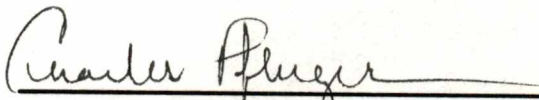
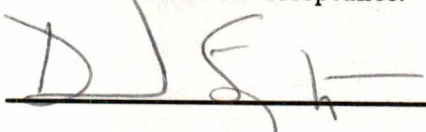


To the Graduate Council:

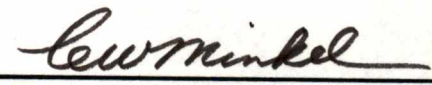
I am submitting herewith a thesis written by David L. Russell entitled "I/O Facilities for SIMON, a VAX Operating System Simulator." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.


Charles P. Pfleeger, Major Professor

We have read this thesis
and recommend its acceptance:


Maria Zemanova

Accepted for the Council:


Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature 

Date November 26, 1986

I/O FACILITIES FOR SIMON, A VAX OPERATING SYSTEM SIMULATOR

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

David L. Russell

December 1986

For G.W.B.

ACKNOWLEDGEMENTS

If there are any bugs left in the I/O portion of SIMON, I am responsible for them. If perchance there are any pleasing aspects to the code or clarity in the explication, I am obliged to share the credit.

Chuck Pfleeger was the driving force behind the entire SIMON project; his knowledge of systems software development and his pedagogical wisdom were invaluable. I cannot imagine a better teacher of computer science.

Thanks go to Beverly Zygmunt (although she abhors acknowledgements), who first suggested I design the I/O facilities for SIMON; her work, and that of Phyllis Marlino, laid a solid foundation for the project's completion. Thanks also to David Straight and Maria Zemankova, who made my oral defense an enjoyable, rather than an excruciating, experience; to Vivien Goff, who managed the VAX system professionally; and to the team of Ethel Wittenberg, Suzie Witenbarger, Don Reagan and Dorsey Bottoms, who have assisted us all immeasurably.

If Wallace Mayo had contributed only her sense of humor, I would still have been overwhelmed; as it was, she also greatly influenced my teaching and my choice of classes, and she was there when I needed a discerning and sympathetic ear. She is a treasure. Risa Schwartz has been a friend practically since childhood, and in one way or another has supported me all the way.

Finally, to my parents, who are my friends (how many can truly say that?), gratitude beyond words.

ABSTRACT

Students of systems software need an environment in which to develop and test operating systems. Simulation, which gives the impression of access to normally privileged features, is a way to provide this environment; it is usually not economically feasible to dedicate a large multi-user computer for such purposes.

This thesis represents the third phase in the creation of SIMON, a VAX-11 operating system simulator. Facilities added during this phase permit students to test I/O services such as assigning channels and performing actual transfers using files that simulate printers and terminals. Simple screen and file I/O capabilities for debugging purposes have also been implemented.

The paper contains an overview of SIMON's previous two phases, an outline of VAX-11 I/O architecture and VMS I/O strategies, a presentation of the I/O facilities provided by SIMON, and two examples illustrating these facilities. Appendices contain reference guides for system space parameters, generic program form, extended operations, and other invariants of the testing environment.

TABLE OF CONTENTS

CHAPTER	PAGE
I. INTRODUCTION	1
II. VAX-11 I/O FACILITIES	5
II.1 Overview	5
II.2 Buses and Bus Structures	6
II.3 Programmed and DMA I/O	7
II.4 I/O Interrupts	8
II.5 I/O Requests	10
II.6 Software Interrupts	11
II.7 VMS I/O Strategies	12
II.7.1 VMS System Services for I/O	13
II.7.2 The Use of Device Drivers by VMS	15
II.7.3 An Overview of the VMS I/O Database	17
II.7.4 Asynchronous System Traps	18
II.7.5 VMS I/O Request Processing	19
II.7.6 Fork Processes and the Use of IPLs	21
II.8 Summary	26
III. I/O FACILITIES PROVIDED BY SIMON	28
III.1 Overview	28
III.2 Simple I/O	28
III.3 Device Controller Registers	30
III.4 Assignment and Deassignment of I/O Channels	31

III.5 Actual I/O Transfers	31
III.6 Timing	33
IV. SAMPLE EXECUTION	34
IV.1 Chapter Overview	34
IV.2 Example 1: Simple I/O for Debugging	34
IV.3 Example 2: Printer and Terminal I/O	40
LIST OF REFERENCES	64
APPENDICES	66
A. I/O address space	67
B. I/O controller registers	68
C. I/O device parameters	69
D. Summary of extended operations	70
E. Generic test program form	71
VITA	74

LIST OF TABLES

TABLE		PAGE
1	Filenames, channel numbers and SCB offsets for I/O devices	69
2	Extended operations using XOPC macro	70

LIST OF FIGURES

FIGURE	PAGE
1 VMS I/O request processing flow	20
2 VMS I/O system use of fork processes and IPLs	24
3 Example 1 – Simple I/O for debugging	35
4 Example 1 output to screen	37
5 Example 1 output file OUT.DAT	37
6 Example 2 – Printer and terminal I/O	41
7 Example 2 output file TEST.OUT	52
8 Example 2 output file LP1.OUT	54
9 Example 2 output file LP2.OUT	54
10 Example 2 input file TT0.IN	54
11 Example 2 output file TT0.OUT	54
12 System space addresses of controller registers	67
13 Data buffer register (printer and terminal)	68
14 Printer control/status register	68
15 Terminal control/status register	68
16 Generic program form	72

CHAPTER I

INTRODUCTION

Operating system simulators, such as ASSIST-V for the IBM 360/370 series, give the impression of access to features of a computing system that are normally reserved for systems managers or other privileged users. These simulators provide students of multi-user operating systems an environment in which to develop and test a special genre of software – that which controls and manages the system itself. It was for this purpose that the VAX operating system simulator, SIMON [SImulator for a Multi-user Operating system eNvironment], was created.

SIMON is primarily an educational tool. The study of operating systems often constitutes a major widening of the horizons for beginning students. Although such students may have much experience with a large machine in a multi-programming environment, they may be totally unaware of the underlying hardware/software interaction that accomplishes familiar tasks. Teachers of computer science are often faced with the difficult question of how to introduce students to the wealth of material in the field of operating systems without dedicating a machine to that end. Since this is often an economic impracticality, an alternate method, such as simulation, must be employed to give students a “first-hand” feel for systems software development.

Logically speaking, SIMON serves as the hardware on which students may design and test operating system software: it consists of the CPU, memory and the various peripheral controllers that comprise a VAX-11 computer system. Truthfully speaking, SIMON is merely software; nevertheless, the enterprising user can build elaborate software creations upon it as if it were the machine itself. The main elements of the

simulated architecture are as follows:

- A major portion of the VAX instruction set (162 instructions out of the total 249) and various extended operations, such as simple I/O for debugging.
- Interrupts and exceptions.
- Instruction timing, including timing of I/O transfers.
- Memory management facilities that make possible the testing of complex routines, such as page replacement algorithms.
- I/O facilities that allow for assigning channels and performing actual I/O transfers.

Input to SIMON is in the form of a VAX MACRO-11 "image file" (the equivalent of an IBM "load module"). In other words, a user of SIMON writes MACRO-11 assembly language source code, assembles and links it, and then executes SIMON on the machine code in the image file. Guidelines for a generic test program form are included in Appendix E. Although only one PSECT is allowed in the current version, logical divisions between "operating system" and several "user" portions are easily constructed. SIMON fully expects interaction between the operating system, the user and SIMON itself. This three-way conversation can be quite complex: system services may be requested by the user and performed by the operating system, processor access modes may be changed to facilitate user requests, memory management features may be enabled or disabled, software interrupts and Asynchronous System Traps may be specified, context switching between multiple user programs may be performed by the operating system, and so on.

The development of SIMON was divided into three virtually distinct phases. The first [7] simulated the instruction set, interrupts and exceptions, and instruction timing. Primary memory is simply an array to SIMON. Initially, the machine code is extracted

from the image file and read into the memory array. Execution of the program is accomplished by stepping through the array with a program counter. Implemented in the first phase are extended operations for dumping the contents of registers (including PC, relocated PC and condition codes) and halting program execution. Examples in [7] illustrate software interrupts, exceptions, context switching and instruction timing.

The second phase [13] incorporated memory management facilities. An additional array was added to simulate the storage of logical pages on disk; paging takes place between the two arrays. Revision of the original addressing scheme resulted in a relocation factor whose value is the memory array address, not the initial entry point as it was in the first phase. Two instructions, `PROBER` and `PROBEW`, were added for protection of code and data; since this protection is provided at the page level on the VAX, inclusion of these instructions was deferred to the second phase. `SIMON` may be run with or without the memory management features. Illustrated in [13] is an elementary paging routine, the added instructions for protection, and extended operations for the I/O associated with paging.

In this, the third phase, I/O facilities have been added to `SIMON`. These facilities include simple output (to screen and/or file) for debugging purposes and simulated peripheral controllers that make possible the inclusion of I/O routines, such as those for assigning channels and performing actual transfers (to and from files). I/O device controllers activate and control the simulated peripherals by manipulating bits in the control/status registers. These registers contain read-only bits to indicate device readiness. These bits are initially set, but are cleared whenever an actual transfer occurs; resetting is accomplished through an extended operation that simulates the device interrupt. The control/status registers reside in system address space alongside the data buffer registers. VAX I/O is memory mapped, so ordinary data movement instructions

are utilized to read and write the data buffer registers. Timing of I/O transfers is also implemented in this phase.

SIMON was developed on a VAX-11/780 and is intended to run on any of the VAX-11 family of computers. Since it is written entirely in C using standard include library files (stdio), it should be portable to other C environments if such systems can assemble and link MACRO-11 code to produce VAX-like image files. C is an ideal language for this type of simulation: it provides low-level features for the extensive bit manipulations required and high-level aspects that promote modularity and hence modifiability.

The authors of SIMON have taken the pedagogical nature of the project seriously. From the outset, SIMON was designed to be not only a reasonable approximation to VAX-11 architecture, but also a usable tool for operating systems classes. The need for simulators by university computer science departments has been affirmed by the widespread academic use of ASSIST-V since its creation in 1976. It is not foreseeable that the study of operating systems will be made obsolete by future technological advances; nor is the cost of mainframe computing expected to decrease drastically in the near future. In view of this continuing need for operating system simulators and industry's ubiquitous acceptance of the VAX, it is hoped that SIMON will find a place alongside ASSIST-V as a well-used academic aid.

CHAPTER II

VAX-11 I/O FACILITIES

II.1 Overview

In many ways the VAX-11 family of minicomputers is similar to both its larger and its smaller predecessors. I/O on the VAX is memory-mapped, which is more common for small microcomputers. One megabyte in the upper half of the physical address space is reserved for I/O device registers. References to the fixed addresses in this area are thus to locations in I/O *controllers* connected to the two main types of I/O buses, the *UNIBUS* and the *MASSBUS*. The I/O address space is marked for Kernel mode (most privileged) access only.

I/O controllers (also referred to as *device controllers*) essentially free the CPU from overwhelming involvement in the details of I/O transfers by acting as slave processors. While the CPU may initiate an I/O activity, it is the controller that carries it out and informs the CPU of the result. For some kinds of I/O controllers (e.g., terminal multiplexers) one controller may be used for many actual devices. Specific bits or bit patterns in the registers of the device controllers serve to activate the peripherals or to return status information. These values are placed in the controller registers or read from these registers by ordinary data handling instructions; there are no specific I/O instructions in the VAX instruction set.

The operating system (which performs all device-independent processing) and the device drivers (which perform all device-dependent processing) cooperate in the management of the logical connection between processes and I/O devices. These VAX features

are among those that combine to make possible concurrent CPU computation and I/O processing; this concurrency is the hallmark of modern multi-programmed operating systems.

II.2 Buses and Bus Structures

The larger VAX-11s (such as the VAX-11/780) use the *Synchronous Backplane Interconnect* (SBI) to connect the processor, memory and I/O buses. The SBI (which has a clock cycle time of 200 nanoseconds) is fundamentally a 32-bit wide (longword) bus, but double transfers allow 64 bits (quadwords) to be transmitted based on one address. Thus the SBI may be considered to be a 64-bit bus with a rated transfer speed of 13.3 Mbytes/second. The SBI uses 30-bit addresses to access both I/O controller registers and physical memory.

I/O controllers are rarely connected directly to the SBI, but rather to either the MASSBUS or UNIBUS, both of which are 16-bit buses. Because the SBI is not, strictly speaking, a part of the architecture of the VAX-11/780, but rather implementation specific, and because it was necessary to design a bus structure that could accomodate existing I/O devices, a *backplane adapter* connects each MASSBUS or UNIBUS to the SBI. The MASSBUS adapter (MBA) and UNIBUS adapter (UBA) are functionally similar to the memory controllers on the VAX in that they serve both to connect devices (in addition to the CPU) to the SBI and to utilize these devices efficiently. Through the internal mapping registers of the MBA and UBA, devices may address SBI physical memory. Address mapping is needed since many UNIBUS and MASSBUS devices were originally designed for various models of the PDP-11 or LSI-11 series, all of which have much less storage capacity (and a maximum of 16- or 18-bit addresses).

The MASSBUS is asynchronous for control and synchronous for data. It is used

mainly for interfacing to high-speed disk devices and can have a data transfer rate of as high as 2 Mbytes/second. Each MBA contains a 64-bit buffer which is used both for input and output. On input, the buffer collects four 16-bit words from the device before it initiates the (64-bit) SBI transfer. On output, the buffer is loaded with 64 bits of data, which it delivers to the device 16 bits at a time. A VAX-11/780 can have as many as four MBAs, each one with up to eight device controllers. Only one device can transmit data at a given time on each MBA, although it is possible to initiate certain operations (e.g., disk seeks) simultaneously.

The UNIBUS is fully asynchronous. It uses handshaking protocols with the slower devices to which it is typically connected. Like the MBA, the UBA maps addresses and buffers data. Buffering is accomplished by means of small (64 bytes), high-speed, first-in/first-out devices called *SILOs* (for Service In Logical Order). The UBA's *SILOs* allow the UNIBUS to transmit data to the SBI in 64-bit chunks. Its data transfer rate can be as high as 1.5 Mbytes/second. Unlike the MBA, the UBA allows as many as 16 data transfers to be active simultaneously. It has 16 (16-bit wide) data paths, each of which buffers data. In other words, only one SBI bus cycle is needed for every four UNIBUS cycles. Smaller VAX-11s use the UNIBUS exclusively.

II.3 Programmed and DMA I/O

Slower devices use *programmed* I/O in which the CPU is involved in the transfer of each byte to or from the controller's buffer register. Most faster peripherals, however, use *direct memory access* (DMA) I/O, also called *non-processor request* (NPR) I/O, which is accomplished partly through the I/O controller and partly through the backplane adapter. The backplane adapters buffer the data for the 16-bit MASSBUS or UNIBUS and attempt to make 64-bit DMA transfers. This allows the SBI to perform alternative

tasks and still catch up with the DMA transfers. Contention for main memory is solved by cycle stealing, in which the the I/O controller has precedence over the CPU.

II.4 I/O Interrupts

I/O interrupts are signals from the various I/O controllers to the CPU which indicate completion (either success or failure) of an I/O operation. The CPU then is automatically diverted from executing the current process (whose status is saved on the Interrupt stack), so that it may begin executing the routine that handles the I/O condition. Thus it is possible to think of an I/O interrupt as a means for the device controllers to call an operating system routine that services the condition causing the interrupt. The stack architecture of the VAX plays a major role in the handling of I/O conditions. It should be noted that all interrupt processing occurs independently of the currently executing process; in other words, interrupts are always serviced on the system-wide Interrupt stack, never on one of the process local stacks in process context.

The VAX/VMS operating system uses interrupts for both programmed and DMA I/O. In programmed I/O, each of the I/O controllers uses its special buffer register for data transfers. On input, the CPU is interrupted when the I/O device has placed a single character into this buffer register. On output, the interrupt occurs when the I/O device is ready for the CPU to load the buffer with the next character to be written (i.e., immediately after the previous character has been written by the device). In some cases, the controller itself may do additional buffering. For example, the LP11 line printer controller reduces the number of interrupts by holding several characters in an internal buffer and interrupting only when all characters have been sent to the printer and the buffer is emptied.

There is only one interrupt generated by a DMA transfer — when the operation

is completed (successfully or not). To initiate DMA I/O, the CPU loads transfer information into special registers known as *control and status registers* (CSRs) in the I/O controllers. This information includes:

- the length of the block to be moved,
- the location of the main memory buffer,
- the device unit and device location involved, and
- the operation to perform.

The VAX family uses a vectored interrupt system. The vector table or *System Control Block* (SCB), which is also used for exceptions, consists of longwords marked for Kernel mode access only. When an interrupt occurs the *Processor Status Longword* (PSL) and the *Program Counter* (PC) are saved by the hardware on the Interrupt stack. Hardware also causes the appropriate vector to be placed in the new PC, thus effectively diverting the flow of execution to the interrupt handling routine's entry point. At the end of the interrupt service routine, the interrupted process is resumed with a RETURN FROM EXCEPTION OR INTERRUPT (REI) instruction, which restores the state of the interrupted process by popping the necessary information off the stack.

Actually the SCB entries are divided into two fields, one of which is the (low order) two-bit code indicating how the event should be serviced. This may include such aspects as whether the event is to be serviced on the Interrupt or the Kernel stack, whether the IPL is to be changed, and whether the event is to be handled by *Writable Control Store* (WCS) microcode. The remaining bits include either the virtual address (30 bits with two 0 bits appended, implying a longword boundary) of the service routine to be executed or information (14 bits) passed to WCS. [Microcoding is not within the range of SIMON's capabilities, so the SCB will be considered simply as a vector table.]

Each interrupt occurs at one of the 32 *Interrupt Priority Levels* (IPLs) so that the processor can arbitrate among requests by using a priority scheme. The IPL of the executing process is contained in bits <20:16> of the PSL; switching between priority levels is thus an automatic consequence of stacking the PSL. All user mode processes and most operating system code use the lowest priority level, IPL 0. Levels 1-15 are used by the operating system as software interrupt levels for routine calls as described below. The highest IPLs are reserved for hardware interrupts, specifically for peripheral devices (16-23), the system clock (24), and urgent processor error conditions, such as power outage (25-31).

II.5 I/O Requests

In general, an I/O request is directed to a channel that has been previously associated with a particular device unit. All I/O is managed by the operating system after the establishment of such a logical path between a process and a peripheral. A process that makes a request for a non-shared device simply allocates that device while it is in use. This may be accomplished by invoking the Allocate Device system service, \$ALLOC, which reserves a particular device for use by a process and its subprocesses. The operating system maintains information about the allocation of devices to ensure that no other process uses a device that has been allocated. Once a device has been reserved, no other process can allocate it or assign a channel to it; the device is released either by the exit of the image of the calling process or by explicit deallocation (using \$DALLOC). For shared devices all I/O requests are checked by the operating system. Under VMS, if the user's privileges are adequate, the requests are queued by priority, handled first-in/first-out within each priority, and processed strictly in queue order.

II.6 Software Interrupts

Any of the software interrupt vectors (IPLs 1-15) may be used for calling an operating system routine. Five of them (IPLs 6 and 8-11) have important functions in controlling VMS I/O operations. These five IPLs lend themselves to the splitting up of I/O processing into manageable pieces that have inherent priorities. For example, a Kernel mode routine can invoke a service routine to perform a specific function by requesting an interrupt at the appropriate level. This is an efficient way for the operating system to provide synchronization for different levels of execution. The following three privileged registers are provided for the management of software interrupts:

- Software Interrupt Summary Register (SISR)
- Software Interrupt Request Register (SIRR)
- Interrupt Priority Level Register (IPL)

The Summary Register contains a bit for each of the 15 IPLs, thereby summarizing the entire list of pending interrupt priorities. For example, if there are software interrupts pending at IPLs 3 and 7, then bits 3 and 7 of the Summary Register would be set. The Request Register could be written by the Kernel mode routine mentioned above by use of a `MOVE TO PROCESSOR REGISTER (MTPR)` instruction. The interrupt requested would be taken either immediately or later, depending on the current IPL of the processor. For example, if the requested level of 9 is moved to the Request Register, an interrupt would be taken immediately if the processor were currently executing at IPL 8; however, if the processor were executing at IPL 11, then bit 9 of the Summary Register would be set instead. Not until the processor level subsequently dropped below 9 would the interrupt occur.

Kernel mode software can directly manipulate its IPL by using the IPL Register to read or set the IPL field (bits <20:16>) of the PSL. Raising the IPL is the direct result of the MTPR instruction that specifies a higher priority than the current value in the PSL; however, requesting a lower level may result in an interrupt from another source if the requested level is below the level of any pending interrupts summarized in the SISR. The software IPLs are used by VMS (see Section II.7.6) to increase I/O device utilization, maintain responsiveness and synchronize operations. Basically this is done by requiring that a specific operation be performed only at a known level.

II.7 VMS I/O Strategies

The fact that the VAX/VMS (Virtual Memory System) operating system was developed in conjunction with the VAX-11 architecture is probably the crux of its success. Because of this co-design process, VMS, which provides support for the management of the CPU, memory and I/O devices, closely mirrors many aspects of the underlying hardware. For example, VMS manages memory and protection by manipulating page tables and privileged registers, and VMS I/O request completion makes use of the interrupt mechanisms, the stack architecture and software IPLs.

The following sections (II.7.1–II.7.6) present *only one* possible implementation (VMS) of an operating system's support for I/O operations. The student of systems software is advised that an infinite variety of operating systems can be constructed for any particular architecture; the fact that this discussion focuses on VMS should in no way limit the imagination of creative software developers. SIMON does not attempt to reflect VMS or any other specific system: it simply simulates the hardware of the VAX-11 family.

The following discussion of VMS I/O strategies is included for two reasons:

1. It introduces students of operating systems to a complex and successful product in widespread use. Many of the features of VMS have been emulated by more recent operating systems. In other words, VMS serves as a useful example of what a VAX operating system can be.
2. It distills information from a wide variety of sources [2,6,10,11,12] for researchers of VAX/VMS I/O facilities.

Those features of VMS that are beyond the scope of SIMON are discussed in the summary at the end of this chapter.

II.7.1 VMS System Services for I/O

VMS provides system service routines for a variety of operations including I/O. These routines may be invoked by programs written in assembly language or higher-level languages. The mechanism need only employ the normal method used to call system subroutines, but one may also use system macros which expand into a series of system subroutine calls. These macros (beginning with the \$ sign and defined on a library file) greatly simplify the call sequence necessary for complex I/O operations. An integral part of VMS I/O processing, the \$QIO macro generates a call to the *Queue I/O* (QIO) routine (described below) and allows the user process to continue without waiting for I/O completion. The \$QIOW macro is similar, except it places the user process in a wait state (by combining the \$QIO and \$WAITFR services); in other words, the calling process is suspended until the I/O completes and the operating system sets the associated event flag.

The QIO routine is called for every VMS I/O request. It executes in Kernel mode, but within the user process context (i.e., it has access to the user process address space).

It is this routine that has responsibility for the following functions:

1. To validate the arguments supplied by the user process. This includes checking the privileges of the user process to perform the requested function on the device specified.
2. To build the data structure that describes the request, the I/O Request Packet (IRP), in system dynamic memory.
3. To queue the IRP to a specific device.
4. To return to the calling user process.

The QIO routine indicates success on its return to the user process if the parameters have been checked and the IRP has been built and queued. This says nothing whatsoever about the successful or unsuccessful completion of the I/O operation itself. The user process can learn about all details of the I/O transfer only by specifying a location, the *I/O Status Block* (IOSB), for the final status to be placed by the I/O postprocessing routine. I/O postprocessing also sets an event flag to indicate completion of the I/O transfer; the user process can examine this flag to determine whether the transfer has completed.

In addition to the system macros mentioned above, there are those associated with the VMS *Record Management Services* (RMS), a facility provided for complex file manipulation. The expansion of such macros includes not only the issuing of QIO requests, but also the generation of data tables that are used to define files and records for RMS routines. Assembly language programs may issue a \$QIO system service directly (as indicated above) or indirectly (to record-oriented devices) by using RMS. Programs in high-level languages normally specify I/O operations using statements belonging to the language; these are translated into RMS operations, which in turn are translated

into \$QIO system service calls. [Because the current version of SIMON deals only with the *device control* or actual *physical* level of input and output, the RMS file-oriented facilities are beyond its scope and will not be considered further.]

II.7.2 The Use of Device Drivers by VMS

Midway between the user process and the I/O hardware, and surrounded on all sides by the operating system, are the various device drivers. VMS performs all device-independent I/O processing and transfers control to the device drivers to perform device-dependent I/O processing. The main jobs of an I/O driver under VMS are:

- to initialize the device and/or controller at system startup and after power failure,
- to define the device for the rest of the operating system,
- to define the driver and its database for the loading process,
- to translate I/O requests into device-specific commands,
- to activate the device,
- to respond to interrupts generated by the device,
- to report errors, and
- to return data and status information to the user process.

Under VMS the code of a specific driver is associated with a certain type of peripheral and is shared by the driver processes invoked to activate and control devices of that type. Thus there may be various driver processes in various stages of execution operating concurrently at any given time.

Drivers execute at higher priority levels than most other processes, so they are typically interrupted only by other drivers at higher IPLs; they cannot be interrupted by the process scheduler. This is not as inefficient as it may at first seem, since the drivers themselves contain the minimum code necessary to perform the actual I/O. Low priority levels are used for the code preceeding and following the invocation of a driver routine — that is, for the code that handles verification of I/O requests and notification of completion. The highest priority critical sections are very short and handled quickly by VMS. A driver may also place itself in a wait state.

When a valid request for I/O services is made, the operation is always performed independently from, but completed within the context of the requesting process. In other words, QIO preprocessing operates within user process context, as does I/O post-processing. But in order to perform the actual I/O transfer, VMS creates a separate intermediate process, a device driver process, which operates in the limited context in which only system address space is accessible. The driver, which contains both code and static tables describing each device, cooperates with VMS by sharing a common I/O database.

In order to initiate the physical I/O transfer, a driver may have to determine the allocation of software as well as hardware resources. Thus there must be some coordination between drivers as well as a mechanism for informing driver processes when a resource becomes available. I/O scheduling and synchronization are among the most complex of the functions of VMS; these aspects will be discussed after the main structures and the basic functions of the I/O database are outlined.

II.7.3 An Overview of the VMS I/O Database

The six data structures described in this section are among those used by VMS to manage the logical paths from processes to peripherals.

The *Device Data Block* (DDB) may be logically deemed to be the head of the system structures in the VMS I/O database in that it contains the generic device name and driver name for the set of devices of a given type connected to a single controller. For example, if Terminal Controller E has several terminals attached to it, the device name string TTE might be used in the DDB to describe each of these terminals.

The *I/O Request Packet* (IRP) contains everything the device driver needs to know in order to perform physical I/O. It is also used for I/O postprocessing, that is, for returning the final status and data to user process memory. As the operation is performed, the IRP is passed around by VMS; various system components at different times check the information in the IRP and/or add to its contents. Among other things, the IRP contains:

- the identification of the requesting process,
- the type of operation requested, and
- the address and length of the user's buffer.

It should be noted that IRPs are dynamically allocated packets, whereas the rest of the I/O database described in this section consists either of system data structures or tables within the drivers themselves.

Under VMS there is one *Unit Control Block* (UCB) for each active device. It bears the same relation to the device driver as the PCB does to the process. In other words, it contains all the state information of the current driver process. It also contains the

status of the particular unit with which it is associated and a header for the queue of IRPs. When QIO queues an IRP, it queues it to the UCB of a particular unit of the requested device type.

The *Channel Request Block* (CRB) defines for VMS the current status of a single I/O controller. For shared controllers, it arbitrates among requests by indicating which particular device unit is in the process of performing an I/O transfer and which other units, using the same controller, are waiting to operate.

The *Interrupt Dispatch Block* (IDB) records the characteristics of an I/O controller and points to the particular device it controls when an interrupt occurs.

The *Adapter Control Block* (ADB) records the characteristics and defines the current status of a UNIBUS or MASSBUS adapter.

Each of the VMS data structures above not only has a static relationship with the other structures in terms of pointers, but also may be associated with other system components or even serve as a member of other queues depending on the current status of I/O activity.

II.7.4 Asynchronous System Traps

VMS provides a feature that may be used for a variety of purposes both by user processes and the operating system itself. The *Asynchronous System Trap* (AST) may be employed by a user process to include its own I/O completion routines which are activated by normal system interrupts. For example, a process may specify an AST routine to be executed when I/O completes. If the process itself is executing at the time of the interrupt, the AST routine will be invoked, effectively interrupting the process to execute code at another point in the process address space. When the AST routine finishes, the process normally would be resumed at the point of interruption. A

process in a wait state would generally return to the wait state after the execution of the interrupting AST. In other words, an AST invokes a routine asynchronously to the execution of a process, but within the context of that process.

Several ASTs may be outstanding for a particular process at any one time. The Software Process Control Block for each process contains a queue of AST control blocks. Actually the operating system manages four logical queues (corresponding to the four access modes) for each process. It is common for VMS to use ASTs to execute Kernel mode routines on behalf of a process within the address space of that process. This practice has important consequences for I/O postprocessing, as described below.

II.7.5 VMS I/O Request Processing

Using the components of the VMS I/O system introduced above, it is possible to clarify the earlier description of I/O request processing. Figure 1 shows the sequence of events that takes place when VMS processes a single I/O request. This flow diagram illustrates what has been described above by breaking down the overall process into the following fundamental steps:

- the calling of QIO by the user process,
- the building of an IRP by QIO,
- the execution of the driver process, and
- I/O postprocessing.

In order for QIO to validate the parameters supplied in the call by the user process, it must locate the UCB and associated database for the particular device involved. The

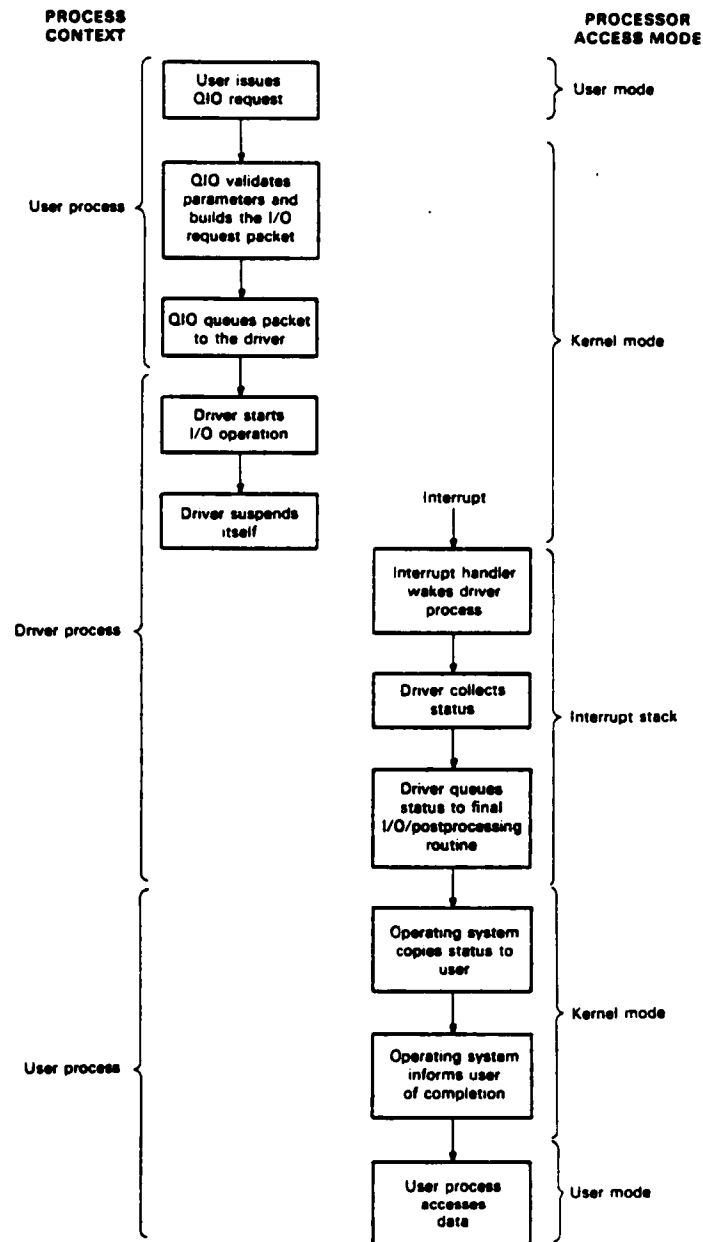


Figure 1: VMS I/O request processing flow

Source: Henry M. Levy and Richard H. Eckhouse, Jr., *Computer Programming and Architecture: The VAX-11*. Digital Press, Bedford, MA, 1980, p. 310.

QIO routine is responsible for checking both device-independent parameters and device-dependent parameters. It does the latter by calling subroutines in the driver itself.

When the driver has dequeued the IRP and initiated the actual transfer, it places its UCB in a wait queue, thereby suspending itself. It stays in the wait queue until the device interrupt occurs and the subsequent interrupt handler routine takes it out of suspension. Once the operating system passes control back to it, the driver copies the status of the device and I/O controller registers into the IRP. It is this status information which determines the nature of I/O postprocessing. Using Kernel mode, VMS returns the information associated with the I/O operation to the user process address space and sets an event flag to indicate that the operation has completed.

Actually, as students of multi-user operating systems have come to expect, the situation is not so straightforward. As mentioned above, VMS seeks to maximize throughput and minimize response time by subdividing I/O request processing into routines of varying priority. In other words, certain portions of all outstanding I/O requests are executed at a higher priority than other portions of the same group of requests; an entire request is not processed sequentially. Instead those functions that are most critical (e.g., reacting to a device interrupt) are performed first for each request, followed by those functions that are slightly less critical, and so on, as described below.

II.7.6 Fork Processes and the Use of IPLs

VMS handles device drivers as *fork processes*, which are created dynamically and contain minimal context. There is a *fork queue* associated with each of the software IPLs used for I/O processing. Special control blocks for fork processes, *fork blocks*, are dispatched from these queues by the *fork dispatcher* on a first-in/first-out basis. When the queue at a particular level is exhausted, execution continues at a lower IPL.

Much of VMS I/O request processing is accomplished through the use of fork processes and software interrupt priority levels. These mechanisms are used by VMS to optimize device utilization, which in turn tends to improve throughput without sacrificing real-time device responsiveness.

VMS uses many of the VAX hardware features in completing I/O requests. Figure 2 shows the four priority levels employed by VMS and charts the activity within each level. All work at one level is done before any at the next (lower) level can begin. Dropping down from level to level is done by requesting software interrupts at lower and lower IPLs (except for the final descent into user process context which is accomplished through an AST).

The highest level (Device IPL) is used for the response to the device interrupt. This is typically the highest priority activity in the VMS I/O system, which is designed to respond to a hardware interrupt as quickly as possible, so that other interrupts will not be locked out. In fact, the only actions taken at this level are as follows:

1. The routine in the driver that is responsible for handling the interrupt is located and initiated by the device interrupt dispatcher.
2. The status information in the device controller registers (CSRs) is copied into the UCB.
3. The driver forks, i.e., asks to be completed at a lower priority at some later time. VMS saves the status of the driver in the UCB for the device and queues the UCB on a fork queue at a lower IPL.
4. The driver requests a software interrupt at the IPL associated with the fork queue.
5. The driver returns from the interrupt, thus immediately enabling interrupts from other devices.

Sometime later, in keeping with the software interrupt mechanism described above, the next level (Fork IPL) of VMS I/O processing begins. The activities that take place at this level are as follows:

1. The fork interrupt dispatcher dequeues the UCB and restores the context of the driver so that it may continue processing the associated I/O request.
2. The driver finishes executing the I/O operation, releasing as it goes any resources it had allocated.
3. The driver places any status information needed for I/O postprocessing into the IRP and queues the IRP at the (lower) I/O postprocessing IPL.
4. The driver requests a software interrupt at the IPL associated with the I/O postprocessing queue.
5. The driver checks to see if there are any IRPs in the request queue for use of the device in question. If there are, it initiates the operation at this (the earliest possible) time, thus keeping the device as busy as possible.
6. The driver returns to the fork interrupt dispatcher, which starts the activities at this level again if there is another fork block queued. If not, Fork IPL is exited (as are all levels in the scheme) by use of the REI instruction.

It should be noted that by forcing all device drivers to queue at a certain level (as fork processes) VMS ensures that synchronization problems do not occur. Since only one driver process can be active at Fork IPL, only that process can access the database at a given time. In other words, the shared device or controller databases which may be accessed by several driver processes concurrently will be safe automatically, thanks to the IPL mechanism.

The next level of VMS activity (I/O postprocessing IPL) is initiated by another software interrupt in exactly the same manner. The events that occur at this level are as follows:

1. The postprocessing interrupt dispatcher dequeues the IRP and completes any processing that can take place within system context.
2. The I/O postprocessing routine prepares the switch to user process context by setting up an AST. To do this it:
 - (a) queues the address of one of its subroutines (to be executed in Kernel mode within user process context) to the Software Process Control Block of the user process, and
 - (b) makes a note in the Hardware Process Control Block that a Kernel mode AST is pending for the particular process.

The lowest level (AST IPL) is initiated by an interrupt caused by VAX hardware when the process next executes after Step 2 above. VMS then executes the Kernel mode AST routine which performs the following functions:

1. It copies the necessary information (final status of the operation and any data involved) into the user's address space.

2. It sets the event flag that indicates completion of the transfer.
3. If the user process had requested its own AST routine, the Kernel mode AST routine would have queued an AST for the user process.

II.8 Summary

The VAX-11 architecture evinces many unique features which make the construction of a simulator a challenging project. The underlying hardware has been exploited by the designers of VMS to provide features that have been emulated by other operating systems for both the VAX and dissimilar machines. Obviously, many of the hardware features that have been supported by VMS cannot be simulated in a project of this size. In all three phases of SIMON, certain architectural aspects of the VAX have been emphasized to the exclusion of others.

In the present phase of SIMON's development, simple output (for debugging) and simulated I/O device controllers have been implemented. All of the transfers are programmed; there are no facilities for DMA operations. Buses and adapters are assumed to be functional and error-free. The interrupt mechanisms, which were established in the first phase, are utilized for each character involved in an I/O transfer.

The basic scenario for I/O requests includes the assignment of a channel, the byte-by-byte movement into the controller data buffer registers, and the generation of a device interrupt for each byte transferred. Software interrupts and ASTs were implemented in the first phase, and can be used for I/O request completion in a manner similar to that employed by VMS. Fork processes are not simulated.

The VMS I/O database and the specifics of VMS handling of device drivers and I/O request completion were included to provide one example of the possibilities available to writers of systems software. The operating systems student should realize that there

are many other methods of providing similar services; furthermore, there are other operating system features that are both desirable and feasible which are not provided by VMS.

CHAPTER III

I/O FACILITIES PROVIDED BY SIMON

III.1 Overview

In order to display their essential features and all intermediate aspects of execution, the operating system simulations prior to the third phase of SIMON utilized register dumps (including PC and condition codes) and messages, both output to screen. I/O facilities in the third phase of SIMON, in addition to providing direct I/O capabilities for debugging, make possible the testing of operating systems that include complex I/O services such as routines for assigning channels and performing actual transfers. A portion of the high end of system address space (see Figure 12, Appendix A) is now reserved for simulated I/O controllers, specifically control/status registers (CSRs) and data buffer registers (DBRs) for three printers and three terminals.

III.2 Simple I/O

Because SIMON was designed for a pedagogical setting, in which the friendly debugging of simulations is desirable, the third phase implemented a feature for direct test output from the user's test programs. Two new extended operations, XOPCs 5 and 6, are provided so that users may perform simple output in a variety of ways. [See Table 2, Appendix D for a summary of SIMON's extended operations.] XOPC 5 toggles between output to screen (default) and user-specified files; XOPC 6 performs the actual output. Both operations expect parameters to be passed in *registers 0 and 1*.

If the user is content to have all debugging messages and dumps sent only to the screen, it will never be necessary to code an XOPC 5. If, however, it is desired to have some or all output sent to one or more files instead of (or in addition to) the screen, XOPC 5 is the means of selection. When XOPC 5 is executed, *register 0* should contain either the address of the first character of a string describing the output file or 0 for selecting screen output. If *register 0* contains 0, *register 1* is not checked. Otherwise, *register 1* should contain the length (byte count) of the filename string descriptor. For zero-terminated filenames (e.g., the result of an `.ASCIZ` directive), *register 1* may contain 0 instead of the string length. If XOPC 5 has not been previously executed (or if it has been executed while *register 0* contains 0), all subsequent debugging output will be sent to the screen. Otherwise it will be sent to the file determined by the most recent XOPC 5. To allow for a nine-character filename, a period and a three-character filetype, the maximum number of bytes that may be specified by *register 1* is 13. If *register 1* contains 0, the string is assumed to be zero-terminated by byte 14.

XOPC 6 supplements the register dumps and error messages of SIMON's earlier phases by providing a means for output of character strings directly from the user program. This output is intended to be information generated during program debugging. When XOPC 6 is executed, *register 0* should contain the address of the output buffer, that is, the address of a string to be printed. The destination of the printing is determined by the previous XOPC 5 or, if none, by default. *Register 1* may contain either the length (byte count) of the string to output or 0 if the string is zero-terminated. The maximum number of bytes that may be specified by *register 1* is 134. If *register 1* contains 0, the string is assumed to be zero-terminated by byte 135. The use of XOPC 5 and XOPC 6 is illustrated in Example 1 (Section IV.2).

III.3 Device Controller Registers

The simulated device controllers manage I/O by manipulating bits in the CSRs. Since the controller registers are at known locations (see Figure 12, Appendix A), it is possible for the test operating systems to read and write them directly. However, certain bits are write-protected in order to prevent bypassing the controller. All device registers are 16 bits long, but only a few bits are used in the CSRs; the DBRs use only the low order byte for character transfers (see Figure 13, Appendix B).

The printer CSRs (see Figure 14, Appendix B) contain a DONE bit (bit 7) and an Interrupt Enable (IE) bit (bit 6). The DONE bit is read-only; it is set by the controller whenever the printer is ready to receive the next character. In other words, after each byte is successfully "printed," SIMON sets the DONE bit to tell the operating system it may move the next character into the DBR. Setting the DONE bit is a consequence of the XOPC 8 extended operation described below.

The IE bit may be used to enable and disable device interrupts. Typically, an operating system would clear the IE bit while it tested the DONE bit. If the test were to reveal that the printer is unready, the IE bit would be set and the operating system would wait for an interrupt. The waiting (XOPC 8) causes the interrupt to be simulated. This is explained in more detail below and illustrated in Example 2 (Section IV.3).

Each terminal controller manages two logical devices — one for output, the other for input — and contains both a *transmit buffer register* (TBUF) and a *receive buffer register* (RBUF). There is only one CSR for each terminal; it contains bits for both input and output (see Figure 15, Appendix B). Output is handled similarly on all six fictitious peripherals; terminal input is accomplished through input files set up by the user. Bit 7, the RDONE bit, is set when a character to be read is present at the controller. The RDONE bit is read-only and is manipulated in a similar fashion to

the printer DONE bit. In other words, when XOPC 8 is executed for a logical input device, SIMON sets the RDONE bit to indicate that a character has become available in RBUF.

III.4 Assignment and Deassignment of I/O Channels

Because of the need for a logical connection between process and peripheral, SIMON includes features for channel assignment and deassignment. Indeed it is not possible to perform simulated I/O without having first communicated with SIMON regarding the channel involved. The XOPC 7 extended operation is the means whereby the user either allocates or deallocates a particular I/O channel. When XOPC 7 is executed, *register 1* should contain a valid channel number (see Table 1, Appendix C). *Register 0* should contain -1 to specify deassignment of the channel indicated by *register 1*; any other value in *register 0* specifies assignment. The channel numbers themselves are statically associated with a particular logical device. The user, however, may see fit to use them as indices into a structure that represents a portion of the device database (see Example 2, Section IV.3).

III.5 Actual I/O Transfers

In keeping with the memory-mapping of true VAX I/O, any byte moved into the output data buffer registers is interpreted as a byte to be output. Although other instructions might work, the most direct method for output is by use of a MOV_B instruction to printer DBR (or terminal TBUF). If the appropriate channel has been previously assigned, any character moved into the DBR is sent to the associated file for that device. For example, if the system being tested wishes to perform output to LP1,

it must first assign the correct channel (with XOPC 7); then subsequent transfers to the DBR for LP1 will result in movement of each such character to the file LP1.OUT in the user's current directory. Requiring prior channel assignment helps guard against unintentional I/O transfers that might take place when the user inadvertently writes a value into the I/O portion of system address space.

All devices are assumed to be error-free. This accounts for the absence of bits for indicating error conditions. The output devices are also assumed to be available and ready for the first byte of a transfer when the user program begins execution. Writing a byte to a printer's output data buffer register clears the DONE bit in the associated CSR. Since this bit is read-only, there is no way for the user to reset it and continue with the operation.

In order to output the next character, the user system is forced to wait for a device interrupt by executing the XOPC 8 extended operation. XOPC 8 eventually results in an interrupt at IPL 20 for all devices. When XOPC 8 is executed, *register 1* should contain the associated channel number. The controller demonstrates the device's readiness for the next operation by setting the appropriate bit in the CSR. On output, SIMON sets the DONE bit for printers and the TRDY (transmitter ready) bit for terminals. A typical sequence would be: XOPC 7, loop for all characters, test the DONE bit, move a byte, XOPC 8, end loop.

Terminal input is similarly accomplished through the use of XOPC 7, XOPC 8 and ordinary data movement instructions (again, likely by a MOV_B from terminal RBUF). It is the user's responsibility to have created an appropriate file for input (see Table 1, Appendix C). The input file must be named TT n .IN for terminal n . For example, if terminal 2 is being used for input, a file named TT2.IN in the user's current directory should contain the desired input. The reading of a character into the device's input data

buffer register (RBUF) is accomplished through the execution of the XOPC 8 instruction itself. The corresponding movement *from* RBUF to the user's area is necessary in order to set the read-only RDONE bit in the associated control/status register, thereby indicating that RBUF is ready for the next character of input. XOPC 7, XOPC 8 and a test operating system with I/O facilities are illustrated in Example 2 (Section IV.3).

III.6 Timing

The first phase of SIMON implemented timing of instruction execution, simulation of the three privileged timing registers, and the associated interrupt mechanisms. The current phase introduces an arbitrary timing value for I/O operations. Although actual peripherals vary greatly in speed, all logical devices simulated by SIMON are statically determined to be two orders of magnitude slower than the average CPU instruction execution time. The average instruction is considered to be one that involves a single memory reference and consumes an amount of time midway between that for a "short" and a "long" operation. Thus the *X-factor* of the first phase (0.5 μ sec.) is added to the average of the *Y-factor* (0.2 μ sec.) and the *Z-factor* (0.4 μ sec.), and the result (0.8 μ sec.) is multiplied by 100 to arrive at the overall time of 80 μ sec. for each byte involved in actual I/O transfers. An illustration of I/O timing is included in Example 2 (Section IV.3).

CHAPTER IV

SAMPLE EXECUTION

IV.1 Chapter Overview

Included in this chapter are two sample programs that execute under SIMON. The first example (see Figure 3) illustrates simple output using the extended operations, XOPC 5 and XOPC 6. The second (see Figure 6, Section IV.3) contains an operating system that consists almost entirely of I/O-related routines; this example introduces the extended operations, XOPC 7 and XOPC 8, and demonstrates simulated I/O, including timing of I/O transfers. The student of operating systems should be aware that these sample programs by no means exhaust all the possibilities for users of SIMON's I/O facilities.

IV.2 Example 1: Simple I/O for Debugging

The first example (see Figure 3) includes a variety of methods of effecting simple output; these features are intended to be used for test program debugging purposes. The extended operations XOPC 5 and XOPC 6 are the means of specifying a destination for debugging output and performing simple string output, respectively. Although there is no operating system in this example, the user is advised that the same procedures apply to any type of test program. Simple I/O is also illustrated in Example 2 (Section IV.3), where a test operating system specifies that all debugging output is to be sent to a file. The results of program execution, output to both screen and file, are shown in

```

0000 1 ;*****
0000 2 ;*      EX. 1: SIMPLE I/O FOR DEBUGGING
0000 3 ;*****
0000 4 ;*      Introduces pseudo-instructions:  XOPC 5 (route debug output)
0000 5 ;*                                         XOPC 6 (simple output)
0000 6 ;*****
00000000 7 .psect TEST,long
0000 8 .macro XOPC operand
0000 9 addb2 #'operand,rl
0000 10 . = . - 3
0000 11 .byte ~xfe
0000 12 . = . + 2
0000 13 .endm
0000 14 ;*****
0000 15 ; Storage and initialization
0000 16 ;*****
0000 17 CR = 13
0000 18 LF = 10
0000 19 testm:
0000 20 outfile:
0000 21 title:
0000 22 mess1:
0000 23 mess2:
0000 24 x:
0000 25 xx:
0000 26 mess3:
0000 27 mess3_end:
0000 28 mess3_size = mess3_end-mess3
0000 29 ;*****
0000 30 ; Initial entry point
0000 31 ;*****
0000 32 .entry TEST,~m<IV,DV>
0000 33 ; 1st message defaults to screen
0000 34 movl testm,r0
0000 35 movl #10,rl
0000 36 xopc 6
0000 37 ; set output to file
0000 38 movl outfile,r0
0000 39 clrl rl
0000 40 xopc 5

```

Figure 3: Example 1 -- Simple I/O for debugging

```

00DA 41 ; title to file
00DA 42      movl    title,r0
00DF 43      movl    #-x14,r1
00E2 44      xopc    6
00E5 45 ; mess1 to file
00E5 46      movl    mess1,r0
00EA 47      clrl    r1
00EC 48      xopc    6
00EF 49 ; dump to file
00EF 50      movl    #4,r4
00F2 51      movl    #10,r10
00F5 52      xopc    2
00F8 53 ; mess2 to file
00F8 54      movl    mess2,r0
00FD 55      clrl    r1
00FF 56      xopc    6
0102 57 ; dump to file
0102 58      movpsl  r10
0104 59      xopc    2
0107 60 ; reset to screen
0107 61      clrl    r0
0109 62      xopc    5
010C 63 ; dump to screen
010C 64      xopc    2
010F 65 ; fancy completion message to screen
010F 66      movl    x,r0
0114 67      movl    #76,r1
011B 68      xopc    6
011E 69      movc3    #mess3_size,mess3,xx+25
0125 70      movl    xx,r0
012A 71      movl    #73,r1
0131 72      xopc    6
0134 73      movc3    #mess3_size,xx,xx+25
013C 74      movl    xx,r0
0141 75      movl    #73,r1
0148 76      xopc    6
014B 77      xopc    1
014E 78      .end    TEST

```

Figure 3 (continued)

testing...

```

*** REGISTER DUMP ***
R0: 00000000 R1: 00000000 R2: cccccccc R3: cccccccc R4: 00000004
R5: cccccccc R6: cccccccc R7: cccccccc R8: cccccccc R9: cccccccc
R10: 040000a4 R11: cccccccc AP: cccccccc FP: cccccccc SP: cccccccc
PC: 0000010c CONDITION CODES: N = 0 Z = 1 V = 0 C = 0

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX Output in file OUT.DAT XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

Figure 4: Example 1 output to screen

TEST.MAR FILE OUTPUT

First dump. R4<-4, R10<-10:

```

*** REGISTER DUMP ***
R0: 00000026 R1: 00000000 R2: cccccccc R3: cccccccc R4: 00000004
R5: cccccccc R6: cccccccc R7: cccccccc R8: cccccccc R9: cccccccc
R10: 0000000a R11: cccccccc AP: cccccccc FP: cccccccc SP: cccccccc
PC: 000000f5 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

```

Second dump. R10<-PSL:

```

*** REGISTER DUMP ***
R0: 00000045 R1: 00000000 R2: cccccccc R3: cccccccc R4: 00000004
R5: cccccccc R6: cccccccc R7: cccccccc R8: cccccccc R9: cccccccc
R10: 040000a4 R11: cccccccc AP: cccccccc FP: cccccccc SP: cccccccc
PC: 00000104 CONDITION CODES: N = 0 Z = 1 V = 0 C = 0

```

Figure 5: Example 1 output file OUT.DAT

Figures 4-5. The discussion below corresponds to the lines in the program listing (see Figure 3).

- 8-13** Until a macro library is developed for SIMON, all test programs must include the XOPC macro in order to make use of the extended operations.
- 32** Test program execution always begins at the initial entry point in the PSECT.
- 34-36** *Register 0* receives the address of the first byte of a string to be output. The length of the string is placed into *register 1*. Because XOPC 5 has not been previously executed, the result of this XOPC 6 is output of the string, TESTM, to screen (the default destination for debugging output).
- 38-40** The user specifies file OUT.DAT as the destination for subsequent debugging output. The address of the filename descriptor string is placed into *register 0*. Because the string is zero-terminated, the result of the .ASCIZ assembler directive, *register 1* need not contain the length of the string, but may be cleared instead. In this case the execution of XOPC 5 results in all subsequent output (dumps and debugging messages) being routed to the file OUT.DAT. The user is advised that any number of XOPC 5's can be used to re-route program output (see lines 61-62 below).
- 42-44** The output file is titled by the string, TITLE, whose address is moved into *register 0* and whose length is moved into *register 1*.
- 46-48** A message explaining the contents of the upcoming dump is similarly sent to the output file. Again *register 1* need only be cleared since MESS1 is zero-terminated.

- 50-52 *Registers 4 and 10 are marked, and a register dump is sent to the output file. Note also in this first dump (see Figure 5) that the values in registers 0 and 1 are those given in lines 46 and 47.*
- 54-59 *A second message and dump are sent to file. Register 10 is overwritten with the current value of the PSL.*
- 61-62 *Output is re-routed to screen by clearing register 0 and executing XOPC 5.*
- 64 *The screen dump is the same as the second dump to file, except for the value in the PC and that in register 0, which was changed in line 61.*
- 66-68 *The first line of Xs in the completion message was made by loading register 0 with the address of the string, X, and register 1 with a value (76) that will cause output to continue beyond the end of X and all the way through the storage at XX.*
- 69-72 *The second line was created by placing the string, MESS3, into the middle of XX storage.*
- 73-76 *The last line was also constructed by use of the MOV C3 instruction, which causes the first part of XX to be overlaid onto the portion that currently contains the MESS3 string. This effectively restores the initial state of XX to all Xs.*
- 77 *XOPC 1 halts program execution.*

IV.3 Example 2: Printer and Terminal I/O

The second example (see Figure 6) represents a single-user operating system that includes I/O system services and device drivers. The user program performs I/O operations on three peripherals (LP1, LP2 and TT0) by requesting services from the operating system; arguments are passed on the stack. When it becomes necessary to invoke certain privileged procedures, the mode is changed from User to Kernel. The operating system utilizes SIMON's features for channel assignment and device interrupt simulation (XOPCs 7 and 8, respectively). Also illustrated is the timing involved with I/O transfers. A rudimentary device database has been constructed to demonstrate possibilities for users of SIMON's I/O facilities. The contents of the various input and output files associated with execution of Example 2 are shown in Figures 7-11. The discussion below corresponds to the lines in the program listing (see Figure 6).

- 13-18 As always for programs that make use of extended operations, the XOPC macro is included first.

- 22-33 The four process stack pointers are initialized, and the AST level is set to 4.
 [The operating system's storage is located *after* the operating system and just before the user storage area (lines 335-384).]

- 37-41 The I/O Data Base (IODB) is initialized. Using the channel numbers as offsets from the beginning of IODB, the addresses of the associated Unit Control Blocks (UCBs) are placed into IODB. There is only one UCB for the terminal in this example (TT0); it is divided into portions for input and output.

```

0000 1 ;*****
0000 2 ;*****
0000 3 ;*****
0000 4 ;*****
0000 5 ;*****
0000 6 ;*****
0000 7 ;*****
0000 8 ;*****
0000 9 ;*****
0000 10 ;*****
0000 11 ;*****
0000 12 ;*****
0000 13 ;*****
0000 14 ;*****
0000 15 ;*****
0000 16 ;*****
0000 17 ;*****
0000 18 ;*****
0000 19 ;*****
0000 20 ;*****
0000 21 ;*****
0000 22 ;*****
0000 23 ;*****
0000 24 ;*****
0000 25 ;*****
0000 26 ;*****
0000 27 ;*****
0000 28 ;*****
0000 29 ;*****
0000 30 ;*****
0000 31 ;*****
0000 32 ;*****
0000 33 ;*****
0000 34 ;*****
0000 35 ;*****
0000 36 ;*****
0000 37 ;*****
0000 38 ;*****
0000 39 ;*****
0000 40 ;*****
0000 41 ;*****
0000 42 ;*****
0000 43 ;*****
0000 44 ;*****
0000 45 ;*****
0000 46 ;*****
0000 47 ;*****
0000 48 ;*****

EX. 2: PRINTER AND TERMINAL I/O
*****
Includes "system service" routines for: assignment of channels
I/O request processing
*****
Introduces pseudo-instructions: XOPC 7 (assign/deassign)
XOPC 8 (wait for interrupt)
*****
Demonstrates: printer and terminal I/O
timing for asynchronous I/O transfers
OS utilization of a rudimentary device database
*****
.psect TEST, long
.macro XOPC operand
    addb2 #operand, r1
    . = . - 3
    .byte ~ xfe
    . = . + 2
.endm
*****
Entry point -- OS
*****
.entry TEST, ~m<IV,DV>
    movl KSTK+47, r1
    mtpr r1, #x00
    movl ESTK+47, r1
    mtpr r1, #x01
    movl SSTK+47, r1
    mtpr r1, #x02
    movl USTK+191, r1
    mtpr r1, #x03
    movl PCB, r1
    movl #x04000000, 84(r1)
    mtpr r1, #x16
    ; initialize PCB
    ; ASTLVL of 4
*****
; Set up I/O Data Base and Unit Control Blocks
*****
    movl IOCB, r1
    movl UCBP1, LP1(r1)
    movl UCBP2, LP2(r1)
    movl UCBT0, TT0_OUT(r1)
    movl UCBT0+8, TT0_IN(r1)
    movl UCBP1, r1
    movl LP1_NAM, (r1)
    movb #~x1e, 4(r1)
    movl UCBP2, r1
    movl LP2_NAM, (r1)
    movb #~x1e, 4(r1)
    movl UCBT0, r1

```

Figure 6: Example 2 -- Printer and terminal I/O

```

61      00000C4B'EF      DE 0092      49      movl      TT0_OUT_NAM,(r1)      ;@ of descriptor
08 A1      04 A1 2E      DE 90 0099      50      movb      #~x2e,4(r1)      ;write, change mode code=11
      00000C3D'EF      DE 009D      51      movl      TT0_IN_NAM,8(r1)      ;@ of descriptor
      0C A1 2C      DE 90 00A5      52      movb      #~x2c,12(r1)      ;read, change mode code=11
      00A9      DE 00A9      53      ;-----
      00A9      DE 00A9      54      ; Set up System Control Block vectors
      00A9      DE 00A9      55      ;-----
51      000003E7'EF      DE 00A9      56      movl      SCB,r1
      17 51      DE DA 00B0      57      mtrpr     r1,#x17
      0104 C1      00000118'EF      DE 00B3      58      movl      LP1_IH,x104(r1)      ;LP1 interrupt handler rtn @
      0108 C1      0000011C'EF      DE 00BC      59      movl      LP2_IH,x108(r1)      ;LP2 int. handler
      0140 C1      00000120'EF      DE 00C5      60      movl      TT0_IH,x140(r1)      ;TT0 int. handler
      40 A1      000000FC'EF      DE 00CE      61      movl      CHMRTN,x40(r1)      ;change mode int. handler
      00C0 C1      00000124'EF      DE 00D6      62      movl      TIME,x40(r1)      ;timer int. handler
      00DF      DE 00DF      63      ;
      00000A16'EF      DE 00DF      64      movl      ISTK+47,r1      ;start on int. stack
      04 51      DE DA 00E6      65      mtrpr     r1,#~x04
      5E 04      DE DB 00E9      66      mfpr      #~x04,SP
      03C000A0 8F      DE DD 00EC      67      pushl    #x03c000a0      ; with user PSL
      00000E8B'EF      DE DF 00F2      68      pushal   USER      ; at USER
      00F8      DE 00F8      69      rel
      00F9      DE 00F9      70      ;-----
      00F9      DE 00F9      71      ; Change mode dispatcher
      00F9      DE 00F9      72      ;-----
      00F9      DE 00F9      73      .align long
50 8E      D0 00FC      74      CHMRTN: movl (SP)+,r0      ;pop code off stack
50 07      D1 00FF      75      cmpl     #7,r0      ;7 indicates printer driver routine
      06 13      D3 0102      76      beql     lp      ;
      50 0B      D1 0104      77      cmpl     #11,r0      ;11 indicates terminal driver routine
      08 08      D3 0107      78      beql     tt      ;
      02 0109      79      rel      ;if neither, return
      00000288'EF      17 010A      80      lp:      jmp     LP_DRIVER
      000002FB'EF      17 0110      81      rel
      82 tt:      jmp     TT_DRIVER
      83      17 0111      82      rel
      84      02 0117      83      rel
      0118      DE 0118      84      ;-----
      0118      DE 0118      85      ; Device ready interrupt handler routines
      0118      DE 0118      86      ;-----
      0118      DE 0118      87      .align long
      58      D6 0118      88      LP1_IH: incl r8      ;r8 counts LP1 interrupts generated
      02 011A      89      rel
      011B      DE 011B      90      .align long
      59      D6 011C      91      LP2_IH: incl r9      ;r9 counts LP2 interrupts generated
      02 011E      92      rel
      011F      DE 011F      93      .align long
      5A      D6 0120      94      TT0_IH: incl r10      ;r10 counts TT0 interrupts generated
      02 0122      95      rel

```

Figure 6 (continued)

```

96 ;=====
97 ; Timer interrupt handler
98 ;=====
99
100 .align long
101     mtrpr #x00, #x24
102     incl rll
103     mtrpr #x20, #x25
104     mtrpr #xdl, #x24
105     rel
106
107 ;disable timer interrupts
108 ;rll counts timer interrupts generated
109 ;20 timer units to NICR
110 ;set timer IE, ICR from NICR, run
111
112 ;=====
113 ; Assign channel system service
114 ;=====
115
116 .entry ASSIGN, m(r4, r5, r6, r7)
117     movl 4(AP), r4
118     movl 8(AP), r5
119     ;r4 <- @ of device name descriptor
120     ;r5 <- @ of block to receive channel#
121
122 ;=====
123 ; Search for device name descriptor match
124 ;=====
125
126     movl @UCBP1, r6
127     movw (r6), r7
128     cmpc3 r7, 8(r6), 8(r4)
129     beql 10$, r7
130     movl @UCBP2, r6
131     movw (r6), r7
132     cmpc3 r7, 8(r6), 8(r4)
133     beql 20$, r7
134     movl @UCBT0, r6
135     movw (r6), r7
136     cmpc3 r7, 8(r6), 8(r4)
137     beql 30$, r7
138     movl @UCBT0+8, r6
139     movw (r6), r7
140     cmpc3 r7, 8(r6), 8(r4)
141     beql 40$, r7
142
143     ;same for TT0 output
144
145     ;same for TT0 input
146
147     ;fall thru to here
148     ; error in device name
149     ; message output to user
150
151     asgn_err, r0
152     rll
153     xopc 6
154     ret
155
156 ; If channel available, assign it to requesting user
157 ;=====
158
159     blbs UCBP1+4, 50$
160     bisb #x01, UCBP1+4
161     movl #LPI, r1
162     brb 60$
163
164     ;if LPI's busy bit set, error message
165     ;else set busy bit
166     ;r1 <- channel# for LPI
167     ;finish up
168     ;same for LP2
169
170     blbs UCBP2+4, 50$
171     bisb #x01, UCBP2+4
172     movl #LP2, r1
173     brb 60$
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Figure 6 (continued)

```

27 00000C1B'EF  E8 01C5 146 30$:      UCBT0+4,50$      ;same for TT0 output
00000C1B'EF  01 88 147      #x01,UCBT0+4
51 00000024'8F  D0 01CC 148      #TT0_OUT,r1
      25 11 01D3 149      60$
      10 00000C23'EF  E8 01DC 150      UCBT0+12,50$      ;same for TT0 input
00000C23'EF  01 88 01E3 151      #x01,UCBT0+12
51 00000020'8F  D0 01EA 152      #TT0_IN,r1
      0E 11 01F1 153      60$
50 00000C8E'EF  DE 01F3 154 50$:      prev_asgn,r0      ;if busy bit set,
      51 D4 01FA 155      r1      ; error in assignment attempt
      08 11 01FF 156      xopc      ; message output to user
      50 D4 0201 158 60$:      r0
      65 51 D0 0203 159      xopc      ; r0 <- 0 informs SIMON that OS requests
      65 51 D0 0206 160      r1,(r5)      ; assignment of channel# in r1
      65 51 D0 0209 161      ret      ;pass channel# back to USER
      65 51 D0 020A 162      ;=====
00FC 020A 163      ; Request I/O system service
      56 04 AC D0 020A 164      ;=====
      57 08 AC D0 020C 165      .entry RIOW,<m<r2,r3,r4,r5,r6,r7>
      54 0C AC D0 0210 166      movl 4(AP),r6      ;r6 <- @ of longword with channel#
      55 10 AC D0 0214 167      movl 8(AP),r7      ;r7 <- r/w operation code
      55 10 AC D0 0218 168      movl 12(AP),r4      ;r4 <- @ of buffer
      55 10 AC D0 021C 169      movl 16(AP),r5      ;r5 <- number of bytes to transfer
      55 10 AC D0 021C 170      ;=====
      55 10 AC D0 021C 171      ; Validate parameters for I/O request and invoke change mode dispatcher ;
      55 10 AC D0 021C 172      ;=====
00000014'8F  D1 021C 173      cmpl (r6),#lp1      ;check range of channel#
      3A 19 0223 174      blss ch      ;if channel# arg. lower than hex 14
      66 D1 0225 175      cmpl (r6),#tt0_out      ; or higher than hex 24,
      31 14 022C 176      bgtr ch      ; error message
53 00000B67'EF  DE 022E 177      movl IODB,r3      ;get address of IODB
      53 66 C0 0235 178      addl2 (r6),r3      ;add channel# (offset) to get @ of UCB
      52 63 D0 0238 179      movl (r3),r2      ;r2 points to UCB
      2E 04 A2 E9 023B 180      blbc 4(r2),unasgn      ;if channel unassigned, error message
      57 03 D1 023F 181      cmpl #3,r7      ;input requested?
      07 13 0242 182      beql intst      ;verify input channel
      57 04 D1 0244 183      cmpl #4,r7      ;output requested?
      09 13 0247 184      beql outtst      ;verify output channel
      30 11 0249 185      brb oper      ;fall thru to here; invalid operation
      2B 04 A2 E0 024B 186      ;
      24 04 A2 01 024B 187      intst: bbs      ;if channel for output, error message
      04 A2 01 0250 188      ok      ;else, all ok, so change mode
53 04 A2 06 0252 189      outtst: bbc      ;if channel for input, error message
      53 02 0257 190      ok: extzv #2,#6,4(r2),r3      ;r3 <- change mode routine code
      53 BC 025D 191      chk  r3      ;invoke change mode dispatcher
      50 00000CC8'EF  DE 025F 192      ;
      51 D4 0266 193      movl chan_err,r0      ;if channel# out of range
      51 D4 0268 194      clrl r1      ; invalid channel number
      1A 11 026B 195      xopc 6      ; message output to user
      1A 11 026B 196      brb q_end

```

Figure 6 (continued)

```

50 0000D02'EF 51 DE D4 026D 197 unasgn: movl      unasgn_err,r0      ;if channel is not assigned,
0274 198 r1      198      ; error in I/O request
0276 199 xopc    199      ; message output to user
0279 200 q_end    200      ;
DE 0C 027B 201 oper:  movl      op_err,r0      ;if invalid operation requested,
0282 202 clrl    202      ; error in operation code
0284 203 xopc    203      ; message output to user
0287 204 q_end:  ret
0288 205 ;=====
0288 206 ; Device driver for UNIBUS lineprinters
0288 207 ;=====
0288 208 LP_DRIVER:
0288 209 cmpl      #lpl,(r6)      ;lpl channel?
0288 210 beql      lplout         ; yes, set up regs
0291 211 cmpl      #lp2,(r6)      ;lp2 channel?
0298 212 beql      lp2out        ; yes, set up regs
029A 213 xopc    213      ; fall thru to here
029D 214 ret
029E 215 lplout: movl      #lpl_csr,r2      ;r2 <- CSR @
02A5 216 movl      #lp1_dbr,r3      ;r3 <- DBR @
02AC 217 mtrpr     #90,~x25      ;timer setting (90 units for LP1)
02B3 218 brb      print         ;start I/O
02B5 219 lp2out: movl      #lp2_csr,r2      ;r2 <- CSR @
02BC 220 movl      #lp2_dbr,r3      ;r3 <- DBR @
02C3 221 mtrpr     #70,~x25      ;timer setting (70 units for LP2)
02CA 222 PRINT: xopc    22      ;dump #2 and #8
02CD 223 ;=====
02CD 224 ; Actions taken by driver and simulator to effect printer output:
02CD 225 ;
02CD 226 ; DRIVER
02CD 227 1. Disables interrupts
02CD 228 2. Tests DONE bit
02CD 229 --If clear (printer not ready)
02CD 230 a. Enables interrupts
02CD 231 b. rl <- channel#
02CD 232 c. Waits for int. (xopc 8)
02CD 233
02CD 234 (d. Goto 1)
02CD 235 --If set (printer ready)
02CD 236 e. Moves character to DBR
02CD 237 (f. Goto 2, if more)
02CD 238
02CD 239
02CD 240
02CD 241
02CD 242 10$: clrb      (r2)      ;clear IE bit in CSR (disable interrupts)
02CD 243 brb      30$           ;start at end of loop
02CF 244 20$: bitb      #x80,(r2)      ;test Done bit
02D1 245 bneq      40$           ; if clear, printer not ready
02D5 246 mtrpr     #x51,~x24      ; else set timer IE, ICR from NICR, run
02D7 247 movb      (r4)+,(r3)      ; send char. to printer (DONE cleared)
02DE 248

```

Figure 6 (continued)

```

ED 55      F4 02E1      248 30$:      sobgeq      r5,20$      ;continue if more to output
24 00      DA 02E4      249      mtptr      #x00,#x24      ;else disable timer interrupts before
11 11      D6 02E7      250      brb          ip_end      ; returning to user
55 55      D6 02E9      251 40$:      r5          ;undo effect of previous sobgeq
62 40 8F    D0 02EB      252      bisb      #x40,(r2)      ;set IE bit in CSR (enable interrupts)
51 66      D0 02EF      253      movl      (r6),r1      ;put channel# in r1 and
24 00      DA 02F2      254      xopc      8          ; wait for interrupt from controller
D3 D3      D1 02F5      255      mtptr      #x00,#x24      ;disable timer interrupts
11 11      D1 02F8      256      brb          10$          ;continue after device interrupt
04 04      D1 02FA      257      ip_end: ret
02FB      258      ;=====
02FB      259      ; Device driver for UNIBUS terminals
02FB      260      ;=====
02FB      261      ;=====
02FB      262      TT_DRIVER:
02FB      263      cmpl      #tt0_in,(r6)      ;tt0 input?
02FB      264      beql      tt0in      ; yes, set up regs
D1 13      D1 0302      265      cmpl      #tt0_out,(r6)      ;tt0 output?
D1 13      D1 0304      266      beql      tt0out      ; yes, set up regs
030B      267      xopc      2          ;fall thru to here
0310      268      ret
D0 04      D0 0310      269      tt0in:      movl      #tt0_csr,r2      ;r2 <- CSR @
D0 00      D0 0311      270      movl      #tt0_rbuf,r3      ;r3 <- RBUF @
11 11      D1 031F      271      brb          read_      ;
D0 00      D0 0321      272      tt0out:      movl      #tt0_csr,r2      ;r2 <- CSR @
D0 00      D0 0328      273      movl      #tt0_tbuf,r3      ;r3 <- TBUF @
032F      274      brb          write
0331      275      ;
0331      276      READ:      xopc      2          ;dump #4
0334      277      ;=====
0334      278      ; Actions taken by driver and simulator to effect terminal input:
0334      279      ;=====
0334      280      DRIVER
0334      281      1. Disables read interrupts      SIMON
0334      282      2. Tests RDONE bit      1. (RDONE bit unaffected)
0334      283      ---If clear (no char available)      2. (RDONE bit unaffected)
0334      284      a. Enables read interrupts      a. (RDONE bit unaffected)
0334      285      b. r1 <- channel#      b. r1 has from which device
0334      286      c. Waits for int. (xopc 8)      interrupt is expected
0334      287      ;      c. If channel unassigned or
0334      288      ;      invalid, error msg.
0334      289      ;      Else char read from file
0334      290      ;      into RBUF, RDONE set and
0334      291      ;      PC <- interrupt vector
0334      292      (d. Goto 1)
0334      293      ---If set (char available)
0334      294      e. Moves character from RBUF      e. Same (simply moves char)
0334      295      (f. Goto 2, if more)
0334      296      110$:      clrw      (r2)      ;clear IE bit in CSR (disable interrupts)
0F 11      B4 0334      297      brb          130$      ;start at end of loop
0F 11      0F 0336

```

Figure 6 (continued)

```

62 80 8F 93 0338 298 120$: bitb #x80,(r2)
63 FF 8F 91 033C 299 140$
42 13 0342 300 bneq #1,(r3)
84 63 90 0344 301 tt_end
EE 55 F4 0347 302 movb (r3),(r4)+
0039 31 034A 303 sobgeq r5,120$
55 D6 034D 304 brw tt_end
62 40 8F 88 034F 305 130$: brw r5
51 66 D0 0353 306 140$: incl #x40,(r2)
D9 11 0356 307 movl (r6),r1
0358 308 xopc 8
035B 309 brb 110$
035E 310
035E 311 WRITE: xopc 2
035E 312 ; dump #6 and #10
035E 313 ; Actions taken by driver and simulator to effect terminal output same
035E 314 ; as for printer output (see LP_DRIVER documentation)
035E 315
62 B4 035E 316 150$: clrw (r2)
0F 11 0360 317 brb 170$
8000 8F B3 0362 318 160$: bitw #x8000,(r2)
OE 12 0367 319 bneq 180$
64 00 91 0369 320 cmpb #x00,(r4)
18 13 036C 321 beql tt_end
63 84 90 036E 322 movb (r4)+,(r3)
EE 55 F4 0371 323 170$: sobgeq r5,160$
000F 31 0374 324 brw tt_end
55 D6 0377 325 incl r5
4000 8F A8 0379 326 180$: bitw #x4000,(r2)
51 66 D0 037E 327 movl (r6),r1
D8 11 0381 328 xopc 8
0384 329 brb 150$
0386 330
04 0386 331 tt_end: ret
0387 332 ;
0387 333 ; OS storage and initialization
0387 334 ;
000003E7 0387 .blk1 24
000009E7 03E7 .blk1 ~x180
00000A17 09E7 .blk1 12
00000A47 0A17 338 KSTK: .blk1 12
00000A77 0A77 339 ESTK: .blk1 12
00000AA7 0AA7 340 SSTK: .blk1 12
00000AB7 0AB7 341 USTK: .blk1 48
00000C07 0B67 342 IODB: .blk1 ~x28
00000C0F 0C07 343 UCBP1: .blk1 2
00000C17 0C0F 344 UCBP2: .blk1 2
00000C27 0C17 345 UCBT0: .blk1 4

```

;test RDONE bit
 ;if clear, no char. available to be read
 ;EOF?
 ; Yes, finish input
 ; else read char at RBUF into user buffer
 ; continue if more to input
 ; or finish up, if no more chars.
 ; undo effects of previous sobgeq
 ; set RIE bit in CSR (enable interrupts)
 ; put channel# in r1 and
 ; wait for controller interrupt
 ; continue after interrupt
 ; clear IE bit in CSR (disable interrupts)
 ; start at end of loop
 ; test TRDY bit
 ; if clear, terminal not ready
 ; null terminator?
 ; yes, finish output
 ; no, move char. for that line to TBUF
 ; continue if more to output
 ; or finish up, if no more chars.
 ; undo effect of previous sobgeq
 ; set TIE bit in CSR (enable interrupts)
 ; put channel# in r1 and
 ; wait for interrupt
 ; continue after interrupt
 ; incl. vectors for device levels 14 & 15
 ; I/O Data Base (contains ptrs. to UCBs)
 ; Unit Control Blocks for printers 1 & 2
 ; Unit Control Block for terminal 0

Figure 6 (continued)


```

55 51 45 52 20 4F 2F 49 20 21 0A 0D 0D02
42 20 54 4F 4E 4E 41 43 20 54 53 45 0D0E
44 45 54 4E 41 52 47 20 45 0D1A
73 69 20 6C 65 6E 6E 61 68 43 0A 0D 0D23
65 6E 67 69 73 61 20 74 6F 6E 20 0D2F
00 64 0D3B
55 51 45 52 20 4F 2F 49 20 21 0A 0D 0D3D
42 20 54 4F 4E 4E 41 43 20 54 53 45 0D49
44 45 54 4E 41 52 47 20 45 0D55
70 6F 20 64 69 6C 61 76 6E 49 0A 0D 0D5E
65 64 6F 63 20 6E 6F 69 74 61 72 65 0D6A
00 0D76
0D77
0D77
0D77
0D77
0000000D
0000000A
00000009
4F 52 50 20 54 53 45 54 20 09 09 0A 0D77
74 6E 69 72 50 20 2D 20 4D 41 52 47 0D83
4F 2F 49 20 6C 61 6E 69 6D 72 65 54 0D96
00 0A 0A 0A 0D 0DA2
0DA7
0DA7
00000003
00000004
0000000B
31 50 4C 00000DB3'010E0000' 0DAB
2C 31 50 4C 0A 0D 21 6F 6C 6C 65 48 0DB6
3F 74 69 20 74 27 6E 73 69 20 0DC2
0DCC
0DCC
00000016
00000DD0
32 50 4C 00000DD8'010E0000' 0DD0
21 32 50 4C 20 73 69 20 73 69 68 54 0DDB
0DE7
0DE7
0000000C
00000DEB
55 4F 5F 30 54 54 00000DF7'010E0000' 0DEF
54 49 5F 30 54 54 00000E06'010E0000' 0DFE
54 54 20 73 69 20 73 69 68 54 0A 0D 0E0C
21 69 48 0A 0D 30 0E7C
0E82
381 UNASGN_ERR: .asciz <CR><LF>"! I/O REQUEST CANNOT BE GRANTED"-
382 <CR><LF>/Channel is not assigned/
383 OP_ERR: .asciz <CR><LF>"! I/O REQUEST CANNOT BE GRANTED"-
384 <CR><LF>/Invalid operation code/
385 ;=====
386 ; USER storage and initialization
387 ;=====
388 RET = ~x0d
389 LIF = ~x0a
390 TAB = ~x09
391 ;
392 TITLE: .asciz <LIF><TAB><TAB>" TEST PROGRAM - Printer and "-
393 "Terminal I/O"<RET><LIF><LIF><LIF><LIF>
394 ; READBLK = 3 ;code for input
395 WRITEBLK = 4 ;code for output
397 ;
398 CHNUM1: .blk1 1
399 DEVNAM1: .ascid /LP1/
400 MESS1: .ascii /Hello!/<RET><LIF><LIF>/LP1, isn't it?/
401 MESS1_END: MESS1_END-MESS1
402 MESS1_SIZE = MESS1_END-MESS1
403 ;
404 CHNUM2: .blk1 1
405 DEVNAM2: .ascid /LP2/
406 MESS2: .ascii /This is LP2!/
407 MESS2_END: MESS2_END-MESS2
408 MESS2_SIZE = MESS2_END-MESS2
409 ;
410 CHINO: .blk1 1
411 CHOUT0: .blk1 1
412 DEVNAM0_OUT: .ascid /TT0_OUT/
413 DEVNAM0_IN: .ascid /TT0_IN/
414 BUFFER0: .blkb 100
415 MESS0: .ascii <RET><LIF>/This is TT0/<RET><LIF>/Hi!/
416 MESS0_END:

```

Figure 6 (continued)

FEB6 CF	DF	0F12	467 ; LP2 channel assignment	
FEB6 CF	DF	0F16	468 pushal chnum2	
F219 CF 02	FB	0F1A	469 pushal devnam2	
		0F1F	470 calls #2,assign	
	0C	0F1F	471 ; LP2 output	
FEB6 CF	DD	0F21	472 pushl #mess2_size	
04	DF	0F25	473 pushl mess2	
FEA1 CF	DD	0F27	474 pushl #writeblk	
04	DF	0F2B	475 pushal chnum2	
F2DA CF 04	FB	0F30	476 calls #4,riow	
		0F33	477 xopc 2	
	12	0F33	478 ; more TT0 output	
FF37 CF	DD	0F35	479 pushl #mess0_len	
04	DF	0F39	480 pushal mess0	
FEAC CF	DD	0F3B	481 pushl #writeblk	
04	DF	0F3F	482 pushal chout0	
F2C6 CF 04	FB	0F44	483 calls #4,riow	
57 07	90	0F44	484 ; end of user program	
		0F47	485 movb #7,r7	
		0F4A	486 xopc 2	
		0F4D	487 xopc 1	
			488 .end TEST	

			467 ; @ of block to receive channel#
			468 ; @ of LP2 descriptor
			469 ; call ASSIGN
			471 ; length of string
			472 ; @ of string
			473 ; code for output
			474 ; @ of channel# assigned
			475 ; call RIOW
			476 ; dump #9
			478 ; length of message
			479 ; @ of message
			480 ; code for output
			481 ; channel# should still be assigned
			482 ; call RIOW
			484 ; mark completion
			485 ; dump #11
			486 ; halt execution

Figure 6 (continued)

TEST PROGRAM - Printer and Terminal I/O

```

R0: 00000d77 R1: 00000000 R2: cccccccc R3: cccccccc R4: cccccccc
R5: cccccccc R6: cccccccc R7: cccccccc R8: 00000000 R9: 00000000
R10: 00000000 R11: 00000000 AP: cccccccc FP: cccccccc SP: 00000b66
PC: 00000ea4 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 00000007 R1: 00000014 R2: 00001fb8 R3: 00001fbc R4: 00000db6
R5: 00000016 R6: 00000da7 R7: 00000004 R8: 00000000 R9: 00000000
R10: 00000000 R11: 00000000 AP: 00000b52 FP: 00000b24 SP: 00000a3e
PC: 000002ca CONDITION CODES: N = 1 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 00000007 R1: 00000014 R2: 00000000 R3: 000079ea R4: cccccccc
R5: cccccccc R6: cccccccc R7: cccccccc R8: 00000015 R9: 00000000
R10: 00000000 R11: 00000000 AP: cccccccc FP: cccccccc SP: 00000b66
PC: 00000ec5 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 0000000b R1: 00000024 R2: 00001fd0 R3: 00001fd8 R4: 00000e0c
R5: 00000064 R6: 00000de7 R7: 00000003 R8: 00000015 R9: 00000000
R10: 00000000 R11: 00000000 AP: 00000b52 FP: 00000b24 SP: 00000b1c
PC: 00000331 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 0000000b R1: 00000020 R2: 00000000 R3: 00007a32 R4: cccccccc
R5: cccccccc R6: cccccccc R7: cccccccc R8: 00000015 R9: 00000000
R10: 0000001a R11: 00000000 AP: cccccccc FP: cccccccc SP: 00000b66
PC: 00000ef7 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***

R0: 0000000b R1: 00000020 R2: 00001fd0 R3: 00001fd4 R4: 00000e0c
R5: 00000064 R6: 00000deb R7: 00000004 R8: 00000015 R9: 00000000
R10: 0000001a R11: 00000000 AP: 00000b52 FP: 00000b24 SP: 00000b1c
PC: 0000035b CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

```

Figure 7: Example 2 output file TEST.OUT

```

*** REGISTER DUMP ***
R0: 0000000b R1: 00000024 R2: 00000000 R3: 00007a32 R4: cccccc R5: cccccc R6: cccccc R7: cccccc R8: 00000015 R9: 00000000 R10: 00000033 R11: 00000000 AP: cccccc FP: cccccc SP: 00000b66 PC: 00000f0f CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***
R0: 00000007 R1: 00000018 R2: 00001fc0 R3: 00001fc4 R4: 00000ddb R5: 0000000c R6: 000000cc R7: 00000004 R8: 00000015 R9: 00000000 R10: 00000033 R11: 00000000 AP: 00000b52 FP: 00000b1c SP: 00000b1c PC: 000002ca CONDITION CODES: N = 1 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***
R0: 00000007 R1: 00000018 R2: 00000000 R3: 00007a0f R4: cccccc R5: cccccc R6: cccccc R7: cccccc R8: 00000015 R9: 0000000b R10: 00000033 R11: 0000000c AP: cccccc FP: cccccc SP: 00000b66 PC: 00000f30 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***
R0: 0000000b R1: 00000018 R2: 00001fd0 R3: 00001fd4 R4: 00000e70 R5: 00000012 R6: 00000deb R7: 00000004 R8: 00000015 R9: 0000000b R10: 00000033 R11: 0000000c AP: 00000b52 FP: 00000b1c SP: 00000b1c PC: 0000035b CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

*** REGISTER DUMP ***
R0: 0000000b R1: 00000024 R2: 00000000 R3: 00007a0f R4: cccccc R5: cccccc R6: cccccc R7: cccccc R8: 00000015 R9: 0000000b R10: 00000044 R11: 0000000c AP: cccccc FP: cccccc SP: 00000b66 PC: 00000f47 CONDITION CODES: N = 0 Z = 0 V = 0 C = 0

```

Figure 7 (continued)

Hello!
LP1, isn't it?

Figure 8: Example 2 output file LP1.OUT

This is LP2!

Figure 9: Example 2 output file LP2.OUT

This is from me
and TT0!

Figure 10: Example 2 input file TT0.IN

This is from me
and TT0!
This is TT0
Hi!

Figure 11: Example 2 output file TT0.OUT

- 42-52** The UCBs are initialized. Because each terminal is considered as two logical devices, the UCB for each is twice as long (octaword) as that for each printer (quadword). The first longword of each logical UCB receives the address of the device name descriptor. The low order byte of the second longword is reserved for various bit fields which indicate the status of the device, the operation that is valid on it, and a code for use by the Change Mode Dispatcher routine. (The three most significant bytes are unused.) The idle/busy bit (bit 0) is 0 for idle and 1 for busy. The read/write bit (bit 1) is 0 for read and 1 for write. If the change mode code (bits <7:2>) is 7, a call to the printer driver routine is indicated; 11 (decimal) means that the terminal driver should be invoked. For example, in line 52 the hex value 2C is moved into the fourth longword of TT0's UCB (the second longword of the input portion). This bit pattern (00101100) defines the device as idle, capable of read operations, and associated with the terminal driver routine.
- 56-62** The System Control Block (SCB) is initialized with vectors for the three device interrupt routines, the change mode routine, and the timer interrupt routine.
- 64-69** The interrupt stack pointer is initialized so that proper test program execution may begin at USER as a result of the REI instruction.
- 424-427** In this example *registers 8-11* are used to count the device and timer interrupts. They are never used (and thus never saved) in the called system routines, so they reflect the true interrupt count in all register dumps.
- 428-433** The output file TEST.OUT is set up and written with a title and the first dump.

- 435-437 The user requests channel assignment for device LP1. The address of the block to receive the channel number is pushed onto the stack, and on top of it is pushed the address of the descriptor for the desired device. The `CALLS` instruction invokes the assign channel system service and indicates there are two arguments on the user stack.
- 108-110 Work registers are saved and the arguments are popped off the stack. *Registers 0-3* are not utilized, because the upcoming `CMPC3` instructions use them for temporary results.
- 114-129 The `ASSIGN` routine checks each of the four device names for a match with the user-supplied argument. Since the descriptors are the result of an `.ASCID` directive in all cases, the length of the string comparison is taken from the operating system's descriptors, and the comparison actually begins at the byte located one longword past the address of the descriptors.
- 131-134 If no device name match is found, the routine outputs a message for the user and returns.
- 138-153 If a match has been found, control is transferred to the appropriate local label for testing of the device's idle/busy bit. If it is set, indicating the channel has already been assigned, an error message is output (lines 154-156). Otherwise, the bit is set and *register 1* receives the channel number.
- 158-161 *Register 0* is then cleared, and `XOPC 7` is executed with the necessary parameters for assignment of the LP1 channel. The channel number is written into the user's buffer, and the `ASSIGN` routine returns to the user at the point immediately following the `CALLS` instruction.

439-443 The user pushes the following four arguments onto the stack before calling the Request I/O and Wait (RIOW) system service:

1. The length of the string to be output.
2. The address of the first byte of the string.
3. The code indicating this is an output request.
4. The address of the longword which has just received the channel number from the ASSIGN routine.

165-169 The RIOW routine saves the work registers and pops the arguments off the stack.

173-176 The first parameter checked is the channel number passed by the user. If it falls outside the expected range, an error message is output (lines 193-195) and RIOW returns.

177-179 Otherwise, the channel number is used as an offset into the IODB to get the address of the associated UCB.

180 The idle/busy bit is checked to make sure that the channel has been assigned prior to the attempted transfer. If not, an error message is output (lines 197-199).

181-185 The operation code passed by the user is checked to determine whether the requested transfer is for input or output. If neither, an error message is output (lines 201-203).

187-189 If the request is for input, the operating system needs to verify that this is indeed an input device. The same is done for output, and in either case, an error message is output (lines 201-203) if there is a discrepancy.

- 190-191 The bit field that contains the change mode code is extracted, and the change mode routine is invoked.
- 74-83 The operand of the CHMK instruction is popped and compared with known codes. In this case, a match is found at lines 75-76. Consequently, the JMP instruction in line 80 transfers control to the printer driver routine.
- 208-212 Since *register 6* still points to the channel number, it may be used here to find a match with printer channels.
- 215-216 Once a match is found, *registers 2* and *3* receive the actual values of the fixed addresses for LP1's controller registers—the CSR and the DBR, respectively.
- 217 All peripherals have a transfer speed of 80 μ sec./byte, so 90 timer units should be sufficient time to wait for an interrupt from LP1. Since the counter ticks up, which results in timer runout occurring on overflow, a negative value is always loaded in the NICR privileged register. The value in NICR will be used to initialize ICR repeatedly just prior to the MOV_B instruction in line 247 that sends a character to the data buffer register.
- 222 Dump #2 shows the change mode code in *register 0*, the controller register addresses in *registers 2* and *3*, the address of the message MESS1 in *register 4*, the length of MESS1 in *register 5*, the address of CHNUM1 in *register 6*, and the operation code for output (4) in *register 7*. (*Register 1* still has the channel number from the ASSIGN routine.)
- 223-258 The interaction between the test operating system and SIMON is documented in the example itself and explained in Chapter III, Section 5. Timing of the transfer is the only addition to the basic scenario.

- 246 When the printer is found to be ready for the next character, the driver allows a reasonable time limit (in this case, 90 units) for transfer completion. The hex value 51 is moved to the ICCS register, effectively enabling timer interrupts, setting ICR from NICR (-90), and activating the clock.
- 255 Timer interrupts are disabled each time after the device interrupt has been generated. In other words, if the printer controller determines that the output has taken place and the printer is ready for the next character, no timer interrupt should occur.
- 249 For the same reason, timer interrupts are disabled before returning to the user.
- 444 Dump #3 shows in *register 8* that 21 interrupts have been generated by device LP1 in the course of outputting 22 characters. This is the expected value, since the device is initially ready for output. Notice also that no timer interrupts have occurred; this verifies that 90 μ sec. is indeed sufficient for the transfer and the few surrounding instructions before timer interrupts are disabled.
- 446-452 Assignment of the channels for input and output on device TT0 is accomplished in exactly the same manner as the previously described assignment of the printer channel.
- 454-458 The following four arguments are pushed onto the stack prior to the user's request for terminal input:
1. The length (in bytes) of the buffer for input.
 2. The address of the buffer.

3. The code indicating this is an input operation.
 4. The address of the longword with the channel number assigned by the operating system.
- 173-191** Validation of parameters by RIOW and invocation of the change mode dispatcher is described above.
- 77-78** When the change mode routine pops the operand of the CHMK instruction, it finds a match with the code for the terminal driver routine.
- 262-274** The procedure for initialization of the work registers with the addresses of the appropriate controller registers is similar to that used by the printer driver. There is no timing illustrated in the terminal driver routine.
- 276** Dump #4 shows the change mode code in *register 0*, the controller register addresses in *registers 2 and 3*, the address of BUFFER0 in *register 4*, the length of BUFFER0 in *register 5*, the address of CHIN0 in *register 6*, and the operation code for input (3) in *register 7*. (*Register 1* still has the channel number from the ASSIGN routine.)
- 277-309** Documentation for the interaction between the test operating system and SIMON is sketched in the example and detailed in Chapter III, Section 5. The only addition to the basic scenario is the test for the end of file condition in Line 300. This is made possible by the nature of VAX C's standard I/O routines which return -1 when attempting to read beyond the last character in an input file.
- 459** Dump #5 shows in *register 10* that there have been 26 interrupts generated by the input routine. This is the expected value, since initially there is

no character available in RBUF. Thus an interrupt is generated for each of the 25 characters in file TT0.IN moved into RBUF. The final interrupt is generated when the standard input routine moves -1 into RBUF. Were the terminal driver not to check for EOF, *register 10* would show 100 interrupts generated, one for each byte in the buffer.

461-465 The four arguments placed on the stack before the call to RIOW for terminal output are similar to those in lines 439-443. The only difference is that in the former case the printer output request was for a string set up by the user, whereas the current request is for output of the characters just read in by the terminal driver routine. Consequently, the length and address of the buffer are the first longwords pushed onto the stack. Since the buffer has not been completely filled, the driver routine will check for a null-terminated string on output. (An alternative strategy might entail setting up an I/O status block to receive the actual count of characters input; this value could then be passed as a parameter.) Otherwise, validation of parameters, invocation of the change mode dispatcher, and terminal output are all essentially the same as previously described for printer output.

311 Dump #6 shows almost exactly the same values displayed in dump #4. *Register 2* has the identical CSR address, but *register 3* now contains the address of TBUF. The address of CHOUT0 is held in *register 6*, and *register 7* shows the output code. (*Register 1* still contains the input channel number used in the XOPC 8 operation, lines 307-308.)

316-331 The terminal driver outputs characters from the user buffer until the entire buffer is written or a null-terminator is encountered, whichever comes first.

- 466 Dump #7 shows in *register 10* that an additional 25 interrupts (hex 33 minus hex 1a) have been generated in order to perform TT0 output. Since the input routine stopped transferring characters into the user's buffer when EOF was encountered (after the 25th byte), and because storage in VAX C is initialized to zero (i.e., the input string is essentially zero-terminated), this is the expected value.
- 468-476 Assignment of the channel for LP2 and the subsequent request for output are identical to the procedures used in lines 435-443 for LP1.
- 221 The difference between LP1 and LP2 is assumed by the operating system to be a matter of speed: only 70 units are allotted for the supposedly faster LP2. This amount is intentionally insufficient in order to illustrate timing of I/O transfers by allowing SIMON to generate a predictable number of timer interrupts.
- 246 Timer interrupts are enabled, the ICR is loaded from the NICR, and the RUN bit is set exactly as for LP1 previously, except that NICR is -70 in this case.
- 247 When a byte is moved into the DBR for LP2, the smaller time value comes into play: it takes longer than 70 μ sec. to print a character on LP2, so the timer runs out, generating an interrupt.
- 100-104 The first thing that occurs in the timer interrupt handler routine is the disabling of timer interrupts. *Register 11* is incremented by one every time the routine is executed, that is, once for each timer interrupt generated. An additional 20 units are then placed into NICR; this is sufficient for the completion of the I/O transfer. The value d1 (hex) is moved into ICCS; this

sets the same bits as in line 246, but also sets the INT bit, which effectively re-enables timer interrupts.

255 As previously, timer interrupts are disabled after each device interrupt is generated (by XOPC 8). This is also done before returning to the user, that is, after the final character transferred (line 249).

477 Dump #9 shows in *register 9* that 11 device interrupts have been generated in the course of printing the 12-character message. This is expected, since the printer is initially ready. *Register 11* shows that 12 timer interrupts, one for each byte moved, have also been generated. This is the expected value, since each transfer takes more than the 70 μ sec. allowed by the operating system. The additional 20 μ sec. placed into NICR by the timer interrupt handler allows the successful completion of each byte transfer.

479-483 The output of a message to TT0 indicates that the correct channel number is still assigned (lines 450-452).

486 The final dump shows in *register 10* the additional 17 device interrupts generated in the course of outputting the 18-byte string, MESS0.

LIST OF REFERENCES

LIST OF REFERENCES

1. Baase, Sara. *VAX-11 Assembly Language Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. Kapps, Charles and Robert L. Stafford. *VAX Assembly Language and Architecture*, Prindle, Weber & Schmidt, Boston, MA, 1985.
3. Kelley, Al and Ira Pohl. *A Book on C*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1984.
4. Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
5. Kochan, Stephen G. *Programming in C*, Hayden Publishing Company, Inc., Hasbrouck Heights, NJ, 1983.
6. Levy, Henry M. and Richard H. Eckhouse, Jr. *Computer Programming and Architecture: The VAX-11*, Digital Press, Bedford, MA, 1980.
7. Marlino, Phyllis J. "A VAX Simulator for Developing Operating Systems," Master's Thesis, The University of Tennessee, Knoxville, 1985.
8. Pfleeger, C.P. and C.E. Hughes. "ASSIST-V: An Environment Simulator for IBM 360 Systems Software Development," *IEEE Transactions on Software Engineering*, November 1978, pp. 526-530.
9. *Programming in VAX-11 C*, Digital Equipment Corp., Maynard, MA, 1982.
10. *VAX Architecture Handbook*, Digital Equipment Corp., Maynard, MA, 1981.
11. *VAX Hardware Handbook*, Digital Equipment Corp., Maynard, MA, 1982.
12. *VAX Software Handbook*, Digital Equipment Corp., Maynard, MA, 1982.
13. Zygmunt, Beverly C. "Memory Management Facilities for a VAX Operating System Simulator," Master's Thesis, The University of Tennessee, Knoxville, 1985.

APPENDICES

APPENDIX A

I/O address space

	LP0 CSR	1FB0
	LP0 DBR	1FB4
	LP1 CSR	1FB8
	LP1 DBR	1FBC
	LP2 CSR	1FC0
	LP2 DBR	1FC4
	TT0 CSR	1FD0
	TT0 TBUF	1FD4
	TT0 RBUF	1FD8
	TT1 CSR	1FE0
	TT1 TBUF	1FE4
	TT1 RBUF	1FE8
	TT2 CSR	1FF0
	TT2 TBUF	1FF4
	TT2 RBUF	1FF8
		2000

SYSTEM SPACE ↑

P1 SPACE ↓

Figure 12: System space addresses of controller registers

APPENDIX B

I/O controller registers

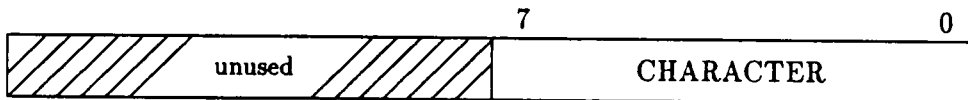


Figure 13: Data buffer register (printer and terminal)

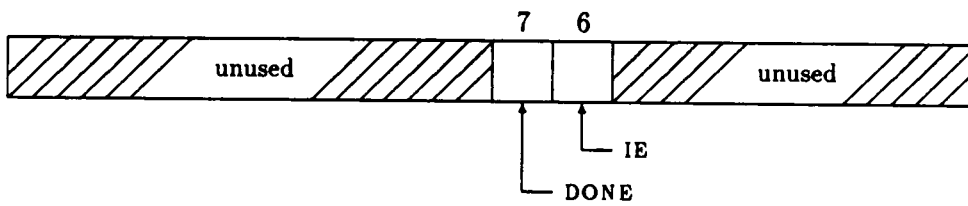


Figure 14: Printer control/status register

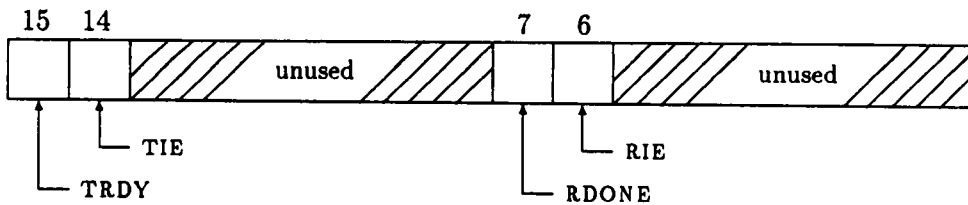


Figure 15: Terminal control/status register

APPENDIX C

I/O device parameters

<i>Device</i>	<i>Filename</i>	<i>Channel Number</i> (hex)	<i>SCB Offset</i> (hex)
Printer 0	LP0.OUT	10	100
Printer 1	LP1.OUT	14	104
Printer 2	LP2.OUT	18	108
Terminal 0 input	TT0.IN	20	140
Terminal 0 output	TT0.OUT	24	140
Terminal 1 input	TT1.IN	28	144
Terminal 1 output	TT1.OUT	2C	144
Terminal 2 input	TT2.IN	30	148
Terminal 2 output	TT2.OUT	34	148

Table 1: Filenames, channel numbers and SCB offsets for I/O devices

APPENDIX D

Summary of extended operations

The table below summarizes the extended operations available to users of SIMON. The XOPC macro (see Appendix E) must be included for these operations to work. For a complete description of the values expected in certain registers or the location counter, the user is advised to consult Chapter III and the two previous theses [7,13]. Chapter III includes descriptions of XOPCs 5, 6, 7 and 8; XOPCs 1 and 2 were introduced in [7]; and XOPCs 0, 3 and 4 are explained in [13].

<i>XOPC operand</i>	<i>Extended operation</i>
0	Flag creation of P1 and P0 addresses
1	Halt program execution
2	Dump registers, PC and condition codes
3	Update disk (page out)
4	Update memory (page in)
5	Route debug output
6	Output debugging message
7	Assign channel
8	Wait for (generate) device interrupt

Table 2: Extended operations using XOPC macro

APPENDIX E

Generic test program form

Described below are guidelines for a generic form of test operating systems. Figure 16 illustrates these general principles. For a more complete explanation of SIMON's internal workings, the user is advised to consult Chapter III and the previous two theses [7,13].

The user may divide test programs into "operating system" and several "user" logical portions, but the current version of SIMON allows only one PSECT with a maximum size of 65,535 decimal bytes. The program name in the initial `.entry` and the `.end` assembler directives must match, but the `.psect` name may be different. The VMS file in which the test program resides must match the `.psect` name, and this name cannot be "MAIN" or more than eight characters long. (A name only one character long is not a good idea, since SIMON must search the image file for the name.) When SIMON is executed, the user is prompted to enter the filename (including the `.exe` extension) of the image file to be tested.

In order to make use of the extended operations (see Appendix D), test programs must begin with a macro (immediately after the `.psect` directive) that establishes the opcode FE (hex).

At the initial entry point, it is suggested that an operating system set up the stack pointers, initialize any Process Control Blocks, and define the AST level in the PCB to a sufficiently high value (4 or above). The addresses (aligned on a longword boundary) of any interrupt handlers should be loaded into the appropriate vectors in the System Control Block. "User" process execution is effected by setting SP to point

```

.psect name1,long
.macro XOPC operand
addb2 #'operand,r1
. = . - 3
.byte ^xFE
. = . + 2
.endm

```

operating system storage

```

.entry name2,^m<IV,DV>
:
:
pushl #^x03C000A0
pushal USER1
rei
:
:

```

operating system

operating system storage

user storage

```

USER1:

```

user1

user2

:

usern

```

.end name2

```

Figure 16: Generic program form

at the interrupt stack and pushing the user PSL, 03C000A0 (hex), and the address of the user program onto the stack; the subsequent REI instruction transfers control to the user portion of the test program.

Storage for the operating system may be located before and/or after the operating system itself. When memory management facilities are enabled, it may be necessary to move parts of such storage to the end of the operating system in order to locate the initial entry point on a page boundary. To SIMON, system space extends from address zero to hex 1FFF (with I/O controller registers in the range, hex 1FB0 to 1FFF). The P1 region extends from hex 2000 to 3FFF; the P0 region grows upward from hex 4000. Delimiting P1 and P0 space is accomplished by adjusting the location counter to hex 4000; this should be followed immediately by the XOPC 0 extended operation. For a complete explanation of memory management facilities, see [13]. See Appendix A for further information concerning I/O space.

VITA

David L. Russell was born on November 1, 1950, in Oak Ridge, Tennessee, where his parents, both members of the National Academy of Science, still live and work as research geneticists at Oak Ridge National Laboratories. He attended Oak Ridge public schools and graduated from Oak Ridge High School in June 1968. He attended Harvard College, Cambridge, Massachusettes, St. John's University, Collegeville, Minnesota, and received a Bachelor of Music from the University of Tennessee, Knoxville, in August 1981. During the period from 1968 to 1984, he was a teacher/performer/composer of music.

While pursuing a Master of Arts in Music History and Literature as a Graduate Teaching Assistant at the University of Tennessee, Knoxville, Mr. Russell had occasion to study computer-aided music education. This interest led to his enrollment in the Master's program in Computer Science at UTK in September 1984. From January to August 1985, he taught introductory classes in programming as a Graduate Teaching Assistant. In 1986 he joined the Association for Computing Machinery. Mr. Russell received the Master of Science with a major in Computer Science in December 1986.