A portable Modula-2 operating system: SAM2S

by LARRY D. WITTE and ARIEL J. FRANK

State University of New York at Stony Brook
Stony Brook, New York

ABSTRACT

The Stand-Alone Modula-2 System (SAM2S) is a portable, concurrent operating system and Modula-2 programming support environment. It is based on a highly modular kernel task running on single process-multiplexed microcomputers. SAM2S offers extensive network communication facilities. It provides the foundation for the locally resident portions of the MICROS distributed operating system for large netcomputers. SAM2S now supports a five-pass Modula-2 compiler, a task linker, link and load file decoders, a static symbolic debugger, a file, and other utility tasks. SAM2S is currently running on each node of a network of DEC LSI-11/23 and Heurikon/Motorola 68000 workstations connected by an Ethernet. This paper reviews features of Modula-2 for operating system development and outlines the design of SAM2S with special emphasis on its modularity and communication flexibility. The two SAM2S implementations differ mainly in their peripheral drivers and in the large amount of memory available on the 68000 systems. Modula-2 has proved highly suitable for writing large, portable, concurrent and distributed operating systems.
INTRODUCTION

The MICROS project is exploring ways to organize networks of thousands of computers (netcomputers) to solve large problems. Its main goals are to develop a portable distributed operating system (MICROS) that can efficiently control many different netcomputers and to produce cost-effective netcomputers that provide high throughput for large classes of applications, that extend easily to form more powerful systems, and that are always available to users at acceptable processing rates even after component failures.

A netcomputer consists of many computer nodes, each with its own primary memory, physical clock, and attached peripherals. Nodes are embedded in a network of communication links over which messages are exchanged to share data from the separate memories. A global decentralized operating system, with some code resident in every node, unifies the nodes into a single computer system. The global operating system strives to provide netcomputer users with a powerful computing facility that can be accessed as a single virtual multiprocessor without regard to physical locations within the network.

Modula-2 is a high-level, general programming language that facilitates the building of simple and practical programming support systems. The Stand-Alone Modula-2 System (SAM2S) is a portable, highly modular concurrent operating system. SAM2S was developed initially to assess Modula-2 as a language for writing large systems and to provide portable software for Modula-2 programming support work stations. SAM2S was first developed for DEC LSI-11 work stations and later ported to Heurikon/Motorola 68000 work stations. When replicated in every node of a netcomputer, SAM2S forms the locally resident portions of the MICROS distributed operating system.

The next section of this paper discusses features of Modula-2 systems that are important for writing operating systems. The main section describes the design principles for SAM2S and the organization of both SAM2S implementations. The last three sections give the current status of SAM2S and MICROS, future research plans for MICROS, and conclusions reached in using Modula-2 to develop and port SAM2S.

MODULA-2 SYSTEMS

Modula-2 is a concurrent programming language convenient for both system and user applications. It is an improvement on Pascal, based on the best features of Modula and MESA. It was designed to be suitable both for high-level programming in an architecture-independent manner and for low-level programming of architecture-dependent aspects such as machine access and input/output (I/O) device handling.

Modula-2 shares most of its conceptual goals and programming language features with Ada, but is much simpler and more comprehensible. Modula-2 systems are simple but practical. They require only a small underlying run-time kernel, typically including fewer than 1,000 lines of assembly code. The module concept is central to Modula-2, as reflected by its name (MODUlar programming LAnguage). Most system facilities, including I/O operations, are provided by standard library modules. The modularization facilities, extensible high-level language interfaces, low-level machine access capabilities, and coroutine-based concurrency mechanisms all provide an effective environment for modern software systems.

Modularization Facilities

For true modularity, it is essential to be able to refer to another module knowing only the abstract properties contained in its specification. Using modular design, several programmers can develop different modules independently. True isolation of module design decisions can hide implementation details and aid in program readability, verification, and maintenance. Modules can be compiled, tested, debugged, and updated without unpredictable effects on other modules. Having separate modules is especially important for large research projects using many, and often inexperienced, programmers. For example, in three years more than 30 students have together contributed more than 70,000 lines of working Modula-2 code to the MICROS project.

Modularization facilities are strongly supported in Modula-2. A module is a program component, normally represented syntactically by a pair of definition and implementation modules. Each module pair provides a separate reference scope for a collection of logically related declarations, procedures, and data. All references across module boundaries involve a matching pair of explicit export and import declarations. All public declarations in the definition module define the module interface to module users. An implementation module provides the body of code implementing the defined interface. Each syntactic module can reside in an independent file and can be separately compiled. However, separate compilation does not mean independent compilation, since strong type checking is enforced between modules. Separate modules can be managed in libraries to enhance software reusability and to encourage software growth through accretion.
Extensible Language Interfaces

A common problem of high-level languages is restrictive linguistic constructs. Modula-2 avoids language inflexibility by excluding all process control, storage allocation, exception handling, and I/O facilities from the language definition. Instead, these facilities are easily provided by extensible language interfaces using standard library modules. The burden of supporting extensions thus shifts from the language to the library, allowing the language, compiler, and language runtime support system to remain small. Users of machines with small memories can configure Modula-2 systems with only the interfaces required by their applications. Modula-2 also supports procedure-valued parameters and variables. Procedure-valued variables are useful for process control and for passing functions to generic procedures. The extensible handling of concurrency in Modula-2 provides an example of these benefits.

Low-Level Machine Access

Since it was designed as a systems programming language, Modula-2 provides facilities for low-level machine access. The SYSTEM pseudo-module embedded in the compiler provides the main machine-dependent interface by encapsulating hardware data sizes and address formats. Machine-dependent operations include pointer and address arithmetic, relaxed type checking, explicit type transfer, and manipulations of bit sets. For I/O interfacing, Modula-2 allows access to peripheral device registers residing at fixed memory locations, specification of hardware priorities, and I/O transfers to support interrupt triggered context switching.

Concurrency

Concurrency provides for logically parallel threads of execution that can cooperate synchronously or asynchronously. The process is the fundamental unit of sequential execution that is combined for concurrent execution. Modern programming languages distinguish the logical process from the physical processor, allowing various mechanisms for allocating processors to processes. All concurrent programming languages provide some mechanism(s) for process interaction.

There are two contrasting trends in concurrent programming languages. Some languages include many high-level linguistic constructs for process interaction, directly providing a user-oriented interface. However, these languages tend to be large and complex, to embed fixed constructs and rigid interaction mechanisms, and to require an elaborate runtime system. Examples include Ada, Argus, and MESA. In contrast, some languages provide only a few lower-level constructs for process interaction, from which flexible higher-level mechanisms can be built. These languages tend to be simple and comprehensible, to support various types of process interaction, and to require only a modest runtime system. Examples include Edison, Modula-2, and SR.

Modula-2 provides only a low-level coroutine mechanism to support concurrent execution. However, coroutines can be used to model the multiprocess scheduling and execution facilities of any single processor system. In Modula-2, any parameterless procedure can be executed as a process. At the lowest system level, processes are declared by a NEWPROCESS system call and activated with TRANSFER and IOTRANSFER calls. Via this coroutine mechanism, control switching can be explicit for transfer and I/O transfer calls or implicit as a result of I/O interrupts. The conceptual unification of planned process switches and forced interrupt transfers provides a clean mechanism on which to base higher-level mechanisms for process concurrency and synchronization. Users directly interact with higher-level concurrency models, such as a time-slicing mechanism for a uniprocessor.

STAND-ALONE MODULA-2 SYSTEM (SAM2S)

The originally released Modula-2 system (M2RT11) is a Modula-2 programming support environment targeted for DEC PDP-11 and LSI-11 systems and dependent on DEC’s RT-11 operating system for services such as file access, editing, and I/O handling. During the summer and fall of 1981, the MICROS research group developed the Stand-Alone Modula-2 System (SAM2S) for the LSI-11 by writing standard Modula-2 library modules for all the RT-11 services used by M2RT11. SAM2S was first developed mainly to find out whether Modula-2 was adequate for producing entire operating systems for programming support work stations. It has proved more than adequate. The original version of SAM2S actually runs slightly faster than M2RT11, primarily because all service routines are kept resident by SAM2S and not paged from disk as for RT-11.

The small memory (60 Kb) addressable by the LSI-11 limits the size of tasks run under the LSI-11 version of SAM2S to about 30 Kb. In practice, this means that we can edit and compile simple modules under SAM2S, but must rely on M2RT11 to change large modules such as the passes of the compiler itself. Since the two systems run on the same processor with exactly the same file format, switching from one to the other requires only a single “boot” command. It was the lack of memory space on the LSI-11, especially as we began to write and test communication software, that led us in 1983 to port SAM2S to work stations based on Motorola 68000 processors.

SAM2S has been designed to provide both a stand-alone programming support environment and a module library that can be the basis for the locally resident portions of the decentralized MICROS operating system. SAM2S is a concurrent system, but not a distributed one. However, it emphasizes flexibility in communications, whether on one machine or many, and includes Ethernet drivers and Xerox communication protocols.

SAM2S Design Principles

SAM2S is a highly portable, independent Modula-2 programming support environment based on a modularized kernel task running on a process-multiplexed microcomputer.
The design for SAM2S uses many advanced features of Modula-2. SAM2S benefits heavily from high-level device drivers and from modularization facilities that allow definition of hidden and hierarchical type managers as well as layered tasks for both system and users.

Hidden type managers

The existence of a module facility does not automatically ensure software modularity. Some programming standards are needed. For example, SAM2S code avoids both exported variables and nested modules. Module structuring in SAM2S is based on abstract data types, encapsulation concepts, and information-hiding principles.13,14

A module should be designed to encapsulate one abstract data type, which imposes modular structure on data and characterizes all allowed operations and values. Each instance of a type is referred to as an object. The procedures in a module that define all operations on an object collectively form the type manager. Basic operations include creation, manipulation, and destruction of objects.

Hidden types in Modula-2 are declared only by name in the type definition module. The component substructure for the type is fully declared only in the implementation module. Hidden type objects are completely encapsulated. Only operations defined by their type manager can access or change them. Other modules do not know their structures and cannot directly manipulate their components. That hidden objects must contain all their own state information also allows their type manager to synchronize accesses efficiently. Process blocking is reduced by enforcing synchronization on individual shared objects only, rather than on the shared manager itself, as is done using monitors.

Hierarchical type managers

Two goals of type manager design are simplicity and generality. Simplicity demands a small module with a clean and readable structure. Generality means that each type manager should support an elaborate type with widely useful operations. These two goals usually are in conflict. Both goals can be achieved using policy/mechanism separation15 and hierarchical type managers.

With policy/mechanism separation, lower levels of the system focus on providing general mechanisms that are as devoid as possible of embedded control decisions, so higher levels have maximum flexibility in choosing policies. Type managers should be designed to adhere to the type policy determined by indicators within the object state. Their mechanisms must accommodate all allowed type policies.

With hierarchical type managers, a first-level manager handles the basic version of a general type. A second-level type manager uses the facilities of the first-level manager to offer more advanced operations and to support an extended type. Even higher-level managers may be defined. An example is a process type manager, which provides basic operations like create, suspend, and resume. A more advanced manager uses additional information in each process object for synchronization.

High-level message-oriented device drivers

Physical and logical devices can be regarded as hidden types requiring storage access, data transfer, and synchronization facilities. Physical device drivers manage the details for peripheral devices. Logical device modules support available I/O formats for character and block-oriented devices and interact with physical device drivers. Each device module is an active type manager, since it contains one or more processes for device handling and user interactions. In SAM2S, the only processes that are genuinely concurrent are physical device processes that do real I/O by using the IOTRANSFER mechanism. All other processes are preemptively multiplexed by a time-slicing scheduler.

Device modules are written in high-level Modula-2 code, instead of assembly language, greatly easing system maintenance. Each device driver requires about 500 lines of Modula-2 code. Device drivers use low-level machine access facilities to manipulate device registers. Depending on the exact configuration of SAM2S, I/O service requests may be made directly through procedure calls, locally by interprocess messages using simple queue interfaces, or remotely through socket interfaces by messages from processes on other computers. Although the message interfaces for I/O are slower than direct-entry procedures, they are extremely flexible and make it easy to reconfigure SAM2S for differing devices.

Layered tasks

A task, or concurrent program, is a software structural unit built from one or more modules. Each task is a separate loading unit. Processes within a task are scheduling units that execute on a single host. Processes communicate and synchronize by passing messages and sharing objects. Linkers, editors, filers, and debuggers are common library tasks.

In Modula-2 systems, a task is specified by the hierarchy of module import dependencies that start from the main module. The modules forming a task are linked together as an overlay onto a host. Normally, the operating system kernel forms the basis for all other task overlays. Other tasks are loaded in layers above it and access its modules by imported procedures. Where there is a system configuration choice of different implementation modules for the same type manager, one has to be specified. Linking the chosen modules automatically selects any library modules that they import. Modules that are needed by higher-level tasks, but have already been provided for lower-level tasks, are not linked again.

The main program module, base task, and selected module choices are presented to the SAM2S task linker to produce a relocatable load unit. The linker manages the module and task libraries, type-checks intermodule interfaces, and places the resulting load file in the task library. The file contains information for controlling task loading.

SAM2S supports the open system concept, which blurs distinctions between system and user tasks to enhance system
Kernel support modules

Low-level kernel modules are machine-dependent. In SAM2S/LSI11, the LSI11 module encapsulates the architecture of the LSI-11/23 microprocessor. It defines machine-specific trap and peripheral addresses that are also used by the assembly subkernel. The MC68000 module in SAM2S/68000 provides similar trap and peripheral access services. On each system, the Exceptions trap handling module is closely coupled to the basic trap facilities in the low-level machine module.

The SystemTypes module exports basic constant and type declarations used throughout the system. Grouping common declarations into a single module lessens the number of interfaces that have to be imported by most modules. Memory management, including compaction, is provided in the kernel by the Memory module. Available memory is managed as a dynamic heap, using a circular first-fit algorithm.

The structured data type modules are hidden type managers for abstract data structures maintained by the kernel and by user tasks. For example, the Lists module can efficiently manage LIST objects created as a regular list, a descending or ascending priority list, a circular list, or a stack. The Maps module manages MAP objects, which are dynamically varying lists that associate an index for a hidden object with a unique identifier. Sets and caches of network communication addresses are maintained by the AddressSets and Caches modules. Other structured data types include queues and character buffer rings.

Process interaction modules

Process interaction facilities are provided by a hierarchy of type managers. The Processes module provides the basic PROCESS type and standard operations, including process creation, blocking, resumption, suspension, and termination. Priority lists are used for process scheduling. Spawning of processes forms tree hierarchies used for process control and termination. Processes can be synchronized by use of the Signals, Gates, and Semaphores modules. Signals are events or conditions on which processes can wait and about which they can be notified. A SIGNAL object manages a list of processes queued on the associated event. A GATE object is used as a binary semaphore to support mutually exclusive access to shared objects or code sections. It can be used to implement monitors. More elaborate synchronization can be achieved with the general SEMAPHORE type that provides conditional blocking of processes. Other synchronization types include event counts and sequencers.

The Names and Groups managers provide services for registration and lookup of symbolic names. The NAME type associates the name string for an object with its attributes, access capabilities, and unique identifier. A capability contains addressing information and possibly object access rights. To provide for hierarchical name spaces, groups of names are managed in tree directories. The GROUP type supports none, one, or more associated NAMEs. Symbolic names can be searched for on the top level of any specified subtree or recursively throughout the subtree.
Communication facilities are provided by another hierarchy of managers. The Ports module uses Queues to support either First-In-First-Out (FIFO) or priority ports for sending and receiving local messages. It controls port access rights, message forwarding, and conditional passing of messages. For network communication, the NetTypes module declares common addresses and services. The Sockets type manager provides location-independent general message transfer services either locally, within the same host computer, or remotely, between processes on different hosts. A SOCKET is a bidirectional port used as an end-address for sending and receiving messages between processes. The Transport and Routes modules provide for forwarding of packets over the communication subsystem.

To provide type uniformity for messages, Ports and Sockets directly manage carriers, which are standard headers for messages. Information in each carrier includes source and destination addresses, a unique message identifier, the message type, and a pointer to the message itself, if it exists. Empty carriers can be posted in ports for incoming messages. The Messages module provides packaging facilities for marshaling and unmarshaling data into and from packets used for remote procedure calls.

I/O service modules

I/O services are provided on three levels of abstraction: physical, logical, and virtual. There are user interfaces at the virtual level for file and terminal services. The virtual level passes user requests as procedure calls or messages to the appropriate I/O format module on the logical level. The logical level interfaces with the appropriate physical I/O driver by messages using either communication or queue services. The DeviceTypes module declares constants and types used by the physical and logical device modules. At initialization, device drivers configured in the Kernel task register their existence with the name manager to give users dynamic access to I/O services. Device modules request I/O services and post results by using the IOREQUEST type as a standard message.

In timing experiments, we have found that serving local I/O requests through communication sockets takes about twice as long as through queue interfaces. As a compromise between speed and flexibility, we ordinarily use sockets for higher, logical-level I/O interfaces and faster queues for the lowest, physical-level interfaces. We have not yet found need for remote calls to low-level physical I/O drivers.

Examples of physical drivers for SAM2S/LSI11 QBUS-based devices include a DEC DLV11J serial driver, RX02 and RP02 disk controllers, and a QE3C400 Ethernet controller. The DLV11J driver manages up to four serial lines and provides type-ahead terminal handling, using a RING object for a character buffer. The RX02 floppy disk controller handles two diskette drives. The RP02 module handles eight logical partitions of a 169-Mb Fujitsu Winchester disk. QE3C400 interfaces to one or two 3COM Ethernet boards used for netcomputer communication.

Functionally similar physical drivers exist for MULTIBUS-based devices in SAM2S/68000. The SCCZ8530 module drives the Zilog serial communications chip on the Heurikon HK-68K board. It also handles up to four serial lines. Currently, SAM2S/68000 has a controller for a Primax 70-Mb Winchester disk. Controllers for several other disks (Vertex, Micropolis) and the four direct memory access (DMA) ports on the Heurikon board are under development. The DMA module will support efficient copying of blocks of data for both disk and Ethernet facilities. ME3C400 provides a dual Ethernet interface for SAM2S/68000.

Logical device modules include handlers for serial terminals and disk formats. The logical devices are independent of actual physical interfaces. The RT11Files module handles RT11 directory and file formats. A UnixFiles module for Unix format files is currently being written. ADM31Lines and VT101Lines control ADM31 and VT101 terminals.

The virtual-level modules provide abstract services to their clients. The Times module provides time and timeout facilities, using the KW11L and Z8536CIO physical clock drivers in SAM2S/LSI11 and SAM2S/68000, respectively. The Files and Lines modules provide an abstract file and serial line interface for users. These modules direct user requests to the proper logical device modules.

Node control modules

The ResidentMonitor executive module receives control after kernel initialization and monitors the execution of user tasks. It interacts with the command interpreter and kernel loader to load, execute, and terminate relocatable user tasks on SAM2S. At present, a single user code file at a time may be run. The file name serves as a load command.

SAM2S user-level tasks

Additional library modules are available for tasks run at the system user level. Some allow changes to file names and options. File I/O can be abstracted into character I/O by using Streams. The Strings module supports standard operations, such as extract and concatenate, on strings represented as character arrays. InOut provides transparent access to characters on either files or terminals. SAM2S currently supports a five-pass Modula-2 compiler, a task linker, link and load file decoders, a mini-core debugger, a static symbolic debugger, a file, an import dependency charter, and other utility tasks.

SAM2S/MICROS STATUS

SAM2S has recently been ported from DEC LSI-11/23 computers to the Heurikon version of Motorola 68000 single-board systems. Both LSI-11s and 68000s are combined in a heterogeneous network of nodes connected by ten million bit-per-second Ethernet links. Between nodes, SAM2S uses flexible communication techniques including location-independent sockets, remote-procedure-call interfaces for file services, and standard Xerox Network System (XNS) packet transport protocols.12
The five existing network nodes, shown in Figure 2, are used as programming support work stations controlling one to three terminals (T) each. One LSI-11 (node 2) controls a color monitor (M) that shows Ethernet traffic among network nodes. Packet glyphs move nearly in real time, with just enough slowing for humans to see. Both LSI-11 systems (nodes 1 and 2) have dual floppy disk drives (F), but two 6800 work stations (nodes 4 and 5) have no attached disks. The flexibility of interfaces in SAM2S allows both diskless 68000 systems to be booted remotely with files supplied either by the other 68000 (node 3) from its Priam Winchester disk (W) or by one LSI-11 (node 1), from its Fujitsu disk (W) or its floppy disk (F). Individual application programs also are remotely loaded into any of the 68000 systems.

Besides the original LSI-11 compiler from Wirth at ETH-Zurich and a VAX/VMS Modula-2 system from the University of Hamburg, there are several locally developed Modula-2 compilers that are being used to port SAM2S to other machines. The most heavily used is a VAX/UNIX cross-compiler that produces 68000 machine code. A recent translation of this compiler from Pascal into Modula-2 allows compilation directly on VAX-11 UNIX systems and generating code for the Intel 8086 and 80186 processors. In addition, there are about 30,000 lines in LSI-11 compiler, linker/loader, and debugging utilities obtained from Wirth. More than 50,000 lines of high-level code have been added to MICROS in the last year.

RESEARCH PLANS FOR MICROS

The major recent theoretical work in the MICROS project has been in analyzing ways in which to implement and to use multicast communication within dynamic groups of computers in large networks, especially ones linked by grids of horizontal and vertical Ethernets. Group communication techniques developed for Ethernet systems should be applicable to many distributed system environments, even those using dedicated links. The research has included analysis of efficient computer mechanisms to maintain distributed lists characterizing dynamically changing groups and to multicast packets within groups. Efficient communication in large groups can require spanning trees of multicast routing information. Single messages multicast to processes scattered over a network can follow tree branches and be copied at each fork.

A modular, integrated group communication subsystem is being implemented within MICROS to provide a basis for construction of a complete network operating system. In the coming year, the subsystem will be used to evaluate proposed group communication techniques. The subsystem will include support mechanisms for planned distributed applications such as the in/out medium-level distributed language system and the BugNet parallel debugging system.

The MICROS system must work well both on different types of computers and on networks that are connected in different ways — ways not known while the MICROS soft-
A Portable Modula-2 Operating System: SAM2S

ware is being written. Distributed control algorithms already designed for MICROS have included a Focus25 initializer that transparently forms a networkwide hierarchical control structure and a distributed Wave Scheduler26,27 that assigns idle nodes to task forces. The wave scheduling technique relies on a control hierarchy, includes mechanisms for avoiding static deadlocks, and can extend to any size network. Research in distributed task force scheduling schemes18 and netcomputer load-balancing mechanisms28 is planned. We also will evaluate other decentralized algorithms for management of globally shared system resources in large netcomputers with thousands of nodes.

Figure 3 shows one use of overlapping communication groups within a decentralized control hierarchy. Each triangular boundary encloses two groups. The working group consists of a number of sibling nodes plus their common parent. The recovery group adds the grandparent to the parent and siblings. The siblings execute user and management tasks as requested by the parent. To avoid overloading the parent during normal working conditions, the siblings pass to their parent only task results and summaries of management information about lower-level groups. If one of the nodes fails, the remaining members of the recovery group all communicate to redistribute the tasks of a failed sibling or to elect a replacement for a failed parent. Management information in a failed parent can be regenerated from the combined states of the siblings and grandparent. A failed grandparent is replaced as a parent by the next higher group in the hierarchy.

The MICROS network currently consists of three Heurikon 68000s and two DEC LSI-11/23s connected by a single Ethernet. An expanded network including two DEC LSI-11/23 nodes, at least seven Heurikon 68000 nodes, some Intel 80186 nodes, and several DEC MICROVAX nodes is planned for future research. Each node will have two Ethernet ports, probably connected to the nearest two busses in a horizontal/vertical grid of Ethernets. Ethernet links to several SUN workstations and VAX 750/780 computers running Berkeley 4.2 UNIX are planned. We have started developing a fully compatible UNIX file system written in Modula-2 that will allow MICROS users to share files and whole disks with UNIX system users. MICROS will unify local operating systems to present a networkwide UNIX environment to users.

CONCLUSIONS

We have found Modula-2 much better for writing system code than the combination of Concurrent Pascal19 and assembly code that we used for MICROS during 1978–81. The Modula-2 system, running on the same LSI-11 processor, is faster by a factor of 4 to 10 in several modalities. The 68000 version is even faster. Compiler and system code run faster because native machine code, not interpreted P-code, is produced. Flexible, selective synchronization operations defined by library modules allow faster execution of highly concurrent systems than do Concurrent Pascal monitors, which block processes too indiscriminately. System errors can be located much faster using the post-mortem symbolic debugging system that is part of the Modula-2 task library. System corrections are faster because only a few modules, not the entire system, must be recompiled for each set of corrections, since there is type-checked, separate compilation of Modula-2 modules. System development by a group is faster, because only the definition modules providing the interfaces between modules need to be approved before all programmers can start producing and compiling code.

The tiny run-time system, small compiler, and use of device interfaces written in high-level code all greatly simplify the porting of Modula-2 systems. We did not encounter major problems in porting SAM2S to 68000 systems. A few high-level modules have been changed slightly to make them truly machine-independent. Almost all the changes have involved the consistent use of long and short variants of integers and cardinals on the two systems. Communication between heterogeneous computers requires an external standard for the order of byte transmission. We have chosen the Xerox Ethernet standard of high-byte-first order, which is also the standard for the 68000 microprocessor. Bytes are reversed in order as they enter or leave any of the LSI-11 systems. Porting SAM2S to a new computer requires rewriting of about 7,000 lines for code generation and loader modules, 1,000 lines for low-level kernel modules, and 1,000 to 4,000 lines for new device drivers.

Use of Modula-2 has allowed us to port SAM2S from LSI-11 to 68000 systems in six months. It has allowed us to combine the efforts of dozens of student programmers into a working operating system. The flexibility and portability of Modula-2 systems will allow us to continue to explore ways to control networks of thousands of computers.

ACKNOWLEDGMENTS

Many members of the MICROS research group helped to develop SAM2S. Especially important contributions have
been made by Shridhar Acharya, Divyakant Agrawal, Bill Earl, Miguel Garcia, Arun Garg, Mike Palumbo, Yanick Pouffary, Soumitra Sengupta, Rick Spanbauer, Shidan Tavana, and Kok Sun Wong.

This research has been supported in part by National Aeronautics and Space Administration Grant NAG-1-249, Army Research Office Contract DAAG-29-82-K-0103, an external research grant from Digital Equipment Corporation, and National Science Foundation Equipment Grants MCS80-06925 and MCS82-03955.

REFERENCES