Chapter 32

The IBM System/38: A High-Level Machine[1]

S. H. Dahlby / G. G. Henry / D. N. Reynolds / P. T. Taylor

One of the primary characteristics of the IBM System/38 that identifies it as an advanced computer system is its high-level machine instruction interface, which incorporates new architectural structures and provides a much higher level of function than traditional machine architectures, such as the IBM System/3. The function and architectural structures are more similar to those of high-level languages than to conventional machines. The purpose of this article is to describe the advantages and salient architectural features provided by the System/38 instruction interface, and how they are realized in the specifics of the System/38 machine.

Relevant system objectives

Many factors influence the choice of the architectural characteristics [Henry, 1978] of a new system. In System/38 the primary influences, such as anticipated user requirements and hardware technology trends, led to the adoption of some major objectives for the total system. Briefly, these were:

- Programming independence from machine implementation and configuration details
- High levels of integrity and authorization capability with minimal overhead
- Efficient support in the machine for commonly used operations in control programming, compilers, and utilities
- Efficient support in the machine for key system functional objectives, such as data base and dynamic multiprogramming.

The following sections highlight the major System/38 instruction interface concepts and features that address these objectives.

Independence from Machine Implementation and Configuration

In previous systems, the ability for users to take advantage of new technology and implement new function was limited by dependence on a specific low-level instruction interface; for example, dependence upon the hardware-implemented address size. One of the major goals of System/38 architecture was to enable users to be as independent as possible of hardware and device characteristics.

In System/38, hardware dependencies have been absorbed by internal microcode functions that provide an instruction interface, which is largely independent of hardware details. Users of the instruction interface, therefore, need not be concerned with hardware addressing [Berstis, Trutal, and Ranweiler, 1978], auxiliary storage allocation and addressing [French, Collins, and Loen, 1978], internal data structures and relationships [Pinnow, Ranweiler, and Miller, 1978], channel and I/O interface details, and internal microprogramming details [Hoffman and Soltis, 1978].

This hardware independence characteristic of the System/38 instruction interface is due in large measure to the use of an object-oriented interface [Pinnow, Ranweiler, and Miller, 1978] instead of the more conventional byte-oriented interface. An *object* is a System/38 instruction interface construct that contains a specific type of information and can be used only in a specific manner. A number of different types of objects are defined in the interface, and various object-specific instructions are provided to operate upon each object type. An example of a System/38 instruction interface object is a data space (file), which has associated instructions for operations such as the adding and deleting of records [Watson and Aberle, 1978].

Each object is created by a System/38 interface instruction that uses a user-specified data structure to define the object's characteristics and initial values. Once the object is created, its internal stored format is not apparent to the user (with the one exception discussed below). The status and values of the object may be retrieved or changed by using interface instructions, but the internal format of the object cannot be directly viewed or modified. That is, objects can be operated upon functionally, but not as a byte string. This approach prevents dependence on the internal format of the object and enables applications to remain independent of evolving internal implementations of the machine.

There is one specific exception to this shielding of the internal format of an object. A *space object* is a construct that can be used by a program for storage of and operation upon byte-oriented operands such as character strings and numeric values.

In addition to this object orientation, main storage and auxiliary storage addresses are not directly apparent in the System/38 instruction interface [Berstis, Truxal, and Ranweiler, 1978; Pinnow, Ranweiler, and Miller, 1978]. All interface addressing of objects is accomplished by resolving symbolic names (supplied by the user) to a pointer. A *pointer* is an object that is used only for addressing and does not permit examination or manipulation of the implied physical address. A *system pointer* gives a user the

[1]*IBM System/38: Technical Developments*, pp. 47–50. © 1978 by International Business Machines Corporation. Reprinted by permission.

ability to address objects; for example, to create or destroy an object or to examine or directly modify its content through associated specific instructions. A *space pointer* allows the direct addressability of bytes within a space object. Both of these pointer types can be contained within a space object, but they can be used for addressing only when operated on by pointer manipulation instructions. Pointers are assured of validity via tagged storage in both main and secondary storage. Direct modification of a pointer area via a "computational" instruction results in the tag becoming invalid and causes the pointer to no longer be usable for addressing purposes.

Similarly, users are not concerned with the addressing structures of either main storage or auxiliary storage [French, Collins, and Loen, 1978], or even necessarily that there are multiple levels of storage, since all storage used for all objects in the system is allocated and managed by the machine. That is, there is no differentiation in the System/38 instruction interface as to where an object or portions of an object reside. The total address space of System/38 thus consists of an unconstrained number of objects, uniformly addressable by pointers.

Similar constructs shield the System/38 instruction interface user from dependencies upon channel and I/O device addresses and low-level communication protocols.

Figure 1 illustrates this basic object-oriented, high-level interface approach.

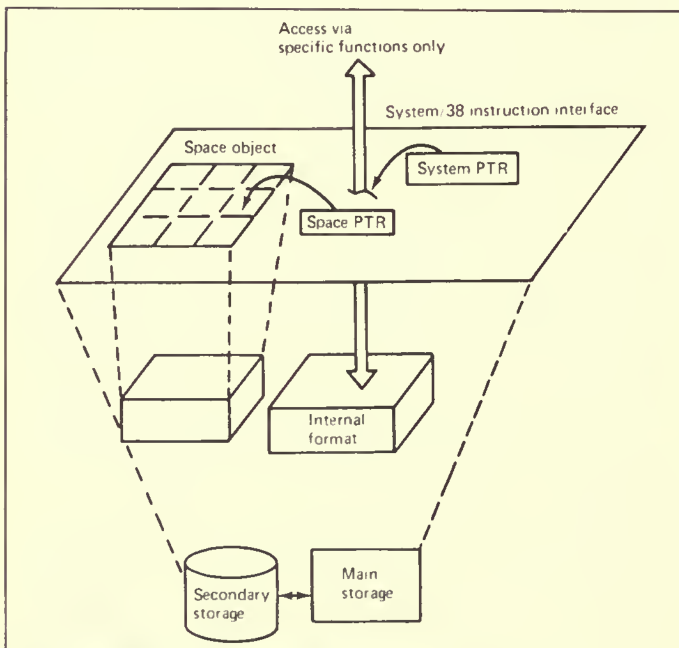

Fig. 1. System/38 object-oriented structure.

Integrity and Authorization

A natural consequence of the object-oriented approach is improved system integrity and authorization mechanisms [Berstis, Truxal, and Ranweiler, 1978]. All user information is stored in System/38 instruction interface objects. Access to that information is through System/38 instructions that ensure the structural integrity of the manipulated objects. An attempt to misuse an object is thus detected and causes the instruction execution to be terminated and an exception condition to be raised. An example is the attempt to directly change a byte within a program object.

Authorization capabilities are likewise facilitated by the System/38 instruction-interface object-oriented structure. Each user of the machine is identified by a user profile, which is itself an object. Each object in the system is owned by a user profile, and the owner may delegate to other user profiles various types of authority to operate on the objects. Processes (tasks) execute under a specific user profile (in the name of a user), and functions executed within a process verify that the objects referenced have been properly authorized to that user.

Figure 2 illustrates this approach to providing integrity and authorization capability.

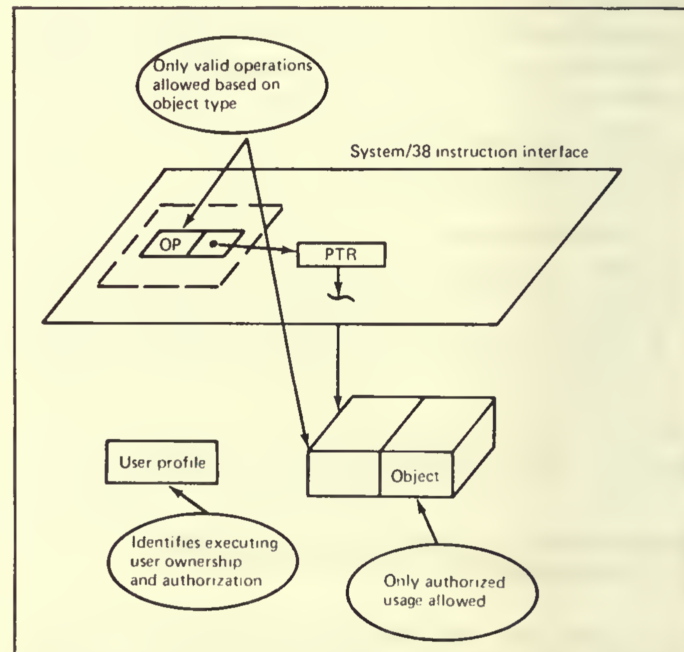

Fig. 2. System/38 instruction interface integrity and authorization scheme.

Support for Common Programming Functions

The System/38 instruction interface is designed to provide direct support for a wide variety of functions common to control programming, compilers, and utilities. This increased level of machine function eliminates the need to implement these common functions in multiple programming components, increases consistency across all programming components, and supports programming approaches conducive to providing integrity and reliability.

There are two basic modes of addressing in the System/38 interface. The first is *pointers*, which allow varying addressability to all objects and bytes within space objects. The second, *dictionary addressing*, deals with program references to values within a space object.

Operands referenced in program instructions are defined in a dictionary portion of the program separate from the instructions themselves. Instruction operands are index references to these dictionary entries which define operand characteristics such as data type and length. Binary, zoned decimal, packed decimal, character, and pointer data types are examples of operand characteristics that may be defined. The dictionary entries do not contain the operand values; the specific location of the operands is not apparent to or required by programs. However, the user can control the general type of location characteristics: for example, relative to the area addressed by a pointer or relative to the storage area allocated for program variables within the executing process.

This approach of having instructions reference dictionary entries describing the operand characteristics allows additional capability over low-level instruction interfaces. For example, the following high-level capabilities are provided:

- Computational instructions are generic with respect to data type and length. For example, there is only one numeric add instruction in the System/38 instruction interface: it operates on whatever data is defined in the operand definition dictionary. This enables the use of source and receiver operands of varying type, length, and decimal positioning with all conversions and scaling being performed by the machine.
- Arrays may be defined in the interface and instruction operands support array indexing to locate specific elements of the array.
- Since applications often allow operations on multiple formats of data, some instructions (for example, the copy instructions) support late-binding of data definition where the data (type, length, and decimal positioning) need not be defined until the instruction is executed.

In addition to these types of high-level data operations, the System/38 instruction interface provides and, in some cases, requires functions intended to support programming constructs more directly than in traditional machines. For example, programs are invoked through call/return functions defined in the interface. Argument/parameter functions provide communications from one program to another. Allocation and initialization of storage for program variables within a process is performed by the System/38 machine. Additional examples are found in [Watson and Aberle, 1978] and [Howard and Borgendale, 1978].

Figure 3 illustrates this System/38 program structure and the general relationship between a high-level language program and the corresponding System/38 constructs.

Support for Key System Functions

The System/38 machine was designed to support a usage environment characterized by a dynamically changing application load consisting of a wide variety of application types—all utilizing advanced functions such as data base. For example, batch, interactive, and transaction processing, along with program development activities, may all be executing concurrently with dynamically changing workloads and priorities. One of the key requirements for the System/38 instruction interface was to provide efficient support in this type of environment for application requirements such as multiprogramming and data base operations. This centralization of function in the machine simplifies the user programming task and reduces overhead in a dynamic multi-user environment.

Two examples of this system function support will be described here—multiprogramming and data base. Similar high levels of

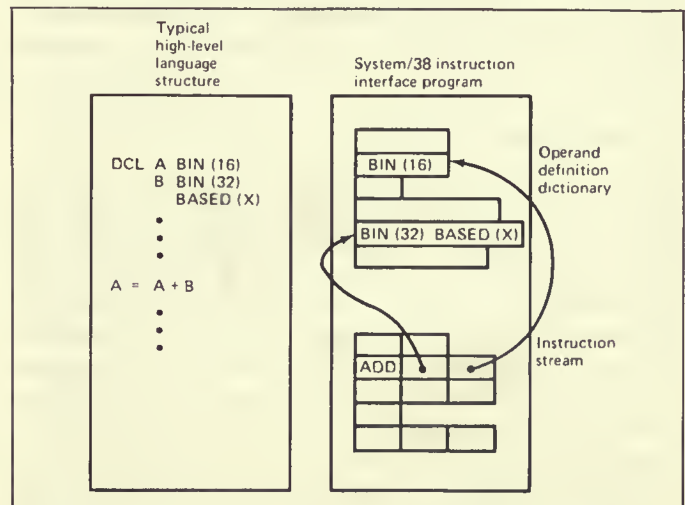

Fig. 3. System/38 instruction interface program structure.

machine capability exist in other major functional areas such as I/O.

System/38 supports multiprogramming through the concept of processes. A "process" is similar to a task in other systems and is the basis for managing work in the machine. The user of the System/38 instruction interface controls the number of processes currently initiated, the priority of each process, and the relationship of one process to another, that is, with respect to processor utilization and storage utilization. The machine then allocates the processor and storage resources based on these parameters as well as on the current status of the process, for example, waiting or dispatchable.

This level of multiprogramming support in the System/38 machine offers advantages like these:

- A single resource management mechanism is applied to processing across all system activities. This reduces overhead and results in better management of resources in a complex and dynamic environment.
- Other efficient resource management mechanisms can be used to take advantage of hardware characteristics without programming dependencies.

Similarly, the System/38 machine provides the basic functional building blocks for a high-function integrated data base. Data base objects include a comprehensive set of functions supporting different access mechanisms, file sharing, record format definition and mapping, efficient record retrieval, update, add, and delete. This allows, for example, a data base file structure to be defined that maps a single physical file into records with multiple formats and content. In addition, a single physical data base file may have multiple indexes (access paths) defined over it, all of which are concurrently updated when the file is changed. Each user of the file may view the data in the form suitable to a particular application.

Overhead Considerations

One of the major problems inherent in the implementation of a high-level instruction interface such as that provided for the System/38 is overhead. In order to reduce the potential overhead, and also to facilitate future extensions, the System/38 instruction interface definition does not require a directly executable implementation of the instruction interface. The instructions and the operand definition dictionary are presented to the instruction interface and are translated into an executable microcode structure called a program object. The internal microcode format is not apparent at the interface. Figure 4, System/38 executable program creation, illustrates this process.

Having an executable program creation step allows the system to have the advantage of both a high-level instruction interface and reduced overhead at execution time.

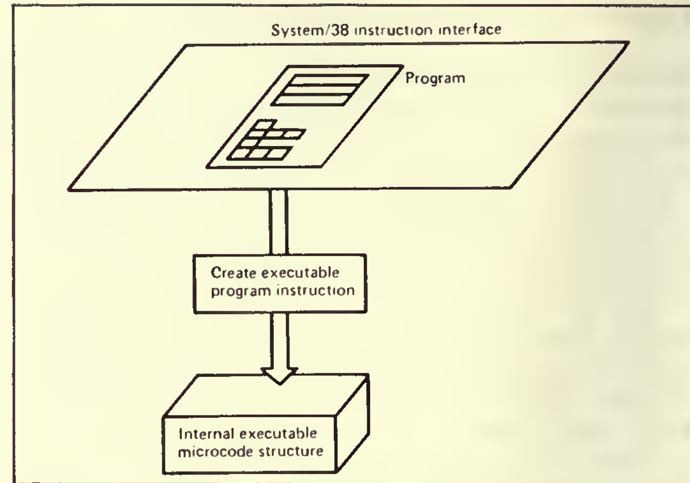

Fig. 4. System/38 executable program creation.

In addition, direct support of high-use functions in the System/38 instruction interface, as previously described, is itself an approach toward reducing *system* overhead. A single implementation of a complex function that can be applied system-wide reduces overhead.

Also, by implementing these functions in the machine, hardware facilities can reduce the overhead that is associated with the higher level implementation typically required in programming.

Summary

The IBM System/38 provides a new type of machine instruction interface that comprises a high level of function together with structures similar to high level language structures and includes computation, addressing, and such traditional programming functions as process (task) management, resource management (storage and processor), data base management, and device handling. This new machine was designed to satisfy major design objectives for the entire system—hardware, microprogramming, and program products. The concept of a high-level machine has been discused in the literature and has been experimented with in both industrial and research environments; however, System/38 is the first IBM system to bring the advantages of a high-level machine to the business user.

References

Berstis, Truxal, and Ranweiler [1978]; French, Collins, and Loen [1978]; Henry [1978]; Hoffman and Soltis [1978]; Howard and Borgendale [1978]; Pinnow, Ranweiler, and Miller [1978]; Watson and Aberle [1978].

The IBM System/38: Object-Oriented Architecture[1]

K. W. Pinnow / J. G. Ranweiler / J. F. Miller

System/38 provides a range of capability not previously available in low-cost data processing systems. This capability is made possible by the use of a number of technical innovations. One of these is the *object*. This article discusses objects—the means through which information is stored and processed on System/38. Included are the concepts, purpose, and characteristics of System/38 machine objects and their use by the Control Program Facility (CPF).

Object Concepts

Previous machine instruction sets have provided bit- and byte-string manipulation capabilities. The machine instruction set in System/38 provides similar functions and also provides machine instructions that operate on complex data structures to accomplish high-level functions.

Some of the data structures are similar to such things as programs and data files in conventional systems. Some are unique to System/38. The data structures that appear in the instruction interface are collectively categorized as objects.

An object is brought into existence through execution of a create instruction. The user controls the creation of the object through a template [Allsen, 1978] that provides a set of attributes and values that are to apply to the new object. The new object also has operational characteristics that define the set of functions that may be accomplished through it. Examples of object attributes and operations are shown in Fig. 1.

The three examples of attributes illustrated in Fig. 1 are (1) a *name* that permits symbolic reference to the object, (2) an *existence* that specifies whether implicit destruction is *allowed*, and (3) *ownership* that identifies who, if anyone, owns the object.

The set of instructions that are operationally meaningful to an object consist of *generic* operations that apply to all types of objects and *unique* operations that apply to a specific type object. The generic operations are primarily *authorization-*, *addressing-*, and *resource*-related [Berstis, Truxal, and Ranweiler, 1978]. The unique operations include a *destroy* that removes the object from the system, some form of *materialize* that identifies the object's attributes or content, and sometimes a *modify* that changes the attributes of the object. Many other unique operations exist that are not identified in Fig. 1.

[1]*IBM System/38: Technical Developments*, pp. 51–54. © 1978 by International Business Machines Corporation. Reprinted by permission.

Object	Attributes	Name	
		Ownership	
		Existence	
	Generic operations	Explicit functions	Authorization
			Addressing
			Resource
		Implicit functions	Atomicity
			Lock enforcement
			Authorization enforcement
	Unique operations	Explicit functions	Materialization
			Modification
			Destruction
		Implicit functions	Atomicity
			Lock enforcement
			Authorization enforcement

Fig. 1. Some examples of object attributes and operations in System/38.

Each operation, whether generic or unique, also provides significant implicit functions. The implicit functions are *authorization*, *lock enforcement*, and *atomic* (exclusive) *operation*.

Object Purpose

The concept of an object gives a common attribute to a group of data structures and enables the definition of an interface that produces a number of benefits.

The existence of objects allows systematic manipulation of structures. Their presence permits the definition of an instruction interface that is consistent across a wide range of supervisor and computational instructions.

Objects exist to make users independent of specific implementation techniques used in the machine. Since it is necessary that users control the data used in supervisor functions, object management capability is provided. When a request for a high-level machine function is made, a specific instruction operator (operation code), optionally an attribute template, and an object are specified. System/38 uses the object to accumulate results of operations, to store them in such a way that they are safe from inadvertent modification, and to assure that they are available for subsequent operations.

Objects exist to make the user independent of the addressing structure actually used in the hardware. Although main storage and auxiliary storage exist in System/38, users are shielded from the mechanics of actually addressing that storage. In other words, objects remove the traditional responsibility of mapping data onto physical storage.

Object Characteristics

For an object like a program, creation establishes the essential content of the object, and subsequent instructions use it operationally. For other objects, the creation is primarily a space allocation mechanism for which succeeding operations establish the content. For example, once a data space has been created, records may be inserted into it. Management of the size of an object and changes to that size are generally transparent to the System/38 user.

All System/38 machine objects are *encapsulated*. Encapsulation is the process of accepting a definition of an object through a create instruction and using this definition to produce an object whose internal structure is only accessible to the machine. Objects are encapsulated to maintain the integrity of the internal structure and to permit different implementations of the machine instruction interface without impact to its users.

It is possible to associate an unencapsulated (byte string) area with each object. This byte-string area is referred to as a *space* and is up to 16 megabytes of virtual storage into which the machine user can build control blocks of other control information or data. As a degenerate case of an object, one with essentially no encapsulated portion, a space exists as an independent object. Whether it is an object itself or is associated with another object, a space has its size modified through explicit instructions by the machine user.

System/38 Machine Objects

The following lists and briefly describes the objects of the System/38 machine-instruction set.

Access group. An object that permits the physical grouping of other objects to achieve more efficient movement of the objects between main storage and auxiliary storage.

Context. An object that contains the *type*, *subtype*, and *name* of other objects to allow addressability.

Controller description. An object that represents an I/O controller for a cluster of I/O devices or a station that attaches groups of communication devices over the same data communication link.

Cursor. An object used to provide addressability into a data space.

Data space. An object used to store data base records of one format.

Data space index. An object used to provide a logical ordering of records stored in a data space.

Index. An object used to store and automatically order data.

Logical unit description. An object that represents a physical I/O device.

Network description. An object that represents a network port of the system.

Process control space. An object used to contain process execution.

Program. An object for uniquely selecting and ordering machine interface instructions.

Queue. An object used to communicate between processes, and between a process and a device.

Space. An object used for storing pointers and scalars.

User profile. An object used to identify a valid user of the machine interface.

CPF Use of Machine Objects

The CPF extends the object-oriented approach of the machine and provides its users with a high-level, object-oriented interface [Harvey and Conway, 1978]. All data stored on the system by CPF users is stored in object form and is processable in terms of control language commands and high-level languages. To the user of CPF, objects are named collections of data, and the functions associated with objects provide the vehicle for processing this data and obtaining work from the system. The 19 objects presented to the user at the CPF interface include conventional constructs, such as files and programs, as well as constructs that are unique to System/38, such as job descriptions and message queues [Demers, 1978].

The functions that CPF provides for its objects include some that are object-type specific and some that are generic with respect to object type. The object-type specific functions define and limit the way in which an object can be used while the generic functions provide for authorization, locking, saving, restoring, dumping, moving, and renaming objects. Through the generic functions, the user has a way of managing objects once they exist.

Objects are brought into existence through the specification of a create command that defines the name, attributes, and initial value of the object to be created. Each object is assigned a type and subtype as a part of the creation process. The object's type is determined by the kind of machine object created to support the object that the CPF user wishes to create; the object's subtype designates the use that CPF intends for the machine object. Each unique use that the CPF makes of a machine object is assigned a unique subtype identifier. This aspect of the design is important because it is through the use of unique types and subtypes that the

system can ensure that each type of object is always used in the way it was intended. After an object has been created, it remains on the system until it is explicitly deleted via a delete command. At the time an object is created, CPF places the name of the object into a machine object known as a *context*.

Contexts are presented to the user as libraries. Because the functions associated with contexts are capable of finding an object based on its name, type, and subtype, libraries can be considered as a catalog or container for the user-created objects. Whenever an object is to be found, CPF initiates a search for the object either in a single library or through an ordered list of libraries that the CPF maintains with each executing job. When the list of libraries is used to find an object, each successive library in the list is searched until the object is found. Using the list of libraries to find the objects to be processed is advantageous because the same commands or program can perform functions on different objects merely by changing the order of the libraries in the library list.

CPF maintains descriptive information for all objects and provides functions for the retrieval and display of this data. The descriptive information records who the object owner is, when the object was created, where the object has most recently been saved, and text information provided by the user to further describe the object.

An important feature of CPF object architecture is the manner in which CPF objects are constructed. CPF uses machine objects as building blocks to produce the objects that CPF users see. Figure 2 shows an example of how one kind of Control Program Facility object is constructed.

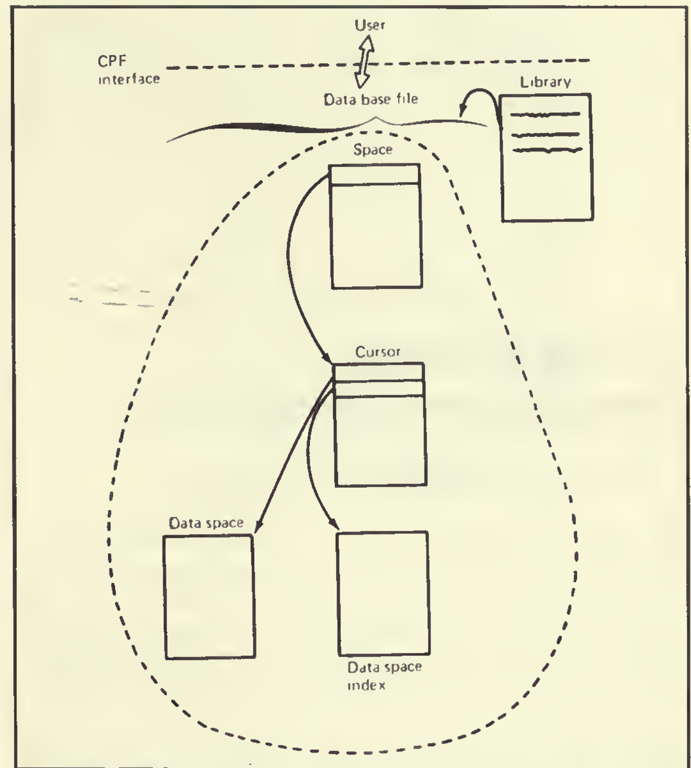

Fig. 2. An example of how one kind of Control Program Facility object is constructed.

In this example, four types of machine objects (a data space, a data space index, a cursor, and a space) are combined to produce the higher level CPF object known to the user as a *data base file*. CPF manages the individual pieces of a file in a way that allows the user to perceive the file as a single entity. For example, the separate pieces of the file come into existence when a single create-file command is processed and remain in existence until the file is explicitly deleted. Thus, the user is relieved of the complexity and organizational details of the data and can process it as a logical entity. When lower level objects are put together to form a higher level object, the higher level object is known as a composite object. CPF object architecture permits any type of System/38 machine or CPF object to be combined to produce a new type of object. In fact, CPF-provided functions for managing objects are table-driven, based on unique object type and subtype combinations. This aspect of the design means that the object-oriented approach can be quickly and easily extended. It also permits new kinds of objects to be compatibly introduced later on in the life of the system.

The key advantages of the System/38 building block architecture, however, is that the implicit functions provided by the machine for its objects are made directly available to the end user in a consistent manner. For example, implicit in all CPF objects are the machine-provided functions of security, lock enforcement, and object resolution by name. The benefits of this architecture are readily apparent when one contrasts the approach of System/38 with that of other systems having different addressing structures for different collections of data, added-on security functions, and user interfaces that require knowledge of the physical aspect of data organization.

Summary

The object orientation of the System/38 machine and CPF interfaces permits common provision of function at each interface. With machine-interface objects, the hardware addressing mechanism and the internal format and organization of data are transparent to the user; serialization and authorization functions

are implicit in the objects. The key characteristic that makes this possible is encapsulation of objects in the machine-instruction interface. Since CPF uses the objects of the System/38 instruction interface as building blocks, its objects possess all the function of the machine objects.

References

Allsen [1978]; Berstis, Truxal, and Ranweiler [1978]; Demers [1978]; Harvey and Conway [1978].

The IBM System/38: Addressing and Authorization[1]

V. Berstis / C. D. Truxal / J. G. Ranweiler

The high-level machine interface of System/38 achieves user independence from the internal machine implementation primarily through the use of an object-oriented architecture. Objects representing storage for constructs such as programs, processes, and data base files are accessed through a consistent, integrated addressing structure. Because authority enforcement and control of shared objects are critical in multiprogramming environments, these functions have been incorporated into the addressing path. This article describes some of the key features of the addressing design of System/38 and how they are presented to the user through the Control Program Facility (CPF), which is described by Harvey and Conway [1978].

Objects and Spaces

Before addressing can be described, it is necessary to define what is accessed. Everything stored in the system is an *object* (see Fig. 1), which consists of a functional portion and an associated space (see Pinnow et al. [1978]). The functional part of an object is used to implement a particular construct. For example, the functional part of a program object is created by the translation of System/38 machine instructions into microcode. The program is said to be *encapsulated* because there is no direct access to the storage used to support it. Instead, the object is manipulated at a high level through the System/38 instruction set. In this way, encapsulation ensures the functional integrity of all objects.

The *associated space* portion of an object is a region of bytes that can be directly manipulated by the machine user. The space is associated with the functional part of the object and provides a convenient way of storing additional (user-defined) data pertinent to that object's usage. One type of object, called a *space object*, has no functional part. Its associated space is used to provide storage for control blocks, buffers, pointers, and other data.

Pointers

There are four different types of pointers. *System pointers* address objects; *space pointers* and *data pointers* address specific byte locations within the space portion of an object; and *instruction pointers* control execution flow. This article covers object addressing through system pointers.

A *system pointer*, used to address an object, contains both the location of the object in storage and object usage rights, as will be discussed later. Only specific System/38 instructions can create pointers. Although pointers can be copied, the user cannot construct pointers by bit manipulation. As a result of these properties, System/38 has the basic elements of *capability based addressing* [Linden, 1976].

Name Resolution

A system pointer exists in one of two states: resolved or unresolved. In the *unresolved state*, the pointer specifies the name of an object and not its location. When the pointer is first referenced (see Fig. 2), the machine searches for an object having the specified name. Once found, the resulting object location is stored in the pointer, thereby eliminating subsequent searches. The pointer is then said to be in the *resolved state*.

The search performed during pointer resolution involves the use of objects called *contexts*, containing object names and locations. Various execution environments are obtained by specifying an ordered list of contexts to be searched. For example, the production and test versions of files can be located through different contexts. Therefore, by simply exchanging the contexts searched, either programming environment can be achieved.

[1]*IBM System/38: Technical Developments*, pp. 55–58. © 1978 by International Business Machines Corporation. Reprinted by permission.

Fig. 1. System/38 objects and pointers.

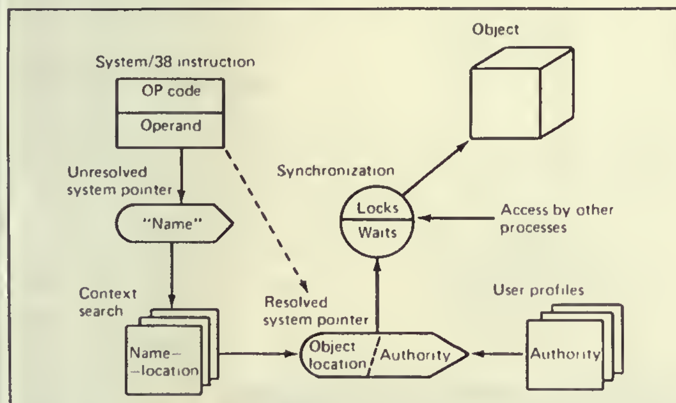

Fig. 2. The object access path.

Authorization

The ability to control pointer resolution in the machine is not sufficient to effectively control the users' access to objects because it is an "all or nothing" type of control. The System/38 object authorization mechanism provides the fineness (granularity) of control needed for the wide range of operations performed on objects.

Every reference to an object requires that the user have the appropriate authority for the operation to be performed; otherwise, the operation is suppressed and the attempted violation is recorded. The authority checking function is uniformly applied to all types of objects. Separate authorities (retrieve or update, for example) can be granted to individual users or to all users (the public). Therefore, a user's authority can be limited to what is exactly necessary for an application. For example, a user might be authorized to retrieve data from a data base file but not to update or destroy the file.

Sources of Authority

A prerequisite for authority verification is the identification of the user. This prerequisite is satisfied through the use of an object called the *user profile*, which identifies the user and the user's authority. Every process is initiated with a specified user profile as the primary source of authority. Object authorities can be granted to or revoked from a user profile, thus providing control over the authority available to the process. Objects can also be publicly authorized, thereby eliminating the need to explicitly authorize every user profile.

In some applications, subprograms require a different amount

of authority than that available to the calling program. To accomplish this, programs can *adopt* a user profile (Fig. 3). The adopted user profile adds its authority to what is already present in a process. When the program calls other programs, the adopted user profile authorities can be optionally propagated to the called program. This provides considerable flexibility in controlling the security environment.

Once authority to an object has been established, it can be optionally stored in the pointer to that object. This provides faster authority verification than with unauthorized pointers.

Other Authorizations

One type of authority not related to objects is the *privileged instruction authority*. Such authorization is used for process initiation, user profile creation, machine reconfiguration, etc. Other *special authorities* range over many machine functions rather than specific instructions. For example, *all object* special authority permits unlimited use of all objects in the system. The control of storage resources is another wide-range authority. The storage occupied by objects is charged against the *storage limit* of

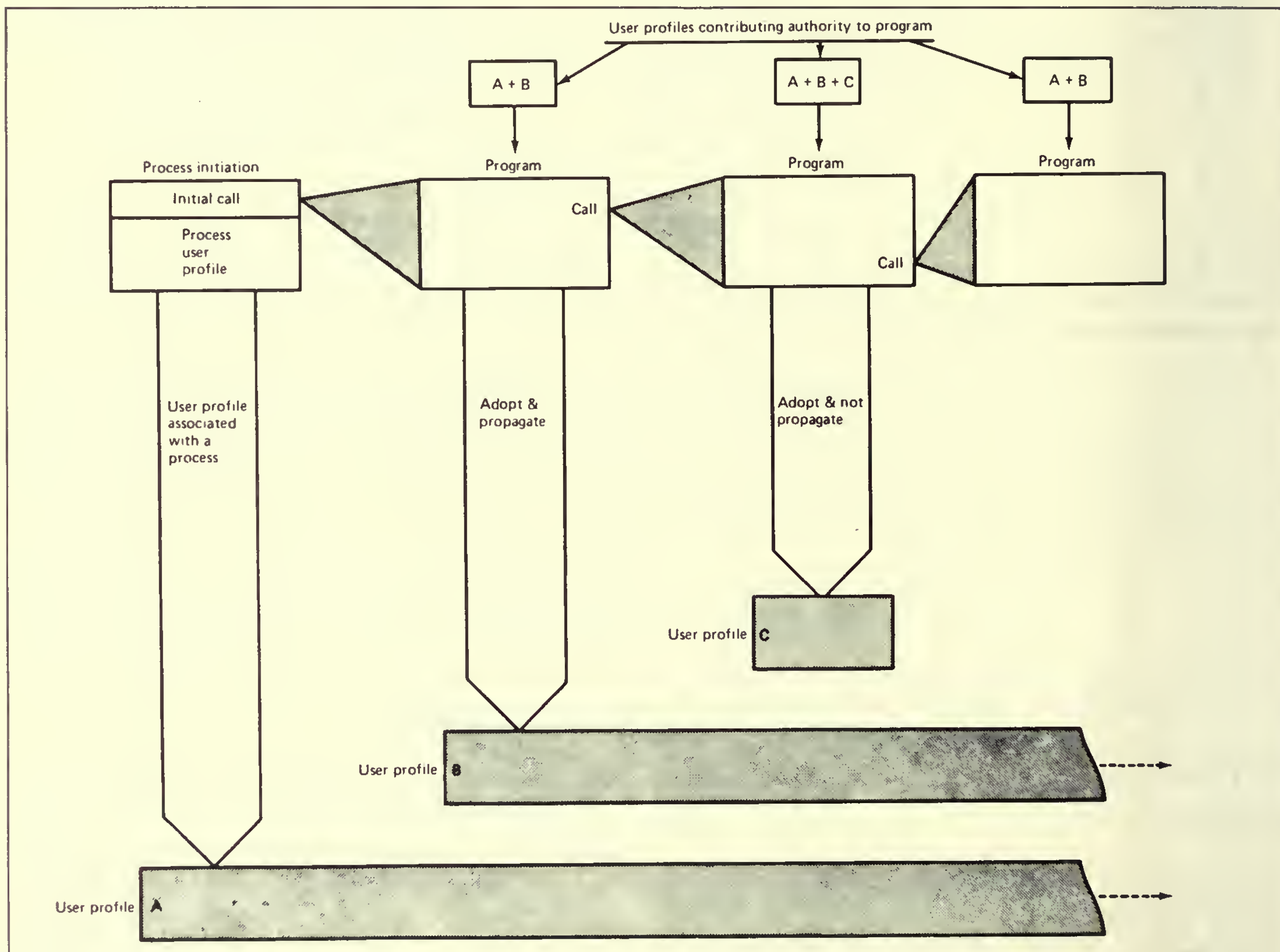

Fig. 3. User profiles as sources of authority.

the user profile (the owner) under which they were treated Owners have implied object authority to the objects they own.

Locking and Synchronization

The authority mechanism of System/38 ensures that an application accesses only objects within its intended rights. When multiple applications attempt to reference the same objects concurrently, additional controls are provided to prevent interference. System/38 incorporates implicit synchronization functions into the object access implementation to accomplish this. For example, if one process is updating an object while another process is attempting to access the same object, the operations are automatically serialized. On the other hand, if both processes are retrieving data from the same object, the operations are allowed to proceed simultaneously. Therefore, contention is reduced and integrity of the object is ensured.

Explicit synchronization is available to the users in the form of *locks*. By locking an object, the user can control the access of other users to the object. Entire sequences of operations can be serialized when required to maintain data integrity. In addition, record level locks in data base files reduce much of the contention that would be present if the entire file were locked.

Synchronization functions complete the machine addressing path, which starts with the object name and continues through pointer resolution and authority verification.

Addressing Path Usage

The Control Program Facility (CPF) is an IBM program product providing the user a high-function, ease-of-use interface to the machine [Harvey and Conway, 1978]. With the high-level machine facilities available in the System/38, the CPF addressing and authorization function uses both capability-based and symbolic object addressing with authority validation at execution time.

CPF uses machine pointer resolution, authorization management, and locking to implement internal CPF security and synchronization. It provides these facilities to the user through CPF interfaces.

Within CPF, the work management component isolates and protects its critical resource control and scheduling functions by executing them under the system user profile. The remaining CPF modules execute under the user's profile. Thus, the machine authorization management directly validates the user's authority to perform every requested function on any specified object. Everything in CPF is an object. I/O devices and Control Language commands are objects, as are more typically files, programs, and libraries. Because of this, an installation can control system resources to the extent desired.

Installation Authorization

This control of an installation's resources has led to the concept of one specific user as an installation's *security administrator*. This user is entrusted with authorities allowing system-wide control of all users and their resources. A set of IBM-supplied user profiles is delivered with CPF, including one for the security officer. This profile has all-object authority, as well as authority to create and modify user profiles. Therefore, the security officer can enroll users on the system and control their use of system resources. When a user profile is created or modified, special authorities, resource allocation parameters, and a user password can be specified. The user password is for verification of user identity at sign-on and for determining the user profile associated with a process.

Once the user is executing, functions are performed by executing programs or commands. These functions reference objects (such as files) by name, and CPF locates the object through the use of the machine-addressing facilities. This is easily implemented because contexts (objects that contain names of other objects [Pinnow, Ranweiler, and Miller, 1978]) are used by CPF as system and user libraries. When an object such as a program or file is created, it is placed in a library. Subsequent referencing of the object initiates pointer resolution, and the machine not only locates the object, but validates the current user's authority to the object and determines whether serialization of an operation is necessary. To expedite authority checking, CPF requests that the authority be set in the pointer for future use.

CPF Object Authorities

When a user creates an object, it can be declared "public" or "private." Subsequently, any of the object's authorities can be granted or revoked to individual users or the public. Display commands are also available to report object authority.

Summary

The System/38 is based on an object-oriented architecture in which everything in the system is an object. An object can be referenced by its name, which is used in a pointer resolution process that includes authorization and synchronization functions. The resulting resolved pointer can contain object location and authority to avoid subsequent searches. The machine enforces

authority requirements on every object referenced, verifying the authority from the pointer or user profile(s). The user profile is an object that identifies a user in the system and contains all of that user's authorities. The CPF uses the machine addressing, authorization, and synchronization facilities, and provides their function to the user.

The System/38 thus delivers the flexibility of named object addressing and the integrity of machine-enforced authorization and synchronization of those objects.

References

Harvey and Conway [1978]; Linden [1978]; Pinnow, Ranweiler, and Miller [1978].

The IBM System/38: Hardware Organization of the System/38[1]

R. L. Hoffman / F. G. Soltis

The IBM System/38 hardware is designed to efficiently support its high-level machine architecture. An engineering design objective was to take advantage of new technologies such that certain high-level functions would be implemented in hardware and microcode. As a result, functions such as task dispatching, queue handling, virtual storage translation, stack manipulation, and object sharing became a basic part of the hardware control structure. A further objective was to provide for sufficient extendability to permit future implementation trade-offs.

Figure 1 shows the hardware configuration of the System/38. This article describes the hardware organization and the functions used by the hardware control structure.

Hardware Organization

System/38 hardware consists of a processor communicating over a high-speed channel to independently functioning I/O units. The processor and the I/O units have access to a main storage array. The System/38 processor, which is implemented in a new, high-performance large-scale integration (LSI) technology [Curtis, 1978], fetches 32-bit micro instructions from the random access memory (RAM) control store shown in Fig. 1 (8K words for both the 5381 Model 3 and Model 5). One micro instruction is executed for each processor cycle. The processor cycle times are 400 to 500 ns for the 5381 Model 3 (200 or 300 ns for the 5381 Model 5), depending on the micro instruction operation. In a single cycle, either one- or two-byte arithmetic operations may be performed on signed binary, unsigned binary, or packed format decimal data.

A new, high-density metal oxide semiconductor field effect transistor (MOSFET) technology main storage [Donofrio, Flur, and Schnadt, 1978] is available at two performance levels: 1100 ns fetch cycle time for the 5381 Model 3 and 600 ns fetch cycle for the 5381 Model 5. Data path width is four bytes to either memory. Available memory capacities are 512K, 768K, 1024K, 1280K, and 1536K bytes for either the Model 3 or 5. In addition, the Model 5 may have memory capacities of 1792K and 2048K bytes. Error correction circuitry (ECC) is used in both models.

Direct memory access for I/O units as well as for the processor is provided by the virtual address translation (VAT) hardware which converts 6-byte segmented virtual addresses to main storage addresses. Address translation tables in main storage and a translation lookaside buffer in hardware provide mapping from virtual to real main storage addresses, as discussed by Houdek and Mitchell [1978]. Virtual addresses are used in I/O operations, and page faults are allowed during data transmissions with low-speed devices.

Page faults are resolved by data transfer from secondary storage. Data is moved to main storage in 512-byte page units from disk storage via the channel.

Each I/O device is connected to a controller which is connected to the channel. Magnetic media controllers (MMC) [Froemke, Heise, and Pertzborn, 1978] are used for high data-rate devices such as disks, while microprogrammed I/O controllers (IOC) [Dumstorff, 1978] handle a multiplicity of lower data-rate devices.

Each system also includes a system control adapter (SCA) which shares an IOC with the keyboard display console. The SCA performs the system maintenance functions, including testing the hardware logic circuitry as described by Berglund [1978].

[1]*IBM System/38: Technical Developments*, pp. 19–21. © 1978 by International Business Machines Corporation. Reprinted by permission.

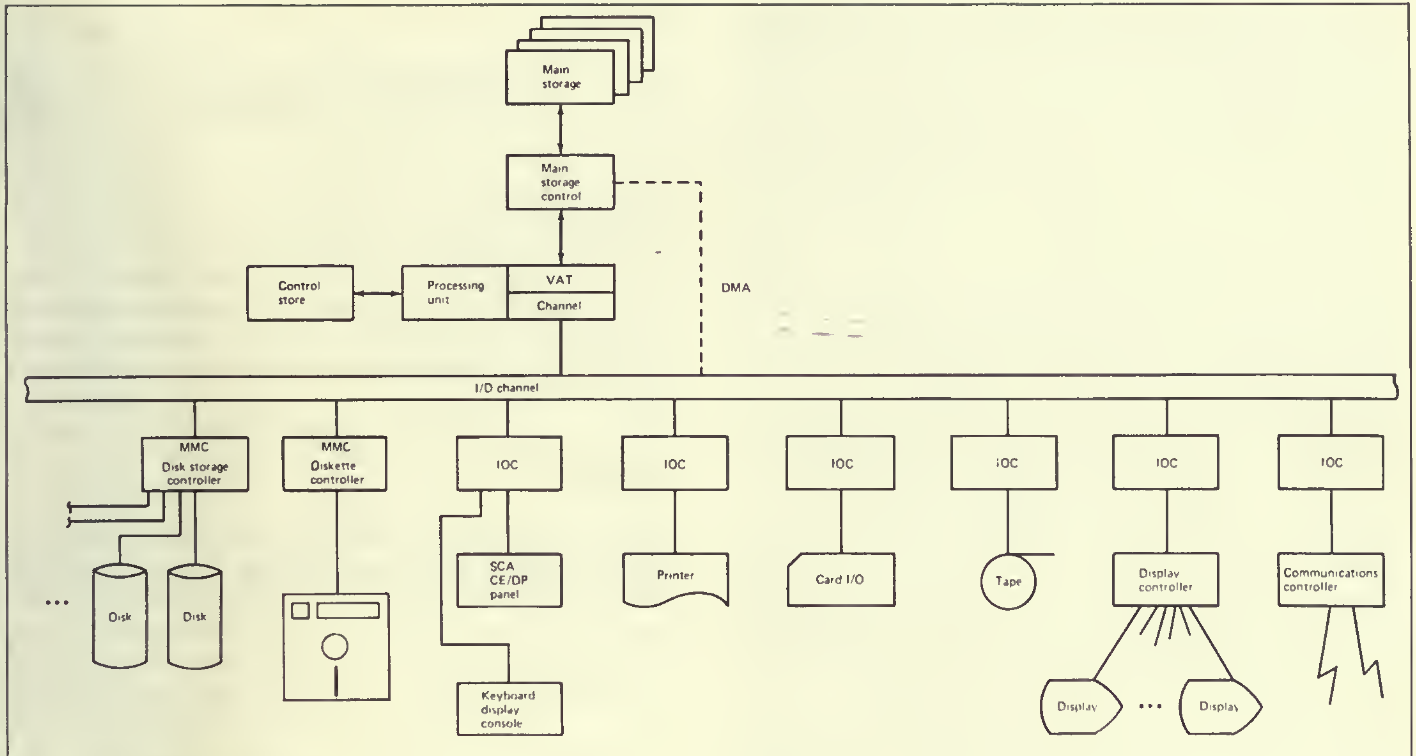

Fig. 1. Hardware configuration.

Control Structure

System/38 manipulates a unit of execution called the "task." All computer systems need to control execution and, in multiprogrammed systems like System/38, switch between units of execution, i.e., tasks. Traditionally, an interrupt structure with a fixed number of interrupt levels or classes, built on the hardware, is transformed by a software supervisor into a multilevel, interrupt-driven system to bridge the gap between the actual hardware and the abstract concepts of multiprogramming. The System/38 replaces this interrupt structure with a single tasking mechanism which is used to control all processing.

A multilevel, queue-driven task control structure is implemented in microcode and hardware on the System/38. A task dispatcher implemented in microcode allocates processor resources to prioritized tasks. I/O and program processing tasks are integrated in a common dispatching structure, with their priorities adjusted for system balance. I/O processing takes place when system resources are available, not when an I/O interrupt occurs.

I/O and program processing requests are stacked in main storage on a linked list called the task dispatching queue (TDQ). The task dispatcher selects the highest priority request from the TDQ and gives it control of the processor. Instructions associated with this task, known as the active-task, are executed until control is passed to another task.

A set of system control operations (SEND and RECEIVE) are used to communicate between tasks and to pass control between tasks via the task dispatcher. If the active task is to communicate with another task, it does so by sending a message to a queue in main storage known to both tasks. If the active task is to obtain a message from a queue, it executes a RECEIVE operation. If the message is available on the queue, the message is passed to the active task and processing continues. If the message is not available (e.g., it has not yet been sent), the active task is made inactive and the task waits for the message. The task dispatcher is then invoked to select the new active task from the TDQ. The task dispatcher is also invoked on a SEND operation if a task of higher priority than the active task is waiting for the message. If the waiting task is of lower priority than the active task, the task dispatcher is not invoked, but the processing request for the waiting task is placed on the TDQ.

I/O in System/38 is implemented with a queue-driven com-

mand structure using the SEND/RECEIVE mechanism to pass information across the I/O interface, which is described by Lewis, Reed, and Robinson [1978]. To a task, a device looks like another task. Commands to devices and responses from devices are exchanged in the same way that messages are communicated between any two tasks in the system. The messages sent to the devices are specially formatted and contain the device commands. In addition to individual commands, a complete channel program can be sent as a single message. Because a queue structure is used, command stacking is automatic. In a similar manner, the device sends response and status information back to a task via a main storage queue. Note that only commands and responses use the queueing structure; data transfers between devices and main storage are direct.

High-level call/return functions are directly supported by another set of system control operations which provide the linkage mechanism between routines executing within the same task. The performance of programs written using structured programming techniques is enhanced by the use of this mechanism. The same linkage mechanism is used by the hardware to report program exceptions. With this mechanism, exceptions for any task (including such things as page faults) execute at the same priority level as the task itself. A low priority task incurring an exception will not interfere with the execution of higher priority tasks.

Summary

The hardware implementation of System/38 provides the foundation on which the high-level machine architecture is built. Through the use of advanced LSI technologies, System/38 achieves a high level of processor performance and reliability. The use of intelligent controllers for I/O device attachments distributes the I/O workload throughout the system.

A unique aspect of the System/38 hardware and microcode is the incorporation of very powerful control functions. These functions provide a single mechanism which is used to control all processing in the system. Other high-level functions implemented in the microcode further enhance the flexibility and performance of the system.

References

Berglund [1978]; Curtis [1978]; Donofrio, Flur, and Schnadt [1978]; Dumstorff [1978]; Froemke, Heise, and Pertzborn [1978]; Houdek and Mitchell [1978]; Lewis, Reed, and Robinson [1978].