing is also involved when the structure of the program is analysed to determine if it is allowed by the language. The

structure of the language includes such things as nesting of parentheses and end statements on if-then-else constructs. The structure is analysed by applying the productions of the language to build a parse tree. Syntactic analysis is often referred to as parsing. Both bottom-up and top-down techniques are used in parsing. Top-down techniques involve building a parse tree beginning with a special symbol at the root of the tree and proceeding downward to a string of terminals at the leaves. Bottom-up techniques begin with a string of terminals at the leaves and proceed upward to the root. As it builds a parse tree, the compiler generates for the program. Intermediate code is a pseudomachine language representation of the source code. It is independent of the details of the target machine but reflects the types of operations which are possible on the machine. The intermediate code is translated into machine code. This translation may be direct or may involve translating the intermediate code into assembler language and then the assembler translating it into machine code. Naturally, good error diagnostics are a requirement of any compiler.

There are two general types of compilers, batch and incremental. Batch compilers make no provisions for user interaction during the compilation process. Although it may seem a contradiction in terms, batch compilers are often used in environments that are otherwise interactive. An incremental compiler is an interactive compiler in which special facilities are available to permit the modification,

execution, and checkout of the program during compilation. The editor is an integral part of such compilers. As the user is entering a program, if a statement has a syntax error, an error message is generated immediately rather than after the entire program has been entered and the compiler invoked. Generally, incremental compilers include a preexecution analysis of the program as a whole to determine if there are context dependent errors once all of the syntactic errors have been detected on a line-by-line basis. Any time an error is encountered, the programmer is notified and given the options of continuing or correcting the error. When a source statement is changed, the entire program does not have to be recompiled. An incremental compiler is similar to an interpreter but has the important advantage of speed.

There are many ways to implement the functions required in a compiler. This is especially true of the numerous parsing algorithms available. In addition, every computer language is described by a different grammar and a special set of keywords. Because of these factors, it is impossible for an operating system to include a standard parser for all compilers. However, some operations that are commonly required in compiler construction, such as stack manipulation and table building, could be supported. The system call to build a stack would be `stack(stack-name, size)`. To push and pop from a stack the calls would be `push(stack-name)` and `pop(stack-name)`. To create a table,

the call would be `table(table-name, x-dimension, y-dimension)`. To insert and delete entries from a table, the calls would be `insert(table, table-name, x-position, y-position, entry)` and `delete(table, table-name, x-position, y-position)`.

In areas related to the compilation process such as linkage conventions where it would be extremely advantageous if conventions were the same for all machines, standardization seems impossible. The terms applied to equivalent operations under different operating systems could and should be the same. For example, linking and task building are the same operations, and using different terminology only serves to confuse the naive user.

VI. DATA MANAGEMENT

A data management system is at the middle of a three layer hierarchy composed of the file system, data management systems, and database management systems. File systems are at the lowest level and are concerned only with being able to locate files on a volume. The files are not structured into any logical organization or interpreted in any way. Data management systems build on the file system and impart a logical structure on the files in the file system but perform no interpretation of these files. Database management systems are concerned with both the structure and interpretation of the file system. Data management systems and database management systems provide facilities for ease of interaction with a file system. They are also used to simplify many of the housekeeping chores associated with the creation and maintenance of files.

Database systems on large machines are very sophisticated programs. Records may be stored using multiple keys to allow users to reference entries according to more than one category. Calculations are performed on raw data in the database to obtain such things as a client's age or the number of seats remaining on a flight. The occurrence of an operation on the database often triggers another event such as the generation of a report. Users may have different views of the database depending on their need to

know. These capabilities along with other capabilities which may be available make database programs very large and complicated. Database designers are faced with many difficult problems such as ensuring the integrity of the database under all circumstances and making sure all pointers to a multiple keyed record are deleted when the entry is deleted.

Data management systems are not as sophisticated as database systems. They are concerned only with providing ways to organize and access the records in files. Generally, they provide facilities to organize records sequentially, by an index, or relatively. Sequential organization is the best for files requiring frequent access. A relative organization is best for files requiring infrequent access. An organization based on an index is chosen for files requiring an intermediate amount of access or as a compromise in situations where sequential or relative organization would be best. Once records have been stored in a file, the user must be able to access those records. The two most common access methods are sequential access and random access. Sequential access may be used with any file organization; however, random access can only be used with a file organized relatively and stored on a direct access device. In addition to selecting the best organization and access method for the data, the user also has the option of performing the buffering or allowing the operating system to provide this service.

As with any other microcomputer applications area, data management has constraints placed on it by the limited amount of memory. The small amount of primary storage available will not let a data management system be all things to all users. This is particularly important in an area like data management where programs can become large very quickly if they are expanded into the realm of database management or if many options are included in the program. If the user has a special requirement, the user should be responsible for incorporating it into the system rather than letting it be a part of the baseline system.

Because of the different requirements which are placed on a data management system, it is felt that it would be inadequate for a data management system for microcomputers to provide only sequential organization of and access to the records in a file. The user should be able to organize his files sequentially, by an index, or relatively, just as is possible with data management systems on larger machines. The user should be able to access files both sequentially and randomly. These requirements are not intended to imply that all sorts of variations and expanded versions of these organizations and access methods should be included. The most basic implementations are all that should be provided. Microcomputer systems do not have either the primary or secondary storage to provide the all encompassing packages that are available on the large machines.

Since there is generally no requirement to read or write tapes which will be used with large machines, fixed block size should be used in transfers. Fixed size transfers simplify buffering requirements greatly. Many access methods on large machines give the user the choice of providing buffering himself or allowing an operating system to provide it. Since the operating system must be created with a certain amount of pool space reserved to it, and since this pool space is available regardless of whether the user performs the buffering, it is a more efficient use of space if the operating system performs the function. Since all blocks in a transfer will be the same length, the operating system will simply set up a buffer equal to the block size.

It is assumed that overlay capability will be available through the operating system. Because file organizations and access methods are mutually exclusive, overlaying will minimize the amount of primary storage required.

In summary, the support which microcomputer operating systems should provide to data management systems consists of the following: Sequential, indexed, and relative file organization and sequential and random access to records stored in these files should be provided. Fixed length blocks will be used in I/O transfers. The operating system will provide the buffering of data during transfers.

Sequential files will be created by the call `make(seq, filename, record-size)`. Indexed files will be created by

the call `make(indexed, filename, record-size, index-size)`. Relative files will be created by the call `make(rel, filename, record-size, first-num, last-num)`. The function to access a file sequentially to retrieve a record will be called by `read(seq, filename, record-name)`. The function call to access a file sequentially and insert a record will be `write(seq, filename, record-name)`. To insert a record into an indexed file where the index already exists, the user issues the call `write(indexed, filename, index, record-name)`. To fetch a record from an indexed file, the call will be `read(indexed, filename, index, record-name)`. To insert a new index into an indexed file, the call will be `insert(indexed, filename, index-name)`. To delete a key from an indexed file, the call will be `delete(indexed, filename, index-name)`. To fetch a record from a relative file, the call will be `read(rel, filename, record-number)`. To write a record into a relative file, the call will be `write(rel, filename, record-number)`.

VII. CONCLUSION

This paper has been an attempt to determine the minimal functional requirement of five areas of applications software, determine the feasibility of incorporating these into an operating system, and specify a standard interface through which these functions could be accessed by applications programs. The applications areas analyzed were computer communications, computer graphics, real-time computing, language compilation, and data management. Analysis shows that, with the exception of language compilation, all software packages in an applications area are based on a nucleus of common functions. This commonality makes it advantageous to have only one set of these functions and make them available to all via standard system calls. The ideal location for these routines would be in the operating system. Such an approach would save the time required to create duplicate software, save the space required to store multiple programs which perform the same function, and allow applications software to be portable among all operating systems which comply with the specified standard. Such an approach does not dictate to operating systems designers the internal implementation of their product but simply specifies the virtual image which their product will display to the applications software.

For the computer communications area, a routine is required to perform a simple file transfer. No error detection or correction is required, the user should not be given any assistance with such things as log-on and abnormal situations. The system call to invoke this routine would be `trans(destination file, source file)`.

In all computer graphics software packages, functions must be available to move the cursor absolutely and relatively, to draw points absolutely and relatively, and to draw lines absolutely and relatively. The frequency with which objects must be rotated makes a rotational transformation a requirement. Often times it is desired to examine only a portion of an object; therefore, windowing is also needed.

It is also often necessary to clear the screen, clear the frame buffer, and initialize the system. Audible feedback is used in many types of programs. Lastly, graphics packages require alphanumeric characters for prompts and labeling. The standard set of system calls for invoking these routines would be as follows: `movrel(x-displacement, y-displacement)`, moves the cursor relative to its current position; `movabs(x-coordinate, y-coordinate)`, moves the cursor to an absolute location; `pntrel(x-displacement, y-displacement)`, draws a point relative to the current cursor position; `pntabs(x-coordinate, y-coordinate)`, draws a point at an absolute location; `drwrel(x-displacement, y-displacement)`,

draws a line relative to the current cursor position; `drwabs(x-coordinate, y-coordinate)`, draws a line from the current cursor position to an absolute location; `rotate(direction, degree)`, rotates the object; `window(x-minimum, x-length, y-minimum, y-length)`, displays only the specified portion of the object; `clrscr`, clears the screen; `clrfrm`, clears the frame buffer; `bell`, rings the bell; `init`, initializes the system; and `letter('string')`, draws the alphanumeric string on the display.

Operating systems will provide language compilation programs support in the areas of stack and table manipulations. The system call to create a stack will be `stack(stack-name, size)`. To push an entry onto the stack, the call will be `push(stack-name)`. To pop the stack, the call will be `pop(stack-name)`. A table will be created by the call `table(table-name, x-dimension, y-dimension)`. An item will be stored in a table by the call `insert(table, table-name, x-position, y-position, entry)`. An entry will be deleted from a table by the call `delete(table, table-name, x-position, y-position)`.

Real-time computing requires trigonometric functions, common and natural logarithms, absolute values, exponentials, and square root. Facilities must be provided for working with complex numbers. These should include functions for taking the conjugate, imaginary portion, real portion, and magnitude of an imaginary number. No less than

32 bits of accuracy is sufficient for normal calculations. Twice this number is required for double precision calculations. It must be possible to represent numbers in fixed point and floating point notation. All system calls for these functions should use standard Fortran notation since it is so engrained in scientific computing and will allow programs already written to continue being used. A fast Fourier transform is required to convert between the time and frequency domains. The FFT routine will be invoked by `fft(array,size)` where `array` is a one dimensional array and `size` is its maximum dimension.

Data management applications software should be provided the primitives to create sequential, indexed, and relative file organizations. Functions should be provided to access these files either sequentially or randomly. It will be standard for the operating system to perform the buffering required in I/O transfers. Each files will contain only fixed length records. The following calls will be used to create sequential, indexed, and relative files respectively: `make(seq, filename, record-size)`, `make(indexed, filename, record-size, index-size)`, and `make(rel, filename, record-size, first-num, last-num)`. The functions to read and write to a sequential file are `read(seq, filename, record-name)` and `write(seq, filename, record-name)`. To insert and delete a record from an index file in which the index already exists, the calls will be `read(indexed, filename, index, record-name)` and

write(indexed, filename, index, record-name). The call to create an index in a file will be insert(indexed, filename, index-name). The call to delete an index from a file will be delete(indexed, filename, index-name). To randomly access a relative file, the calls will be read(rel, filename, record-number) and write(rel, filename, record-number).

LIST OF REFERENCES

LIST OF REFERENCES

1. Aho, A. and Ullman, J. Principles of Compiler Design, Addison-Wesley, 1977.
2. Akkoyunlu, E., et al. "Interprocess Communication Facilities for Network Operating Systems," Computer, June 1974, pp. 46-56.
3. Allan, R. "Local-Net Architecture Protocol Issues Heating UP," Electronic Design, April 16, 1981, pp. 91-102.
4. Ashenhurst, R. "A System for Interprocess Communication in a Resource Sharing Computer Network," Communications of the ACM, April 1972, pp. 221-230.
5. Barnes, R. PL/I for Programmers, North Holland, 1980.
6. Blackman, M. Design of Real Time Applications, John Wiley, 1975.
7. Bowman, W. and Flegal, B. "Toolbox: A Smalltalk Illustration System," Byte, August 1981, pp. 369-376.
8. Brignell, J. and Rhodes, G. Laboratory On-Line Computing, John Wiley, 1975.
9. Brown, P. Writing Interactive Compilers and Interpreters, John Wiley, 19710.
10. Calingaert, P. Assemblers, Compilers, and Program Translation, Computer Science Press, 1979.
11. Carr, S. et al. "Host-to-Host Communication Protocol in the ARPA Network," AFIPS Conference Proceedings, 1970, pp. 589-597.
12. Day, J. "Resource Sharing Protocols," Computer, September 1979, pp. 47-56.
13. Digital Equipment Corp., Introduction to Minicomputer Networks, 1974.

14. Digital Equipment Corp., RMS-11 User's Guide, 1979.
15. Folts, H. "Status Report on New Standards for DTE/DCE Interface Protocols," Computer, September 1979, pp. 12-19.
16. Forsdick, H. et al. "Operating Systems for Computer Networks," Computer, January 1978, pp. 48-57.
17. Goldberger, A. "A Designer's Review of Data Communications," Computer, May 1981, pp. 103-112.
18. Gries, D. Compiler Construction for Digital Computers, John Wiley, 1971.
19. Hannula, R. System 360/370 Job Control Language and the Access Methods, Addison-Wesley, 1977.
20. Heart, F. et al. "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, 1970, pp. 551-567.
21. Hughes, J. PL/I Structured Programming, John Wiley, 1979.
22. Hutt, A. A Relational Data Base Management System, John-Wiley, 1979.
23. Ingalls, D. "The Smalltalk Graphics Kernel," Byte, August 1981, pp. 168-194.
24. Kimbleton, S. et al. "A Perspective on Network Operating Systems," AFIPS Conference Proceedings, 1976, pp. 551-559.
25. Kuo, F. Computer-Communication Networks, Prentice-Hall, 1973.
26. Levy, W. "Packet-Switching Systems Go Private," Mini-Micro Systems, March 1981, pp. 87-96.
27. McKenzie, A. "Some Computer Network Interconnection Issues," AFIPS Conference Proceedings, 1974, pp. 857-859.
28. McLeod, J. "Local-Net Terminals Take on New Look For New Functions," Electronic Design, April 16, 1981, pp. 133-140.

29. Metcalf, R. "Strategies for Operating Systems in Computer Networks," Proceedings of the ACM Annual Conference, 1972, pp. 278-281.
30. Newman, W. and Sproull, R. Principles of Interactive Computer Graphics, McGraw-Hill, 1979.
31. Pollack, B. Compiler Techniques, Auerbach, 1972.
32. Rabiner, L. and Schafer, R. Digital Processing of Speech Signals, Prentice-Hall, 1978.
33. Retz, D. "Operating System Design Consideration for the Packet-Switching Environment," AFIPS Conference Proceedings, 1975, pp. 155-160.
34. Schinder, M. "Networks May Look Alike But Software Makes the Difference," Electronic Design, April 16, 1981, pp. 121-126.
35. Schneider, G. M. "Computer Network Protocols: A Hierarchical Viewpoint," Computer, September 1979, pp. 8-10.
36. Soi, I. and Aggarwal, K. "A Review of Computer Communications Network Classification Schemens," IEEE Communications Magazine, March 1981, pp. 24-32.
37. Stonebraker, M. "Operating System Support for Database Management," Communications of the ACM, July 1981, pp. 412-418.
38. Tektronix, Corp., Tektronix Plot 10 Terminal Control System Users' Manual, January 21, 1978.
39. Turner, R. Real-Time Programming with Microcomputers, Lexington Books, 1978.
40. Ullman, J. Principles of Database Systems, Computer Science Press, 1980.
41. Walden, D. and McKenzie, A. "The Evolution of Host-to-Host Protocol Technology," Computer, September 1979, pp. 29-38.
42. Wecker, S. "Computer Network Architectures," Computer, September 1979, pp. 58-72.

43. Yencharis, L. "Local-Netcommunications Improve at Both Ends," Electronic Design, April 16, 1981, pp. 111-117.

VITA

Kathryn Sowder was born in Knoxville, Tennessee, on November 3, 1950 and attended primary and secondary schools in the Knoxville area. Upon graduation from high school, she enrolled at The University of Tennessee. In August 1973, she received a Bachelor of Science degree in Electrical Engineering from that institution.

Upon her graduation, she was commissioned through ROTC as a Second Lieutenant in the United States Air Force. From December 1973 until June 1974, she attended Communications-Electronics Engineer course at Keesler Air Force Base in Biloxi, Mississippi. After completion of the course, she was assigned to Richards-Gebaur Air Force Base at Kansas City, Missouri. Her duties there included the design and installation of digital microwave communication systems and the evaluation and analysis of the ground equipment and radiation patterns of TACAN, ILS, and VOR aircraft navigational aids. She resigned her commission in November 1977.

In 1978 she returned to Knoxville to attend graduate school at The University of Tennessee. While in graduate school, she was employed part-time by the Department of Electrical Engineering as a graduate research assistant in the field of computer generation and recognition of speech and by the Computer Science Department in the department's minicomputer laboratory. In December, 1981, she received the Master of Science degree in Computer Science.