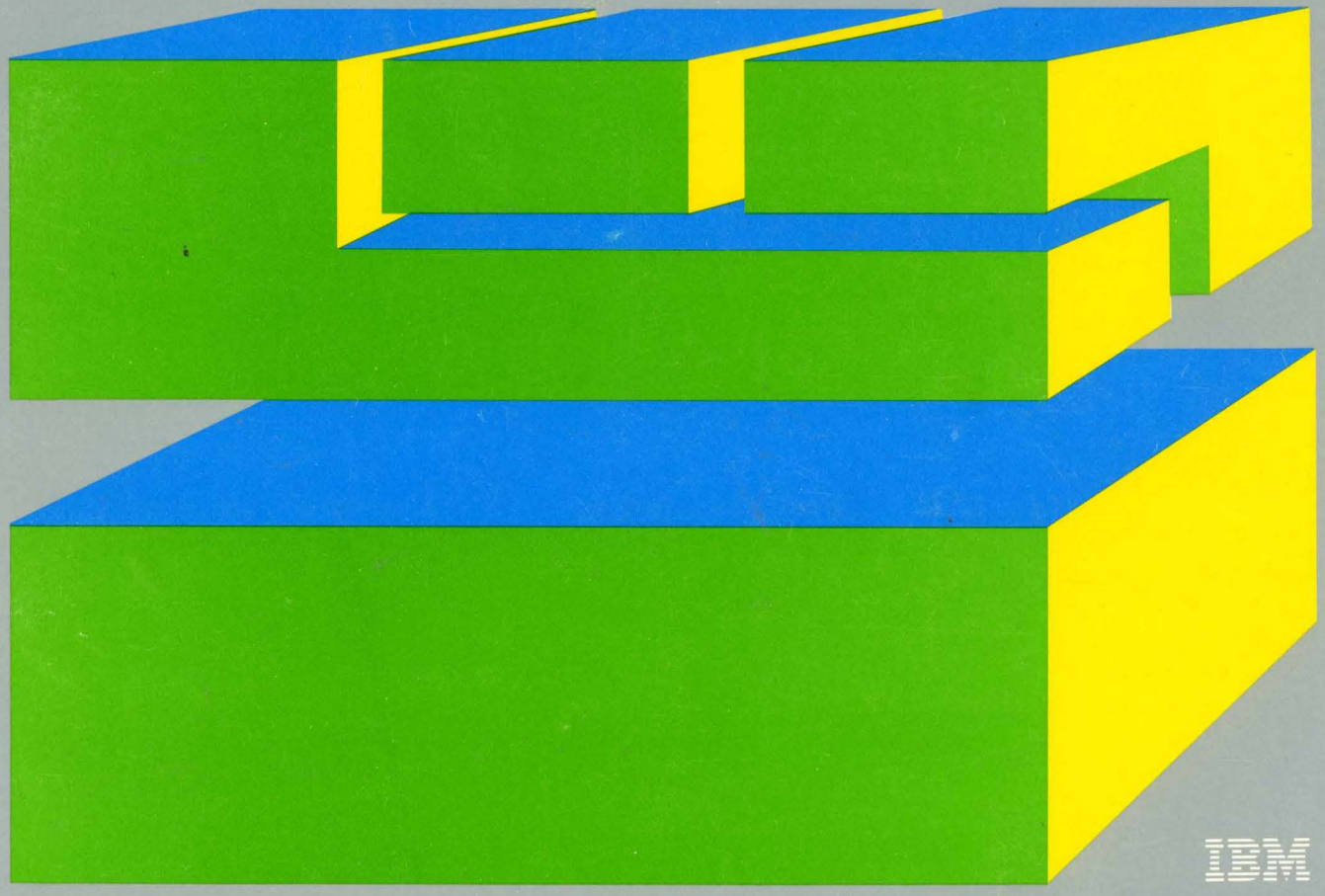
IBM System/38
Technical Developments



IBM

Preface

The IBM System/38 employs both advanced technology and many new data processing concepts. While the laboratory in Rochester, Minnesota, had primary responsibility for design and development, IBM people in laboratories in Boeblingen, Germany, Burlington, Vermont, and East Fishkill, New York, made important contributions.

Our mutual objective was to produce a system that would be both accessible and extendable, and at the same time offer efficient conversion facilities.

Function menus, help keys, multilayer messages, and a system-wide control language are essential elements of System/38, along with a flexible "user authorization" scheme for system integrity and security. We chose to avoid traditional hardware-dependent addressing and storage management and instead to readily accommodate new technologies and storage organizations through a high-level machine architecture that not only frees the user from earlier restrictions but also supports a new kind of data base facility.

Finally, because System/38 is viewed as a growth path from present systems, especially the IBM System/3, we developed conversion techniques rather than an emulator to give these users an opportunity to benefit from the novel, even unique, capabilities of the System/38.

Some 50 authors are represented in this special collection of papers. I want to thank them and their many colleagues whose combined efforts made System/38 a distinguished family of IBM products.

B. G. Utley
Manager, GSD Advanced Systems

Second Edition (July 1980)

This is a major revision of and obsoletes G580-0237-0. Changes include minor technical changes and an update of authors' biographies.

The new cover design reflects a high-level view of the architecture of System/38.

The papers in this volume are not intended to replace IBM publications in describing the capabilities of the system components and how to use them. Keep in mind that the papers are for general technical communication purposes; they do not represent an IBM warranty or commitment to specific capabilities in the referenced products.

Different structures and levels of detail may exist in the papers because they were written as technical articles by various developers of the System/38. In order to preserve their technical integrity and vitality, they have not been integrated relative to consistency of style, language, or method of presentation.

Note that these papers will not be updated as changes are made over time to the System/38 products.

IBM Product design and development
General Systems Division

IBM System/38 Technical Developments

IBM S/38 TECH DEV, pp. 1-110, 1980

Table of contents

 ISBN 0-933186-03-7. Printed in U.S.A.

HIGH-LEVEL MACHINE STRUCTURE

SOME ADVANCES IN PROGRAMMING SUPPORT

States that new concepts in System/38 architecture include a layered structure providing consistent interfaces, a unique high-level machine architecture, and capabilities for virtual addressing and task management. Support functions are summarized and architectural concepts are described.

Introduction to IBM System/38 architecture

G.G. Henry

The IBM System/38 is a new general-purpose data processing system designed to provide a high level of function, ease of use, reliability, serviceability, and nondisruptive growth. It supports advanced data base and interactive work station applications as well as traditional batch applications. These extensive capabilities are made possible by the use of novel architecture and design concepts, advanced technologies, and new implementation of system components, both hardware and software. The new concepts include a layered structure providing consistent interfaces, a unique high-level machine architecture, and powerful capabilities for virtual addressing and task management.

This paper first summarizes the user support provided by the System/38 components and then introduces some of the salient architectural concepts that pervade the design of System/38.

System function

System/38 consists of a machine and three major IBM licensed programs: Control Program Facility (CPF), RPG III, and Interactive Data Base Utilities (IDU). The CPF provides operating system functions to other programming components and to the end-user. RPG III is an enhanced version of the well-established RPG II language. IDU provides interactive data entry, source language entry, and query functions. These components fit together to provide a comprehensive and cohesive set of capabilities oriented to support advanced user requirements.

Figure 1 shows a high-level view of the system, including the dependency of all programming upon the machine instruction set and the dependency of RPG III and IDU upon CPF functions.

Some key characteristics of the user-oriented support are:

- Extensive data base facilities providing field-level described data, program independence from physical file structures through use of logical files, and simultaneous access to data by multiple users. The data base concepts are discussed by Watson, et al [1].
- Flexible work management functions supporting several different application approaches and allowing dynamic sharing of storage and other system resources. Work management aspects are treated by Norton, et al [2,3].
- High levels of device independence provided to programs—including screen formats and local/remote transparency. Various aspects are covered by Truxal and Ridenour [5], and Fess and Benson [12].
- New control language and data definition interfaces providing consistent access to all control program and utility functions. These are discussed by Botterill and Evans [4], and by Truxal and Ridenour [5].
- Comprehensive and integrated authorization facilities, which are reviewed by Berstis, et al [6].
- Powerful program development facilities such as library, test and debug, source maintenance, and data base utility functions.

The remainder of this paper addresses the key concepts of System/38 architecture.

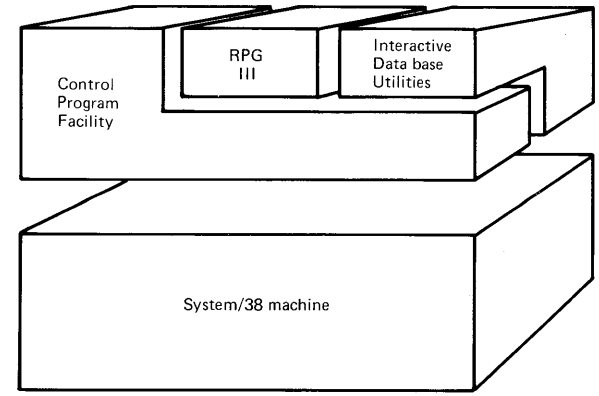


Figure 1 A high-level view of the architecture of System/38

System-oriented design and implementation

A *total-system* approach was used for the architecture, design, and implementation of System/38. That is, system-wide design trade-offs were made during the design and implementation of *all* components of the system. This approach toward eliminating potential problems of design mismatch was supported by imposing no *internal* compatability constraints with previous systems. Furthermore, almost all design and implementation of hardware and programming was done in the same location by a single organizational entity. This total system trade-off and design process has resulted in a high degree of fit between system components, thus eliminating redundancy and unused function.

Unified function

The combination of the System/38 machine and the programming products provides a very high level of function, such as extensive data base facilities. Typically, these kinds of advanced functions have been provided on other systems by discrete "subsystems," each having different user interfaces, specific configuration restrictions, special resource tuning requirements, and separate installation and service characteristics. Such a subsystem approach is partly the result of the lack of system-oriented design and implementation and the lack of well-defined architectural structures.

In System/38, the total-system design approach allows these advanced functions to be integrated into a single machine and a single control program with a single user interface to all functions on all system configurations. That is, *all* control program functions (other than I/O dependencies) are automatically available in *all* System/38 installations. Furthermore, no "system generation" or complicated tuning procedure is required to adapt the programming products to different machine configurations.

Figure 2 shows the System/38 integrated structure contrasted with a general subsystem structure.

Layered system structure

This approach to system-wide design and integrated function is supported by the layered structure of the system. System/38 support is structured into horizontal layers, each providing a consistent interface that is not dependent upon implementation details of the other layers.

Most systems have a layered structure to some degree. The significant concepts of the System/38 approach are that there is only one interface for any general type of usage; this interface is designed such that all functions are presented in a consistent and extendable fashion, and the interface does not require or *permit* use of implementation details of the next lower level of system support.

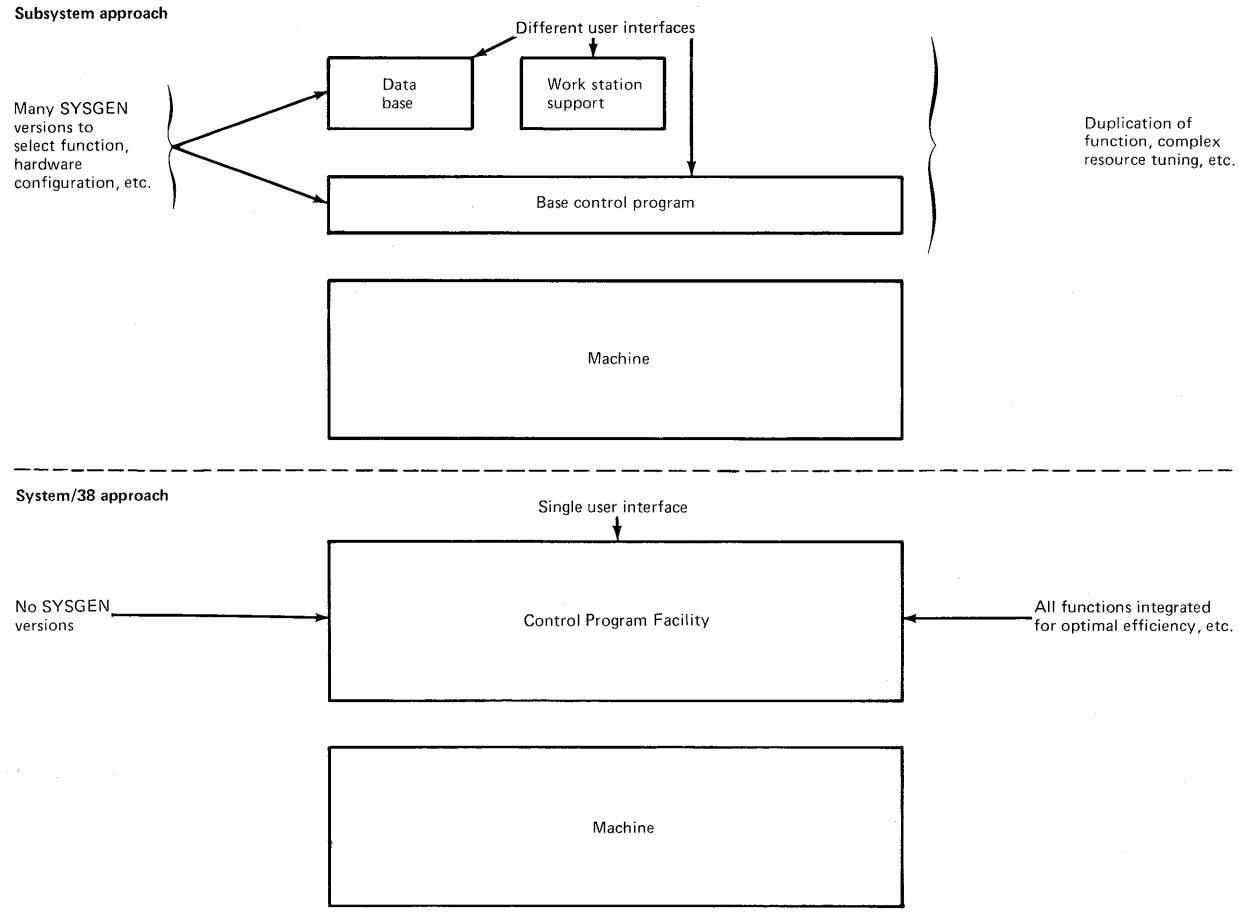


Figure 2 Unified structure of System/38 design

The System/38 RPG III and Interactive Data Base Utilities provide traditional high-level language and utility interfaces.

The Control Program Facility [7] executes on the System/38 instruction interface and provides three interfaces, each providing consistency and implementation-independent characteristics similar to those of high-level programming languages.

- A single new control language [4] provides access to *all* end-user CPF execution functions through a consistent and extendable high-level interface. This is in contrast to having one specialized control language for the system operator, another for the work station user, a third for the programmer, a fourth for the user of a system utility function, and so forth.
- A single new data and file definition language [5] supports both data base and device file definitions for all devices. This permits easy interchange of devices or device files with data base files without modifying programs or file definitions.
- The data management function interfaces [12] used by both IBM and user programs provide high-level capabilities and consistency across all devices and data base functions.

High-level machine architecture

Just as the program product interfaces are improved in structure and consistency over previous such interfaces, so is the machine instruction set interface. The primary characteristic of System/38 is its unique high-level machine interface [8, 9]. This provides many of the basic supervisory and resource management functions previously found in operating systems.

Examples of the System/38 advanced instruction set functions include physical record level data management, tasking management, queue management functions, generic and late-bound computational functions, and high-level program linkage functions. To provide these functions, it was necessary to employ new architectural concepts. For example, the instruction interface addressing structure, discussed by Dahlby, et al [9], is an "object-oriented, uniformly addressable store"; that is, all objects reside in storage and all storage on the system can be addressed with a single device-independent addressing mechanism. Furthermore, the addressing mechanism incorporates integrity and authorization checking [6] for valid usage. (The use of objects is discussed by Pinnow, et al [10], and machine storage management is discussed by French, et al [13].)

This generalized addressing scheme demonstrates one of the most critical general characteristics of the System/38 instruction set. Its structure is independent of underlying implementation characteristics such as hardware registers, physical I/O access mechanisms, detailed data formats, and control block structures.

The significance of this approach is that it extends the advantage of high-level language interfaces to lower levels of programming; namely, the high-level nature of the System/38 instruction set provides the capability to make design and implementation changes to the hardware and functions implemented in microprogramming without affecting IBM or user programming. In addition, by placing advanced functions such as data base in microprogramming, a tight fit with the hardware structures can be achieved.

The concept of a layered system structure extends into the machine itself. There are three layers of support: the physical hardware, and two layers of microprogramming. The boundaries between these layers represent internal design criteria and are not available for external use.

Figure 3 illustrates the total System/38 structure, including the internal machine divisions.

Several new hardware technologies are used in System/38 [14, 15, 16]. The combination of advanced capability provided by these technologies and the system-wide design approach result in new hardware structures and capabilities. For example, very powerful virtual addressing capabilities [11] and task management support functions [8] are provided by the hardware.

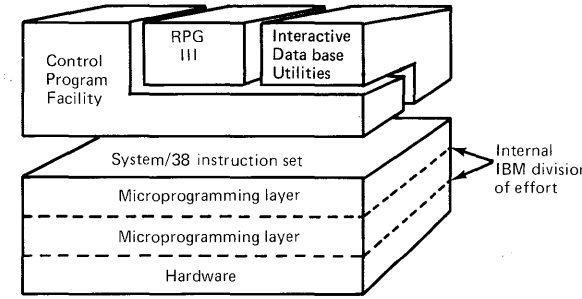


Figure 3 The total structure of System/38

Summary

From a user viewpoint, System/38 offers a high level of function and ease of use. This is made possible by the use of advanced approaches in the underlying structure, design, and implementation of the system components. The salient elements are:

- Use of new hardware technology
- Application of system-wide architecture concepts across all system components
- Implementation of advanced technical approaches

The papers that follow address some of these technical elements in more detail.

References

1. C.T. Watson, F.E. Benson, and P.T. Taylor, "System/38 data base concepts," page 78.
2. H.T. Norton, R.T. Turner, K.C. Hu, and D.G. Harvey, "System/38 work management concepts," page 81.
3. H.T. Norton and T.R. Schwalen, "Table-driven work management interface in System/38," page 94.
4. J.H. Botterill and W.O. Evans, "The rule-driven Control Language in System/38," page 83.

5. C.D. Truxal and S.R. Ridenour, "File and data definition facilities in System/38," page 87.
6. V. Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.
7. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
8. R.L. Hoffman and F.G. Soltis, "Hardware organization of the System/38," page 19.
9. S.H. Dahlby, G.G. Henry, D.N. Reynolds, and P.T. Taylor, "System/38–A high-level machine," page 47.
10. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.
11. M.E. Houdek and G.R. Mitchell, "Translating a large virtual address," page 22.
12. R.O. Fess and F.E. Benson, "File processing in System/38," page 91.
13. R.E. French, R.W. Collins, and L.W. Loen, "System/38 machine storage management," page 63.
14. H.W. Curtis, "Integrated circuit design, production, and packaging for System/38," page 11.
15. M.N. Donofrio, B. Flur, and R.T. Schnadt, "Memory design/technology for System/38," page 16.
16. N.C. Berglund, "Processor development in the LSI environment," page 7.

Presents the problems of testing in the LSI environment and describes the design techniques used in the development of the System/38 to solve the testing and test generation problems.

Processor development in the LSI environment

N.C. Berglund

The IBM System/38 central processor is implemented with IBM's new high performance large scale integration technology. This technology uses the master-slice concept wherein each chip contains a fixed number of logic circuits of various types—receivers, drivers, nands, etc.—which the system logic designer interconnects to perform a function. The processor is made of many such chips, each uniquely personalized, as described in the article by Curtis [1], to perform a portion of the total function. The processor, which consists of 29 LSI logic chips with approximately 20,000 circuits and five arrays, is packaged on one planar board, 10 by 15 inches. This concentration of function presents significant new problems in many phases of development and manufacturing. In particular, the requirements of manufacturing testing must be considered on a par with the needs of the system designers.

The principal problem in designing with LSI is the inaccessibility of internal signals. This is critical to problem isolation during the initial debug of engineering prototype hardware, during manufacturing testing, and in the customer environment. The conventional techniques used in the past involved testing the chips with complex sequential patterns which would attempt to exercise all the internal circuits and to propagate the state of internal signals to the output pins of the chip where they could be observed. This process is too complex to lend itself to efficient utilization of program-generated test data. System/38 uses a design system called level sensitive scan design (LSSD) to solve problems of testing and test data generation at all levels of packaging—chips, boards, and system. The LSSD technique allows the LSI chips (Figure 1) to be completely tested for dc faults using computer-generated test data.

The LSSD technique

In the LSSD technique, the only type of storage element (other than arrays) permitted in a logic design is called a shift register latch (SRL), shown in Figure 2. An SRL is a pair of polarity hold latches (type D) with the output of the first latch, called L1, permanently connected to the data input of the second latch, called L2. The L1 latch is a functional storage element to be used by the system designer. The purpose of the L2 latch is to improve the effectiveness of chip testing. The L1 and L2 latches, as connected, form a single stage of a shift register. The L2 latch has a single data input which is connected to the output of the L1 latch and a single clock input, called the B clock, which is used to load the L2 latch from the L1. The L1 latch can be set from two sources because it has to function as part of the test system and as a storage element for the system designer.

One input, called the scan data input (SDI), is reserved to be connected to the output of another L2 latch on the LSI chip. A clock input, called the A clock, is used to clock data from the SDI into the L1 latch. The other input is the normal functional data input used by the designer. A separate clock input, called the system clock input, is used to load data from this input. All the SRLs on the entire chip are connected together into a long shift register by connecting each L2 output to another L1 SDI. The first L1 latch in the shift register is connected to a chip input pin which is designated as SDI for this chip. The output of the last L2 in the shift register is connected to a chip output pin which is designated scan data out (SDO) for this chip. The A and B clock inputs of each SRL are connected in common to a pair of chip input pins designated as the A and B clock inputs (scan clocks) for this chip. The designer has lost the use of four chip pins and the circuits required to implement the L2 latches and associated clock drivers, but the connection of the SRLs into a shift register in no way interferes with the normal functional operation of the chip. When the chip is tested, the four pins and the L2 latches enable the test system to control and retrieve the contents of any storage element on the chip by means of a simple shift technique.

Reprinted from Electronics, March 15, 1979; Copyright © McGraw-Hill, Inc., 1979.

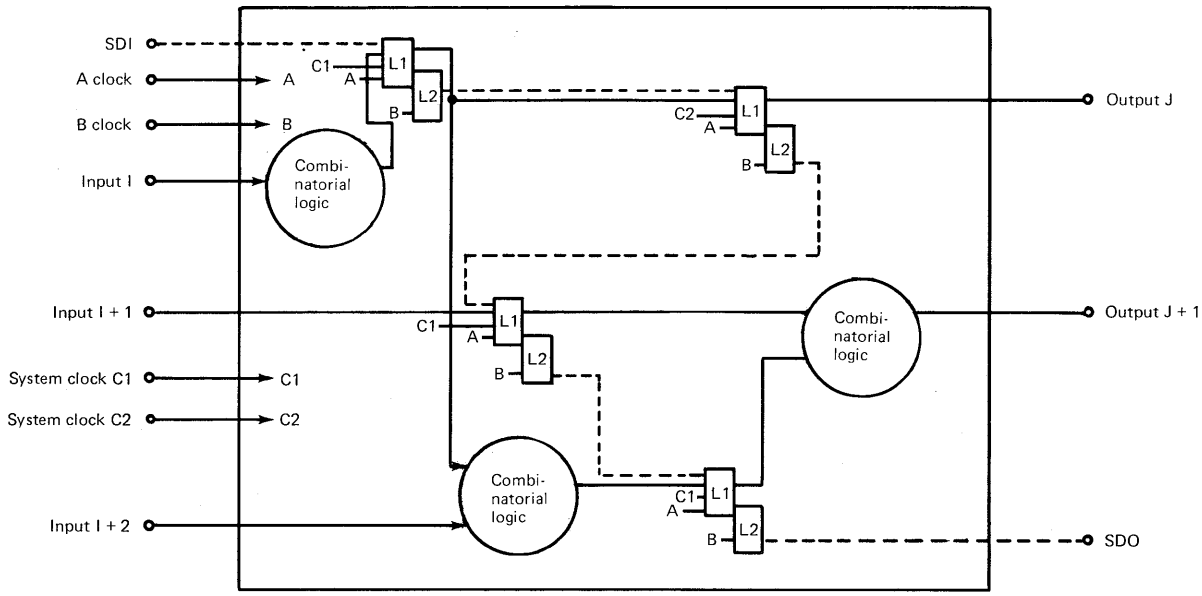


Figure 1 Typical LSSD LSI chip

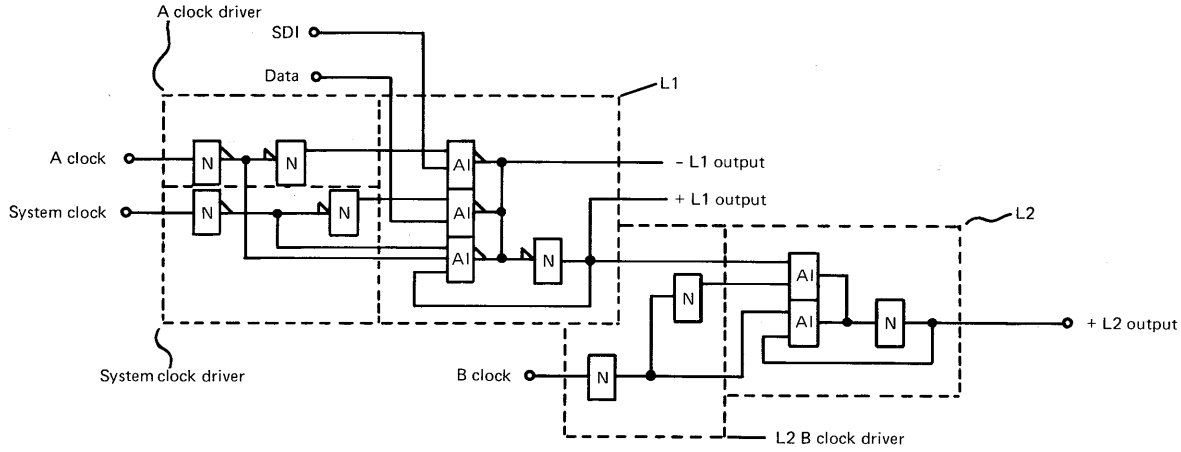


Figure 2 Shift register latch

Chip testing

When a chip is tested, a test system provides a test pattern, which is a serial string of binary data, to the SDI pin of the chip. It operates the scan clocks (A and B) causing the test pattern to be shifted (scanned) into the SRLs on the chip. This test pattern, which now resides in the latches on the chip, and stimuli applied to the chip input pins cause the combinatorial logic on the chip to take some particular state. Some of the combinatorial logic is connected to SRL data inputs and some is connected directly to chip output pins (Figure 1). The output pins can be observed to determine if the combinatorial logic is functioning properly; but, to test the logic which is connected to SRL data inputs, the system clocks must be applied to transfer the state of the combinatorial logic into the SRL L1. The test system applies a pulse to some or all of the functional clock inputs of the chip and this changes the state of some or all of the SRLs. The test system applies the scan clocks once more, this time observing the serial binary data coming out of SDO for this chip. This data represents the state of the SRLs after the system clocks were operated (which is the same as the state of the combinatorial logic before the system clocks were applied). The data is compared to the expected state of the SRLs as determined from a simulation model. In this manner the logic on the chip is tested for typically 98% to 100% of all dc faults with program-generated test data.

No complex sequences of system clocks are necessary to test all stages of counters, shift registers, etc., which are buried in the logic of the chip. When using the LSSD technique, patterns are loaded to test all stages of counters, etc., without stepping the counter through all its states. Each system clock will be pulsed no more than once per test pattern, and this will be sufficient to test the combinatorial logic connected to the data input, the clock driver of the SRL, and the SRL itself.

Circuit cost considerations

But what of the cost of such a system? On first

analysis, the LSSD system appears to carry a significant overhead in unusable circuits. The extra input to the L1 latch, the L2 latch, and the extra clock drivers do require circuits which are then unavailable to the designer for his unrestricted use in implementing the processor function. These circuits represent the hardware cost of LSSD and they can approach 20% of the available circuits. These circuits do not, in fact, have to remain strictly as overhead because they can be used for implementing the processor function and for several other features in addition to their use in the LSSD system.

At the chip level, for example, the L2 latch can be used to make functional shift registers, counters and control latches. This is accomplished by logically OR-ing a system clock with the B clock input to the L2 latch. The A and B clock inputs are used only when the chip is tested so no interference exists if the L2 is used functionally. When combined with a 2-phase, non-overlapping clock, the L1 and L2 latch acts as a master-slave storage element which can be used to implement any function that can be implemented with the more traditional storage elements (J-K flip-flops, etc.). Furthermore, the L2 latches of a register provide a double buffer function which has many uses such as a backup register for retry or as a double buffer for data storage.

The 2-phase clock system, while appearing to present performance disadvantages for counters, etc., (since two clocks are required to advance a counter or shift register by one position), can actually be used to advantage where overlapped processing is used. Frequently, in high performance processors, the execution of the next cycle will begin before the current cycle completes. The SRL is uniquely suited to this since it consists of two independently clocked latches. The L2 can be used to hold the information necessary for the current cycle while the L1 is loaded to begin the next cycle. In the System/38 processor, more than 85% of the L2 latches are used functionally; these and other uses to be described serve to overcome the hardware cost of LSSD.

Test patterns

By extending the LSSD shift register concept to the planar level, the board can be tested in a manner consistent with the means for testing chips. All the chips on the planar are connected into one long shift register, as indicated in Figure 3, by connecting the SDO of one chip to the SDI of another. SDI of the first chip and SDO of the last chip in the shift register are connected to planar input/output pins. Test patterns for the entire planar are computer generated in the same manner as for the chips. In practice, the chips are grouped into several shift registers of shorter length which are loaded in parallel to reduce test time. The planar is inserted in a test fixture, test patterns are loaded into the SRLs of every chip, stimuli are applied to the planar input pins, and the system clocks are pulsed. The output pins are measured, and the SRLs are scanned out; both are compared to the expected results. Essentially, no additional hardware is required to support planar test because the LSSD hardware in each chip is also utilized for this purpose. This technique tests the entire processor before it is installed in a system.

The same technique is applied again at the next level of packaging. The LSSD technique is used at the system level in the customer environment to provide a processor checkout each time the machine is turned on, or when necessary to aid service personnel in problem isolation. The testing problem in the field is more complex, however, since the planar is mounted in a system and not in a test fixture. In the system, the planar signal pins are connected to channels, memories, etc., hence, they are not directly observable. But the processor SRLs can be controlled and they are connected to 90% to 95% of the logic, so an effective test can be performed. To extend the LSSD concept to the system level, the planar I/Os—SDI, SDO, A and B clocks—are connected to the system control adapter (SCA). The SCA is a separate microprocessor (not on the planar) used to perform several system maintenance-related tasks. The SCA has the ability to provide serial data on the SDI and the ability to observe the serial data on SDO while pulsing the A and B clocks to the shift registers on the planar. Hence, the state of nearly every storage latch on the entire planar can be observed and

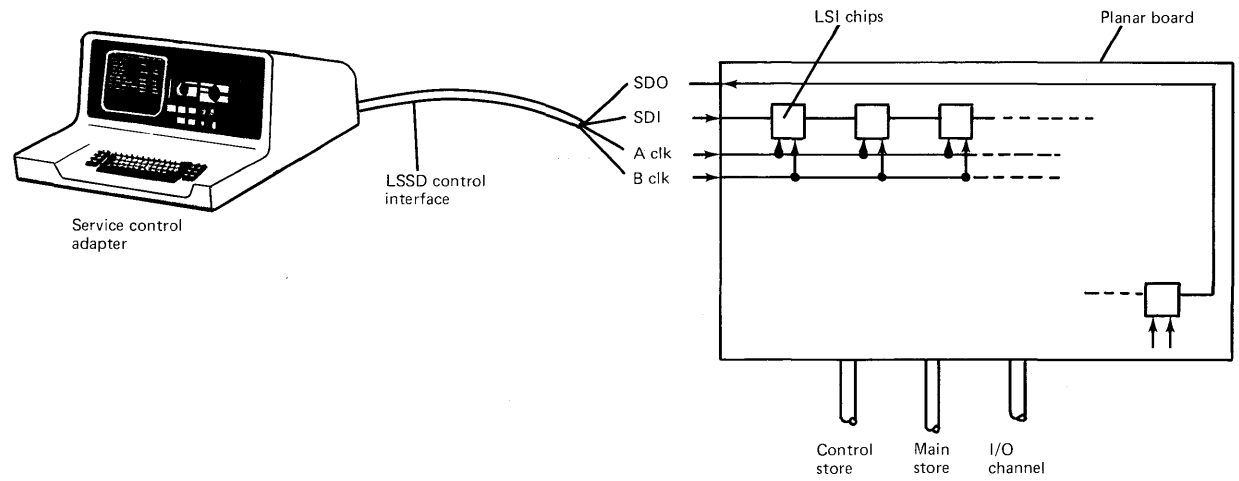


Figure 3 System/38 planar board and interfaces

controlled from the SCA. A small amount of maintenance interface logic on the planar gives the SCA the ability to shut off the system clocks to the processor and to pulse those same system clocks for test purposes. (The latches in the maintenance interface logic cannot be scanned because they function in conjunction with the SCA to allow the latches of the processor to be scanned.) With this support, the SCA reads test patterns from a system file, loads them into the latches on the planar, pulses the system clocks, retrieves the contents of the latches, and compares the results to the expected results that are also obtained in the file.

Historically, processors are tested in the field with diagnostic programs. With the LSSD technique, test patterns can set up conditions to test for specific faults much easier than can be done by diagnostic programs which are limited to the capabilities of the machine instruction set. LSSD patterns provide, however, only a dc test of machine operation. Time dependent or ac problems must still be located with diagnostic programs operating at machine speed. The combination of LSSD patterns and processor diagnostics provides an effective means to verify proper operation.

Console operations

Since the shift registers used in the LSSD concept provide a way of altering or displaying the state of every storage element in the processor, they lend themselves to the support of console manual operations. Typically all systems will have a console for displaying the contents of registers, memory, and critical control latches as an aid for diagnosis of hardware and programming problems. System/38 provides this function by using the LSSD shift registers as the means for displaying and altering machine registers from the console.

The SCA has access to the latches through the same mechanism it used to conduct the LSSD test of the planar. Maintenance interface logic on the planar gives the SCA the ability to bring the processor to a controlled stop, while the SRLs are scanned, and to restart the processor when the scan is complete. The SCA scans the data from the shift registers and formats it for display on the CRT. If the displayed data is altered, the SCA will take the altered value and replace it in the processor by scanning new contents into the machine latches. This technique saves the extra hardware normally required to get into and out of the facilities to be displayed and altered. This technique further provides the capability to alter and display every latch in the processor.

Concluding remarks

In System/38 IBM has found the LSSD technique to be a cost-effective solution for processor design in the LSI environment. LSSD, while conceived to solve LSI chip test problems, has been used to provide an integrated test and maintenance approach from the chip to the system level. The circuit overhead and inherent restrictions of a single storage element design are overcome with no significant sacrifice in cost or performance through functional use of the L2 latches and by capitalizing on the unique characteristics of an LSSD design. Program-generated test data, high test coverage, and the ability to observe and control every latch are valuable LSSD attributes which were used to reduce the cost of both development and manufacturing and to raise the quality of the shipped system.

References

1. H.W. Curtis, "Integrated circuit design, production, and packaging for System/38," page 11.

Discusses steps in the production and packaging of logic chips for System/38. Includes discussion of circuit and chip topology, processing of the master slice, design automation, and packaging.

Integrated circuit design, production, and packaging for System/38

This survey paper on System/38 semiconductor component technology represents the summation of technical contributions made over many years by East Fishkill Development and Manufacturing personnel too numerous to mention.

Huntington W. Curtis*

The announcement of System/38 provides the first public disclosure of a new level of compatibility in bipolar, integrated-circuit, array and logic technology. Based on Schottky T^2L circuitry with a nominal delay of about 3 ns per gate, the new logic chips are less than 25mm square, contain up to 704 logic gates plus more than 60 off-chip driver circuits, employ three layers of interconnection wiring above the silicon surface, and are physically attached to ceramic single-chip carriers by 132 solder connections. A unique feature at this level of integration is the logic designer's ability, through a corporate-wide design automation system, to request almost any desired interconnection of all or part of the 704 available logic circuits. This is accomplished through the production use of electron-beam direct exposure of photoresist-coated wafers. Interchangeable with optical mask technology, the electron beam is used at several process steps to avoid the use of masks and their attendant fabrication time and yield problems.

All logic chips, regardless of ultimate system function, are produced with the same optically defined device patterns in the silicon; a silicon wafer containing these repetitive device structures in each chip area is termed the "master slice." As the need for specially designed "part number" circuit configurations arises, "personalization" through metallic interconnection between devices and between individual circuits is accomplished using electron-beam photolithographic technology. The logic designer's flexibility, using this "open part number set" approach, enables particularly efficient use of silicon and of packaging space and materials.

In the material which follows, logic circuit and chip topology will be discussed, the geometrical form and dimensions of devices and metallization will be identified, and design automation capability will be outlined, highlights of the electron beam's role will be described, and the method of single chip packaging will be illustrated.

Circuit and chip topology considerations

Each chip in the System/38 master slice contains a total of over 7,000 resistors, diodes, and transistors. These devices are arranged in a series of narrow bands across each chip, and take up a total area less than one-half the chip area. The bands are divided into rectangular circuit component areas. The slightly wider rows separating the device bands are used as channels along which the first level of wiring is placed. The second level of wiring is placed at right angles to the first, and a second level conductor may be electrically connected to the first level wiring at any intersection by providing an etched via at the desired location in the insulating layer between the conductors.

First level wiring directly above the device bands can be configured to provide any of over 100 logic functions in each circuit area to meet design needs. Of these possibilities the predominant logic circuit in System/38 chips is a Schottky T^2L gate with either a three- or four-emitter input transistor, the two associated Schottky barrier diodes, and four resistors. Power for these logic gates is supplied from a 1.5 volt bus. An additional power supply level, 4.5 volts, is used by emitter-follower off-chip drivers, necessitated by the increased IR drops and capacitance of the long path lengths compared to the on-chip interconnections.

One further configuration consequence warrants discussion: the problem of how, with a limited number of off-chip connections, to test a chip completely. Of the electrically conducting solder connections to the chip, 94 are available for signal input/output use; the remainder are allocated for power supply distribution. At a much lower level of integration than employed here, a chip may provide only combinational logic, in which each signal pattern applied to the inputs determines a corresponding unique signal pattern at the output ports; for this case, a test pattern for complete testability may readily be derived. At the 704-circuit LSI level available in System/38, however, sequential logic within a chip must be employed; that is, memory registers or latches store intermediate results of several levels

*The author was associated with the IBM System Products Division (now the Data Systems Division) in East Fishkill, NY, when this paper was written. He is now at IBM IRD Medical Systems, Mt. Kisco, NY.

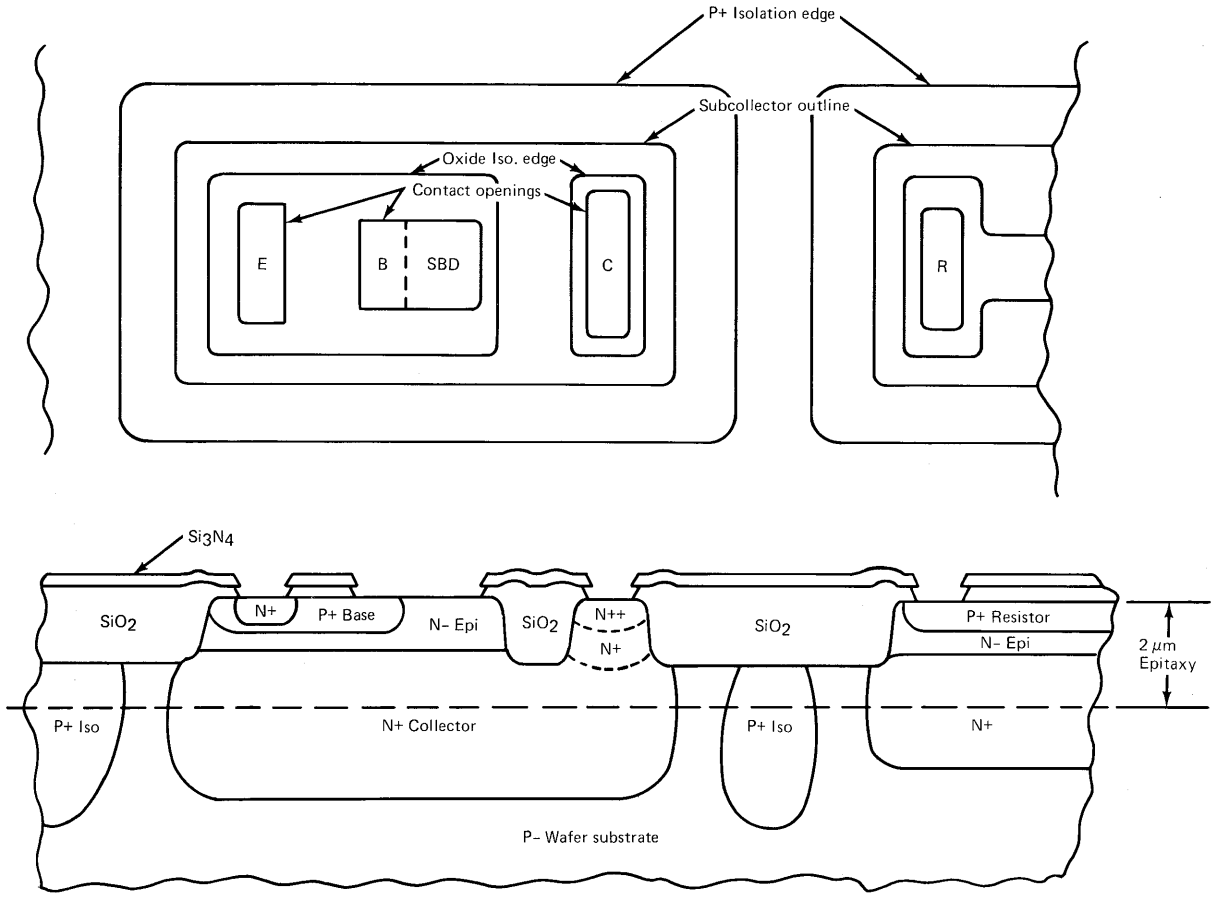


Figure 1 Top and section drawing of transistor, Schottky barrier diode, and resistor

between input and output. Testing by output-input comparison in this case would be prohibitively lengthy, even for a single chip.

This problem has been solved by additionally connecting all required latches as a serial shift register. Not used as such in the normal computer logic applications of a chip, the shift register latch function is available in testing to read in initial settings, or to read out memory states after one logical operation, thus simplifying the testing problem from sequential to combinational logic. Only four terminals (input, output, and two shift clocks), are required for this important feature, leaving 90 I/O ports for the logic designer's use.

Master slice

Starting with polished 82 mm diameter wafers cut from single crystal silicon, the familar steps of subcollector and subisolation diffusion, epitaxial silicon layer growth, collector reach-through diffusion, base and resistor diffusion, and emitter diffusion are carried out. Appropriate sequences of silicon oxidation, application of photoresist, exposure to light through mask patterns, and development and etching determine the location, type, and spacing of the devices within each chip area. Figure 1 depicts the resulting configuration of a typical transistor and Schottky diode clamp, and also the contact at one end of a resistor, after master slice processing has been completed. Some typical dimensions are as follows: epitaxial layer thickness 2 μm; transistor emitter top surface, 3 μm x 8 μm; resistor 4.5 μm x 70 μm; Schottky diode, 5 μm x 6 μm. From these dimensions it should be evident that, although most of the processing steps are conventional, many improvements in process control and photolithography precision have been achieved to provide the required circuit density.

The last steps in master slice processing include etching openings in a blanket surface insulating layer for later contact to devices by metallic conductors in the personalization process, vacuum depositing a thin platinum layer over the entire wafer, and sintering to form platinum silicide in all contact openings, thus simultaneously providing ohmic contacts and the Schottky barriers. After unreacted platinum is removed by etching, the wafer is placed in stock until required for personalization.

Design automation

In the development of System/38, use is made of an engineering design system at two levels of packaging: First, the interconnection of the devices and resulting circuits within the master slice chip areas to provide desired logic chip part numbers; and second, the design of the wiring patterns of the next level of packaging, i.e., the planar board on which the logic and array single chip modules are mounted and are interconnected. The objective of this system is to minimize manual intervention during the design process. Identical versions of this engineering design

system are now used at more than 25 IBM locations worldwide.

The chip designer's input to the system is a description of the logical functions that a chip is to perform. The automated system provides a logic diagram as a printed output, through simulation performs a design verification, and, when this meets the designer's requirements, does the necessary transformations to generate the physical design of shapes and patterns and their precise locations on each of the three variable-format mask levels needed in the personalization process. In addition, the system generates the complete set of data required for functional testing of each chip.

In performing the automated design function for a particular master slice type, the technology parameters for that type are stored as a set of rules. The physical design of a logic chip resulting from a designer's input goes through the following steps.

Preliminary checking for possible rules violations such as logic errors, fan in or fan out violations, exceeding chip circuit count, or exceeding chip input/output connection count.

Automatic placement of logic circuit gate locations, to minimize wiring channel use and to allow for maximum circuit utilization (out of the 704 possible circuits in the case of System/38).

Automatic wiring which, together with the placement function, results in the decision of the final location for all interconnection patterns on the chip.

Shapes generation which, through use of a graphic language, identifies the optical mask or electron-beam patterns needed by manufacturing in appropriate personalization process steps. As part of this shapes-generation step, computer based checking is performed for possible rules violations on spacings, overlaps, or other shape constraints. This checking function includes a physical net check to assure that allowable signal line voltage drops are not exceeded, and that maximum capacitance rules affecting circuit speed are not violated.

Test generation is also performed automatically, and this digital information, together with the results of the other design steps, is consolidated as part of a single magnetic tape. This information is transmitted to the manufacturing location, providing all information needed for rapid fabrication of engineering or production quantities of a chip "personalized" to meet the unique needs of the designer.

Personalization

In this portion of the integrated-circuit manufacturing process, chip sites on master slice wafers receive interconnection wiring and terminal metallurgy, followed by functional testing. Information on the shapes message from the designer for a particular part number, after transformation to electron-beam control signals, determines the location of first-level aluminum-copper metal conductors, the location of via holes through a blanket of planarized SiO_2 deposited over the first metal pattern, and the location of second-level interconnecting wiring. Another insulating layer is then deposited, and for all part numbers there is an identical pattern, optically exposed, of via holes to be etched through this layer. A third interconnecting metal pattern, also identical for all part numbers, is used principally for power distribution across the chip, and this is followed by deposition of a final SiO_2 insulating layer. Vias are etched through this final layer in a standard pattern, after which thin layers of chromium, copper and gold are vacuum deposited at 132 locations through a metal mask; then lead-tin solder is deposited through the metal mask.

During development of individual chip designs (chip "part numbers") for the processor for System/38, the system's complexity required many engineering changes for improving the balance between individual chip functions and performance in order to optimize system performance; thus, it was essential that designers receive very rapid delivery of chips for each new design. Although classical methods of optical mask production could be used for the three patterns unique to a specific part number, many production steps with their associated delays are obviated by use of direct electron beam "writing" of the required patterns on photoresist-coated wafers. In addition to the saving in time, greater precision in registration is obtained since each chip site is individually aligned at four points, as opposed to two-point alignment of an optical mask to an entire wafer.

Figure 2 shows the vertical structure added during the personalization process. The three layers of wiring are separated by deposited SiO_2 approximately 2 μm thick, and a 3 μm SiO_2 layer covers the third metal pattern. First and second level metal interconnecting patterns, each formed from a blanket deposited aluminum-copper alloy by a selective "lift-off" process rather than by subtractive etching in order to obtain better coverage over device areas, have a minimum width of 4.4 μm, with 2.5 μm spacing. In the wiring channels, first-level conductors are 5 μm wide, the via holes etched to allow first to second level connections are 6.5 μm in diameter, and second level conductors are 6.5 μm wide. This "zero overlap via" technology, made possible by extremely tight photolithography tolerances, is an important factor in successfully achieving the present wiring density. The third metal layer is also aluminum-copper, deposited to be twice as thick as the first and second layers. It is etched subtractively to provide conductors with four times the cross-section of the first and second (principally signal) conductors, thus improving power distribution capability.

The last steps in the personalization process consist of etching holes in the top SiO_2 layer to reach third-

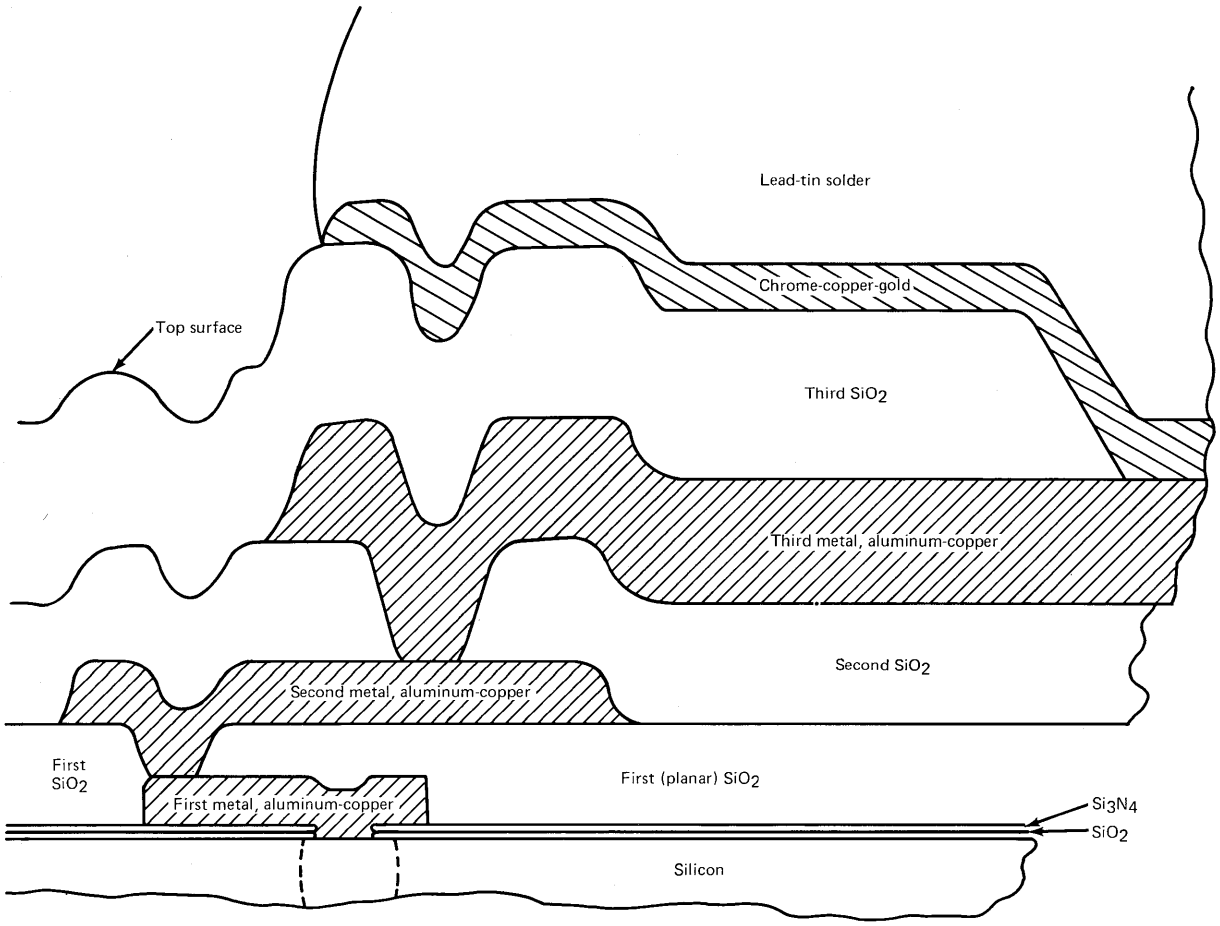


Figure 2 Three levels of metallization, showing silicon to solder ball electrical path

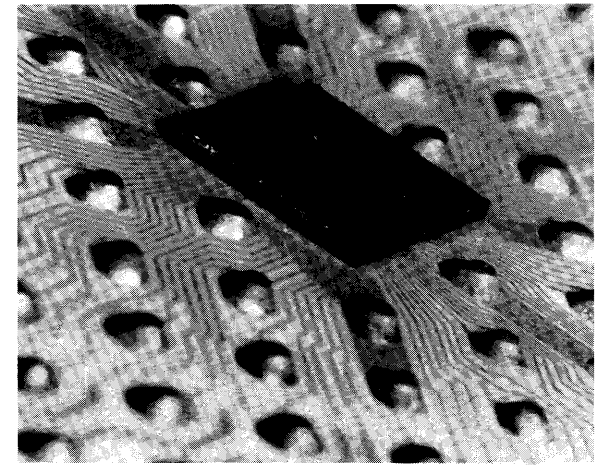


Figure 3 Closeup of mounted chip

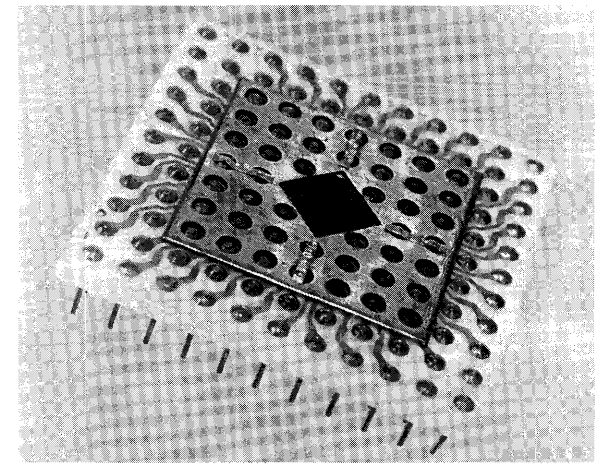


Figure 4 Single chip module before cover is added

level metal at each point where an external connection will be located and then vacuum depositing protective chromium-copper-gold metallurgy through a metal mask. Lead-tin solder is deposited through the same mask; when the mask is removed, wafers are heated until the solder flows and surface tension causes each solder pad to assume a hemispherical shape. The wafer is then ready for transfer to final test where electrical functionality of each chip is evaluated and locations on each wafer of defective chips are automatically recorded. Finally, the tested wafers are diced, followed by automatic selection and storage by part number of all good chips.

Packaging

Logic circuitry for System/38 is supplied by 29 logic chips comprising a total of 22 part numbers. Each chip has signal I/O plus power connections in an area array on the device side of the chip. The carrier for each chip is a 25 mm square of 1.5 mm thick ceramic (Al_2O_3) which has been fabricated with 116 pins for planar board mounting. Figure 3 shows the metallization pattern on the ceramic surface in the vicinity of an attached chip. The outward extension of this pattern provides the required "space expander" function connecting each of the 116 relatively wide-spaced module pins to appropriate chip

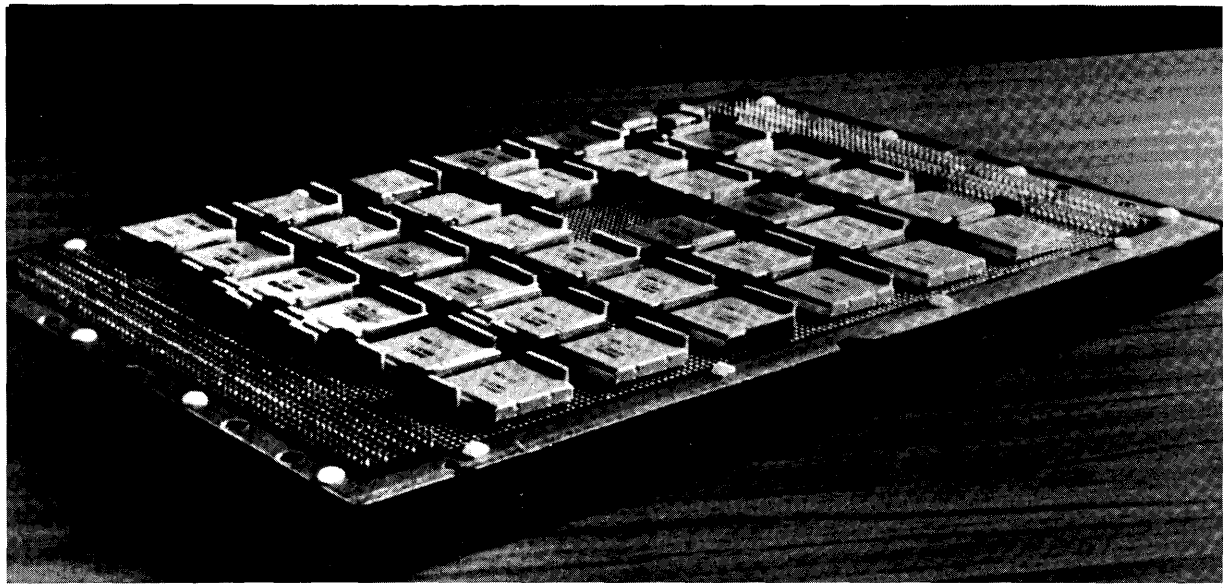


Figure 5 Planar board

contacts. Figure 4 is a picture of the module with a mounted chip and with a copper ground plane in place to minimize coupling between signal leads. The completed single-chip module is protected by an aluminum cap, is designed to dissipate about one watt, and is air cooled.

For System/38 a planar board, shown in Figure 5, provides interconnections among the logic modules and five array modules. The board is a multilayer conductor and insulator laminate with through-plated holes for module pin connections to appropriate layers. Four signal distribution planes with copper conductor patterns determined by computer instructions generated in the design automation system are combined with four fixed-pattern power distribution and ground plane layers. The copper conductive patterns are supported by and are insulated from each other by epoxy bonded glass fiber sheets. These layers are further bonded into a single planar sheet, into which the single chip module pins are inserted and soldered.

Summary

This description of some of the steps in IBM integrated circuit production and packaging for System/38 has centered on the logic chips used in the central processor. One key element in the successful use of the "open part number set" concept at the 704 circuit level of integration has been identified as a design automation system of extraordinary complexity and capability. A second is the use of electron-beam direct wafer exposure. For this powerful photolithographic tool, the principal features being utilized are its flexibility in handling a multiplicity of designs and its ability to align with great precision to each chip area being exposed; the ability to expose patterns too small for optical wavelengths is not yet required, and remains as a potential for further advances.

There are many interrelated factors involved in technology development decisions; for example, chip size, circuit density, power dissipation, design flexibility, testability, and the logistics problems of rapid response by manufacturing to engineering change requirements. Some of the problems involved increase factorially with increases in circuit count per chip; very few are made easier. The trade-offs among such factors, in order to optimize the system performance and manufacturability, are particularly complex. This article has identified some of the principal features which are important to this extension of IBM's bipolar, integrated-circuit technology.

Memory design/technology for System/38

Describes a new FET technology for a family of single-cell array, random access, memory chips ranging in density from 18K to 64K bits per chip. Shows array designs for control store module and for main memory modules.

N.M. Donofrio, B. Flur, and R.T. Schnadt*

Three random access memory chip designs of 18K, 32K, and 64K bits per chip—all of them manufactured in a new field effect transistor (FET) technology—allow fabrication of modules containing up to 256K bits. Compared to IBM's previous main memory modules, the new modules provide up to 32 times improvement in module density.

Both 32K- and 64K-bit chips are used in the System/38 random access memory cards. Each card contains 256K bytes of memory.

The 64K-bit memory chip includes circuitry that provides additional function at the chip level, such as a high speed 8-bit register that allows for improved system data rates.

The chips take advantage of the improved densities offered by the one device memory cell invented by IBM in 1967, and the added density efficiency of a new FET semiconductor technology and manufacturing process developed by IBM. In satisfying System/38 needs, the new memory array designs are utilized to cover the need for high performance control store memory and high density (low cost)/moderate performance main memory. The silicon and aluminum metal oxide semiconductor (SAMOS) process technology allows for the flexibility in performance and density offered at the array chip and module level of assembly. The array chip densities and array chip performances for these new memory chips span a factor of 2.4 change in chip density (area/bit) for a factor of 3.1 change in chip performance (access time).

This paper will briefly describe the new FET process and the various memory array design and tradeoff considerations that result in this range of new technology offerings.

Technology

IBM's newly developed FET process represents a significant departure from other FET processes. The SAMOS process is an n-channel FET process which implements metal gates, relies upon silicon nitride to enhance gate reliability relative to gate shorts, and employs a conductive polysilicon field shield to control surface leakage.

The objective of this technology was to provide high density memory chips that have acceptable performance characteristics, are small in size, simple in process, and offer optimum manufacturing yields. The process was developed and optimized for low cost memory chips by using one-device cell design (Figure 1) and minimizing the cell and chip area by: (1) eliminating contact holes in the cell, (2) using a doped oxide to provide a self-aligned diffusion source, (3) using the polysilicon field shield as the reference plate of the storage node capacitor to simplify wiring in the cell, and (4) designing to tight lithographic ground rules.

The process was developed to maximize yield by minimizing process complexity and by incorporating on-chip redundancy. Process complexity was minimized by reducing the number of discrete process and mask steps. Redundancy is implemented by a write once read only memory unit built into the second layer metal of each array chip. This memory unit is used in conjunction with appropriate on-chip address compare circuitry to allow one or more extra storage lines provided on the chips to be used to replace a like number of possibly defective storage lines identified when the chips are initially tested. This significantly enhances productivity and reduces chip cost.

*Mr. Donofrio and Dr. Flur are with the General Technology Division in Burlington, VT; Dr. Schnadt is with the System Products Division in Boeblingen, Germany.

Additionally, better reliability was achieved by using a multiple layer oxide/nitride gate dielectric and a multiple layer insulator between the first and second metal layers. An organic polymer layer is used as the final insulator layer over the second level metallization. As many as four chips are then mounted on 2.5 cm (1 inch) metallized ceramic modules using conventional IBM chip mounting technology.

Array designs

High performance control store random access memory array module. In order to satisfy the need for a high performance memory to meet the control store memory application requirements of the System/38, an 18K-bit array chip, shown in Figure 2, has been developed that provides 36K bits worth of storage in a 2.5-cm square module. The module has an access and cycle time of 140 nanoseconds and 280 nanoseconds respectively. This array chip is designed and organized to permit a module organization that meets System/38's performance, density, granularity, and reliability requirements with minimal system overhead (i.e., support circuitry requirement, cooling and space considerations).

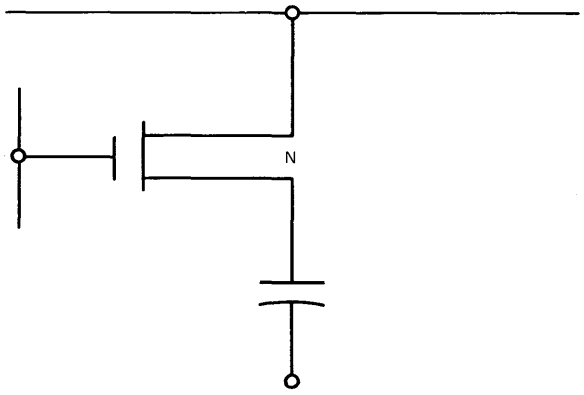


Figure 1 One-device-cell structure

Design of the 18K-bit chip included a tradeoff of density for performance. By optimizing the area occupied by a new one-device-cell design in the new semiconductor technology, maximum array signal strength is obtained in the minimum possible time, allowing for best chip/module performance.

High density main memory random access memory array modules. For main memory applications of the System/38, two array chips have been developed; one for cost/performance-driven applications and one for cost-driven applications.

The 32K-bit array chip shown in Figure 3 provides 128K bits of storage in a 2.5-cm square module at an access and cycle time of 285 ns and 470 ns, respectively. The chip organization, performance, and function additionally allows for module level characteristics that satisfy the System/38 application with minimal system overhead.

The 64K-bit chip shown in Figure 4 provides 256K bits of storage in a 2.5-cm square module with an access and cycle time of 440 ns and 980 ns respectively. This design achieves maximum density and minimum cost through minimum cell size and a new IBM-developed sensing circuit. The chip organization, performance, and function for the 64K-bit chip, as for the 18K-bit chip and 32K-bit chip, allow for module level characteristics that satisfy memory system requirements with minimal system overhead.

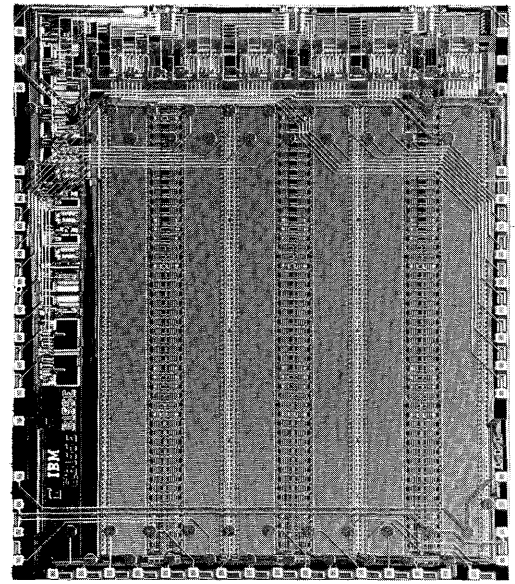


Figure 2 18K-bit array chip

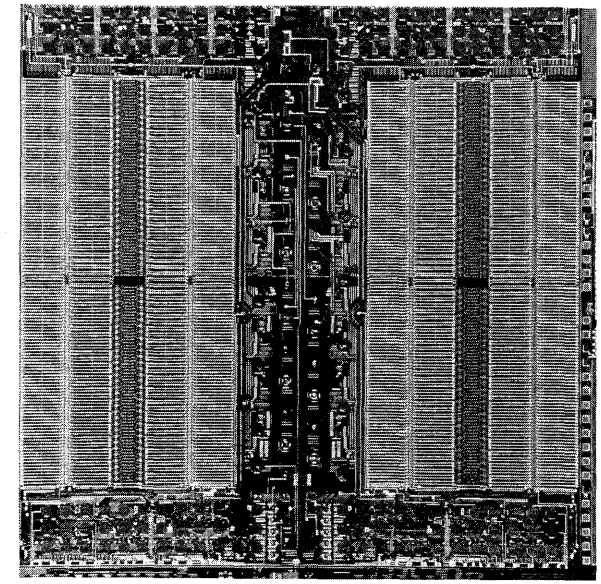


Figure 3 32K-bit array chip

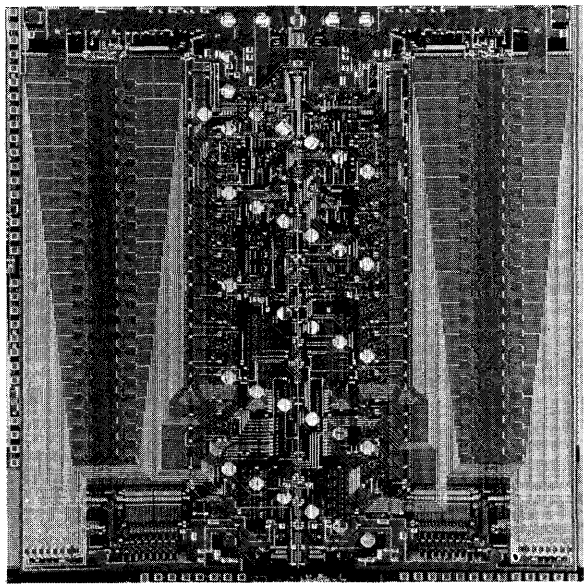


Figure 4 64K-bit array chip

Summary

By using a family of single-cell array random access memory chips ranging in density from 18K to 64K bits per chip, the System/38 takes advantage of a new FET technology to provide memory modules covering a range of 36K to 256K bits in density and 140 ns to 440 ns in performance. Design and process decisions were made to optimize performance, performance/density (cost), and density (cost) driven designs to provide the greatest flexibility to the System/38 in addressing its memory application needs.

Gives an overview of the hardware and microcode architectures for the System/38. Also describes how some functions traditionally found in programming systems are incorporated into the hardware design.

Hardware organization of the System/38

R.L. Hoffman and 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 [1], fetches 32-bit micro instructions from the random access memory control store shown in Figure 1 (8K words for both 5381 Model 3 and 5). One micro instruction is executed for each processor cycle. The processor cycle times are 400 or 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 [2] 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 [3]. 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) [4] are used for high data-rate devices such as disks, while microprogrammed I/O controllers (IOC) [5] 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 [6].

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

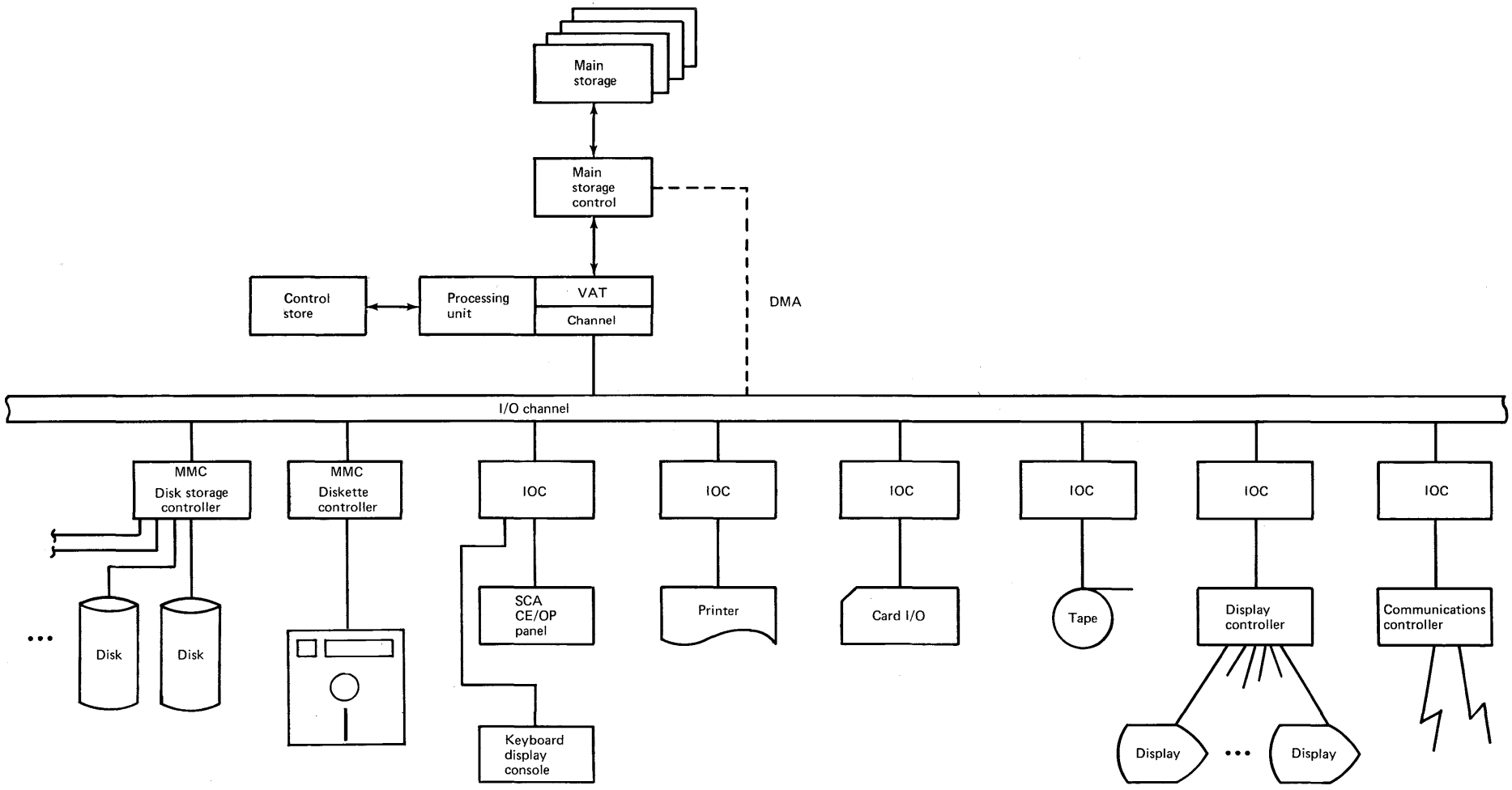


Figure 1 Hardware configuration

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 command structure using the SEND/RECEIVE mechanism to pass information across the I/O interface, which is described by Lewis et al [7]. 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

1. H.W. Curtis, "Integrated circuit design, production, and packaging for System/38," page 11.
2. M.N. Donofrio, B. Flur, and R.T. Schnadt, "Memory design/technology for System/38," page 16.
3. M.E. Houdek and G.R. Mitchell, "Translating a large virtual address," page 22.
4. J.W. Froemke, N.N. Heise, and J.J. Pertzborn, "System/38 magnetic media controller," page 41.
5. E.F. Dumstorff, "Application of a microprocessor for I/O control," page 28.
6. N.C. Berglund, "Processor development in the LSI environment," page 7.
7. D.O. Lewis, J.W. Reed, and T.S. Robinson, "System/38 I/O structure," page 25.

Translating a large virtual address

Presents the unique aspects of the virtual storage structure in the System/38. Shows the development of the virtual address translation method and explains how a large virtual address is converted to a main storage address.

M.E. Houdek and G.R. Mitchell

The System/38 supports a large virtual address space structure, large enough to contain all programs and data required by the system. To reference this space, a 48-bit virtual address yielding a 281-trillion-byte address space is implemented. This virtual space is very large compared to the portion of the virtual space that can be in main storage at any given time [1]. Since the processing unit references the virtual address space and the hardware references a physical main storage space, there must be a translation from the 48-bit virtual address to the smaller main storage address. Because the virtual address used in System/38 is so very large, the conventional techniques which have been used to translate will not work efficiently. This article describes the translation process developed for this system.

Translation

The virtual address space is divided into 512-byte blocks called "pages." When a page resides in main storage, all 512 bytes of that page are located in an area of main storage called a "frame." The part of the virtual address that uniquely identifies that page is called the page address, and the part of the main storage address that identifies the frame is called the frame identifier (FID). The part of the main storage address that identifies the byte within the frame is identical to the virtual address part identifying the byte within a page. This byte address is called the byte identifier (BID). No translation needs to take place on the BID. However, the page address needs to be translated to the FID.

The translation of the page address to the FID is accomplished by using two tables, a hash index and a page directory, as shown in Figure 1. The page directory contains one entry for every frame in main storage. The index of a particular entry into the page directory is identical to the FID for that entry. Thus, the first entry of the page directory corresponds to the first frame of the main storage, the second entry to the second frame, and so on.

One field of the page directory entry contains the page address of the virtual address located in the corresponding frame of the main storage. When this field matches the page address of the virtual address to be translated, the index of that page directory entry becomes the FID for that virtual page. Thus the FID translated from the page address, along with the BID from the virtual address together form the main storage address.

Specific bits from the virtual address are combined or hashed by the hash generator to select an entry from the hash index table. The selected hash index table entry contains an index into the page directory. A part of the page directory is reserved as a pointer or index to indicate where additional entries with the same hash are to be found, if there are any. Thus, all of the entries with the same hash value are found on a linked list (or chain) in the page directory. The last entry on each chain is distinguished from the others by an end-of-chain indicator.

During the translation of the virtual address, the page directory searching mechanism need only find the chain that contains the virtual page and search only those entries on that chain, looking for a match of the virtual page address. If a match is found, then the index of that page directory entry is the FID for that virtual page. If an end-of-chain bit is encountered before a match is found, a page fault is signaled to the page fault handling routine. This routine can then resolve that page fault by bringing the page corresponding to that virtual address from secondary storage to main storage and updating the page directory.

If, at any given point in time, several virtual addresses were to hash to the same hash value, long page chain lengths would result and the performance of the machine would be degraded. It is therefore advantageous to have many short page chains. This is accomplished by making the number of entries in the hash index table larger than the number of entries in the page directory and providing a hash generator that produces a uniform distribution of hash index

Reprinted from Electronics, March 15, 1979; Copyright © McGraw-Hill, Inc., 1979.

table entries. It can be shown [2] that, with this uniform distribution, the average number of page directory entries probed, N, is dependent on the ratio of the hash index table size to the page directory size, R, or

$$N = 1 + \frac{1}{2R} \qquad (1)$$

Thus, if the hash index table is twice the size of the page directory, the average number of probes is 1.25 entries.

If the hash generator does not provide a uniform distribution of hash index table entries, Eq. (1) does not hold and the average number of entries probed would increase since some entries of the hash index table would be favored over others. Therefore, to minimize the average number of probes, the hash generator must provide a uniform distribution of hash index table entries. The actual hashing algorithm required to provide the uniform distribution depends on how addresses are assigned.

Assignment of virtual addresses

Data structures or "objects" [3], as they are called in this article, are created, destroyed, grow in size, or shrink in size during the life of a computer system. In order to facilitate the handling of these objects, the virtual address space is divided into independent address spaces called "segments." Each segment consists of a linear sequence of addresses, from a starting virtual address to some maximum. One object may be contained in a segment and then is allowed to grow to the maximum size of the segment. Only the portion of the segment that contains data physically exists in main storage or secondary storage. Since the segment is generally larger than the object it contains, some of the virtual pages associated with the segment are not used. This leads to a sparse usage of the virtual address space. The portion of the virtual address that uniquely identifies the segment is called the segment identifier (SID) and that portion of the address that identifies the page within the segment is called the page identifier (PID).

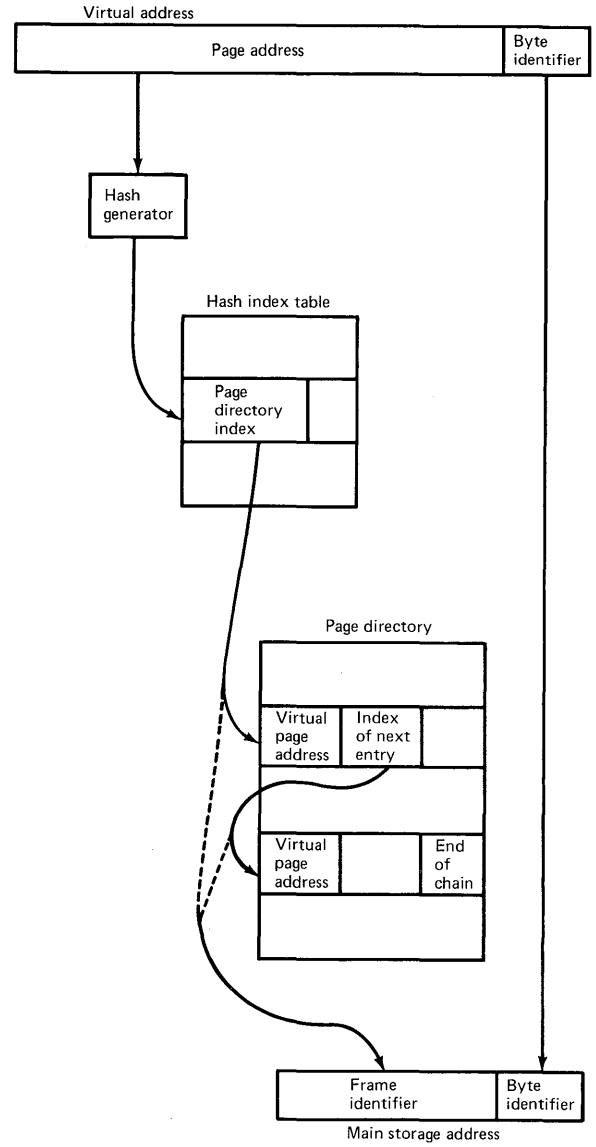


Figure 1 Virtual address translation

Consider a computer system with a fixed amount of main storage and secondary storage. There is a relationship between the number of objects and the average size of the objects in that system. The system can be characterized as having a large number of small objects, a small number of large objects, or somewhere in between. A system with a large number of small objects has a large SID and a small PID. Conversely, a system with a small number of large objects has a small SID and a large PID.

In System/38, two segment sizes are allowed, a small segment of approximately 65,000 bytes and a large segment of approximately 16 million bytes. Depending on its potential size, an object can be assigned either to a small segment or a large segment, leaving a portion of the segment vacant. Thus, some mechanism is needed to transform the nonuniform distribution of virtual addresses to a uniform distribution of hash index table entries. This transformation is performed by the hash generator.

Hashing algorithm

The hashing algorithm must transform the sparse usage of the virtual address space to a uniform distribution of hash index table entries. It is also desirable that consecutive virtual pages, small segments, or large segments cannot hash to the same hash index table location since there is a relatively high probability of consecutive pages or segments being referenced.

Thus, the hash generator of Figure 2 is used to meet these requirements. The hash is developed by taking the exclusive-or of the reverse order of the PID bits, with the low-order bits of both the small SID and the large SID.

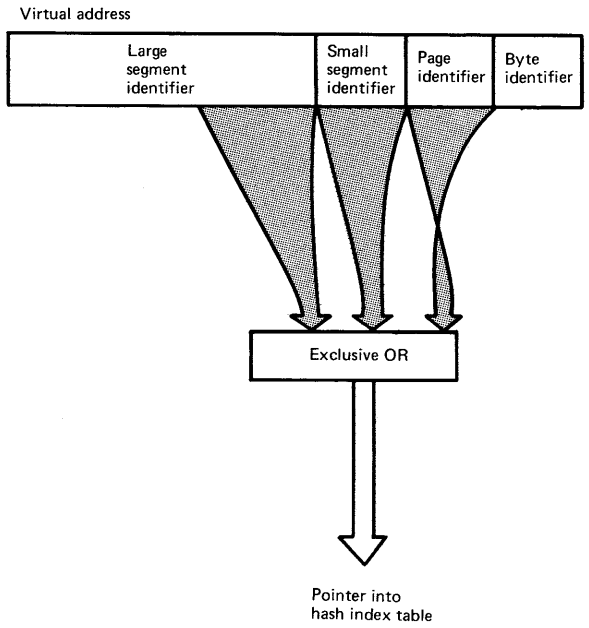


Figure 2 Hash generator

The effect of this hashing algorithm when the system is using a large number of small objects is that more bits from the SIDs and fewer bits from the PID are effective in generating the hash. On the other extreme, in the system using a small number of large objects, fewer bits from the SIDs and more bits from the PID are effective in generating the hash. Thus, the hash generator compensates for variations in the number or the size of the objects contained in the system.

Since virtual address bits are taken from both the small segment identifier and the large segment identifier, the ratio of the number of large segments to small segments is not important to the effectiveness of the hashing algorithm.

Conclusion

The method has been described for translating a large virtual address to a comparatively small main storage address on System/38. A hash generator is used to provide a uniform distribution of hash index table entries which in turn minimizes the average number of probes into the page directory resulting in fewer main storage accesses during translation. Since the FID is derived from the index of the page directory, no FID field is required. The page directory is easily updated without moving entries, by just changing the chain or chains associated with the virtual addresses added or removed. This page directory design lends itself to reverse translation since a frame identifier can be used to directly index the page directory entry containing its virtual address.

References

1. R.E. French, R.W. Collins, and L.W. Loen, "System/38 machine storage management," page 63.
2. R. Morris, "Scatter storage techniques," *Communications of the ACM,* 38-43, (January 1968).
3. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.

Provides an overview of the System/38 I/O structure by describing the operation of the I/O channel and the methods used to attach devices to the system.

System/38 I/O structure

D.O. Lewis, J.W. Reed, and T.S. Robinson

Design objectives

The I/O structure for IBM System/38 was designed to achieve three major objectives. The first was to develop a channel architecture which allows model implementation tradeoffs, exploits current LSI technology, utilizes the system's virtual addressing capabilities, and allows multiprogramming at the channel program level. The second objective was to decouple the processing unit from the channel by means of a queued asynchronous structure which allows channel program stacking with minimum impact on the processing unit. The third objective was to provide multiple I/O product attachment interfaces for flexibility of added features and to accommodate user migration.

User views of input/output

There are two views of input/output apparent to the System/38 user. The first is at the data management level. This level provides device and data independence. Input/output managers (IOM) support that data management level by translating data management I/O requests into channel programs. The second view is at the physical attachment level, that is, the external interface. This physical level provides a number of unique machine (UMI) and multimachine (MMI) interfaces. These two user views of input/output are combined by means of an internal structure, as shown in Figure 1. This structure consists of:

- A queued asynchronous system channel boundary.

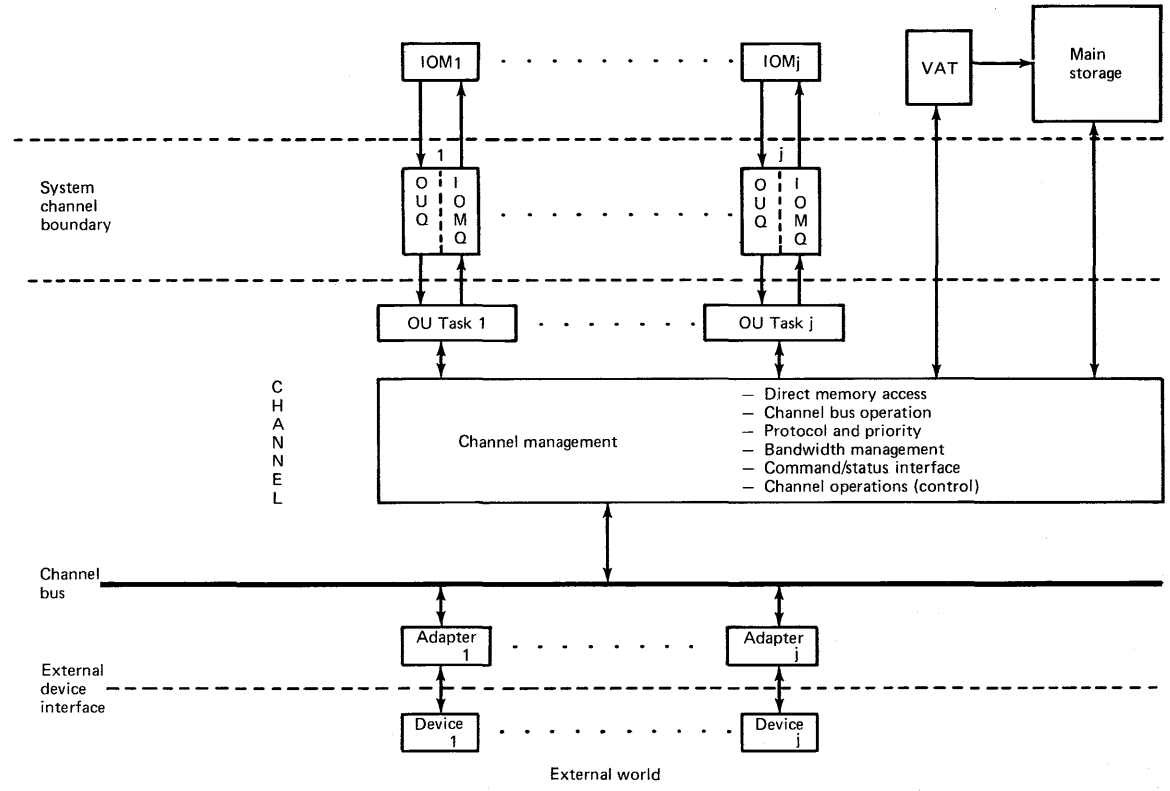


Figure 1 System/38 I/O structure

• A channel processor which executes channel commands or channel programs (multiple commands), allows direct memory access, multiplexed I/O, and supports intelligent I/O adapters (IOA) via a common channel bus.

• Internal IOAs which give to system designers the capability of distributing function from IOM components to an IOA.

The operational unit task

An important attribute of the System/38's I/O channel is the concept of an operational unit (OU) task. An OU task performs all of the functions of what is commonly termed a "subchannel" in many channel structures; that is, it contains all of the information necessary to sustain an I/O operation with its associated I/O unit. In addition, the OU task is the channel component which executes IOM-formed channel commands and is capable of competing for system resources as a system task. Communication between the IOM and its associated OU task is accomplished by the sending and receiving of messages to the operational unit queue (OUQ) and the input output manager queue (IOMQ). These messages either carry an IOM-formed channel command or point to an IOM-formed channel program for OU-task execution. They also carry an OU task-formed response, to the IOM-requested work. There are five channel command types generically referred to in System/38 as operation blocks (OB). The five OB types give the IOM programmer the capability of writing sophisticated channel programs. The OBs are 16-byte fields and contain the information necessary to initiate, sustain, and terminate an I/O operation. The data address contained in the OB is a 6-byte virtual address, hence, System/38 I/O participates fully in the system's virtual addressing structure. I/O unit addressability is accomplished with an operational unit number, a one-byte descriptor which is unique to an OU task, and its associated I/O unit.

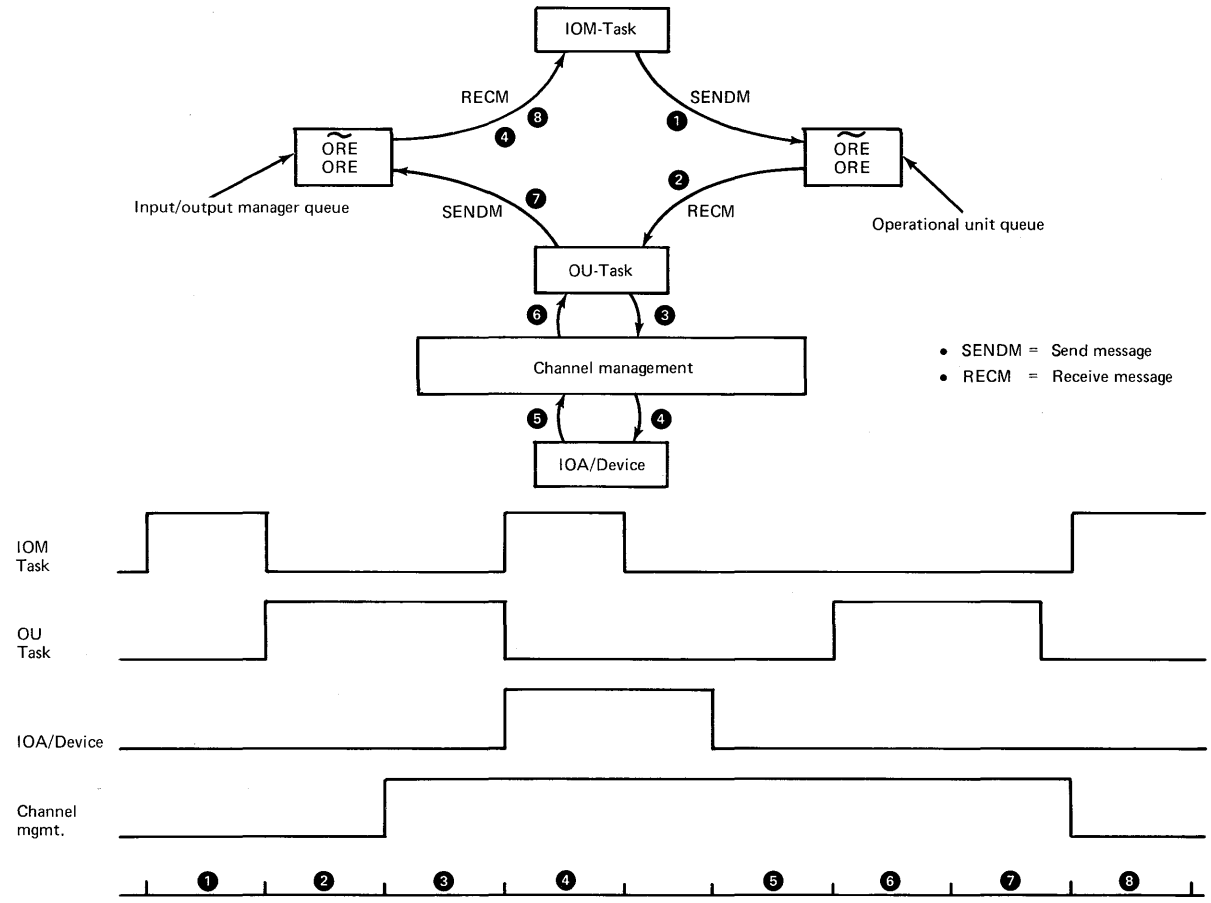


Figure 2 Channel operation

Channel operation

An overview of channel operation is shown in Figure 2. An IOM task requests an asynchronous I/O operation by sending a send/receive message (SRM) containing a channel program to the OUQ. The OUQ task responsible for servicing the queue receives the SRM and initiates the I/O operation by executing the first OB in the channel program. A channel program may contain one or more OBs. The distinction is important because the OU task participates in the execution of each OB. The IOM task, however, only participates in the forming and transmission of a channel program, receiving ending status from the OU task at the completion of the channel program and not for each OB executed within the channel program. Processing and I/O overlap is, therefore, greatly enhanced with the effective utilization of channel programming by the IOM.

Each OB (channel command) results in the OU task sequencing through three distinct phases:

1. Receiving, decoding, and execution of the OB. In effect, the OU task initializes channel management such that subchannel operation and the IOA/device may be initiated and sustained.
2. The OU task quiesces to the dispatchable-wait state pending completion of the device command by the IOA/device. During this period, channel management selects, transmits the device command, monitors, and services the IOA/device on a multiplexed prioritized basis. When a command-ending status is presented to channel management by the IOA/device, a channel processing function called the I/O event handler services and presents this ending status to the OU task which goes to the dispatchable-ready state.
3. Upon being dispatched again by the system, the OU task must present the IOA/device ending status as OU status to the IOM via a message sent to the IOMQ if the completed operation block (command) is the final or only operation block in the IOM-issued channel program. If there are additional OBs in the channel program, the OU task will return to Phase 1 and process the next OB.

Channel management communicates with attached IOAs over a common channel bus, as shown in Figure 3. The channel, in selecting and servicing IOAs, utilizes three distinct sequences:

START A particular IOA is selected and informed of a pending command in channel.

POLL A particular IOA is informed on a priority basis that a channel service grant is available.

GRANT The polled IOA is granted channel service.

The particular sequence is reflected by the aggregate state of the ten TAG lines, six channel-activated and four IOA-activated. During the POLL-GRANT sequences, the channel operates in a block-multiplexed mode over a bidirectional bus with the length of the block transfer determined by the IOA.

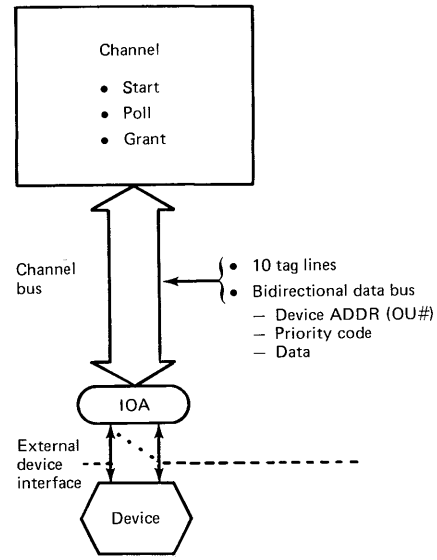


Figure 3 Channel dialog with device

Input/output adapters

The intelligent IOAs are either hardwired [1] for high-speed devices or microprogrammed controllers [2] for low- and medium-speed devices. A given IOA may, depending on speed and function requirements, service multiple device attachments. This is reflected in the many unique machine and multimachine interfaces to external devices. Examples of multimachine interfaces seen on System/38 are SDLC remote links and the SDLC local loop.

Summary

The I/O structure of System/38 has been developed to meet demands now posed by the concept of the attached work station, the distribution of function to intelligent adapters, and the ever-increasing use of networking. This, coupled with device multiattachment methods, will permit the user of low-end computers to utilize the functional capabilities normally associated with much larger systems.

References

1. J. W. Froemke, N. N. Heise, and J. J. Pertzborn, "System/38 magnetic media controller," page 41.
2. E.F. Dumstorff, "Application of a microprocessor for I/O control," page 28.

Application of a microprocessor for I/O control

E. F. Dumstorff

Characterizes the microprocessor that was designed specifically for the attachment of devices to the System/38.

Microprocessors are significantly influencing the design of system structures, particularly in input/output control. The advent of LSI has made the extensive use of microprocessors in I/O subsystems economically feasible. I/O subsystems using microprocessors have three primary advantages over conventional hardware designs. First, more function from the CPU can be moved into the I/O subsystem, leaving more CPU power to drive more microprocessor-controlled devices or to simply improve processing unit performance with the same number of devices. Second, in the LSI environment, the microprocessor approach minimizes the number of engineering changes and unique part numbers. Once developed, the microprocessor and its role in I/O subsystems became the design standard. The shallowest possible device-unique adapters are then developed, making the best possible use of the microprocessor for each device attached to the system. The adapter design process in general becomes more structured and easier to control. Third, the microprocessor approach is more flexible. As development progresses, it is often desirable to move function to or from the processing unit or hardware portion of the adapter, or simply to redefine a function already designed. In the LSI environment these types of changes would be disastrous to a conventional hardware design. However, with the microprocessor approach, these changes are more easily handled with microcode changes. Many times no hardware changes are required. For these reasons, a microprocessor-based I/O subsystem was chosen for the IBM System/38.

The I/O structure used in the System/38 [1] makes multiple use of a microprocessor as an I/O controller. General characteristics of the I/O controller (IOC) and its connection to the system control adapter (SCA), system channel, and device adapter are presented in this article.

I/O subsystem structure

The I/O subsystem structure used in the System/38 is shown by Hoffman and Soltis [2], who indicate the multiple use of the IOC in the I/O structure.

The IOC is connected directly to the system channel through the channel adapter. It is packaged on a logic card which contains both the channel adapter and the IOC. Logic associated with the channel adapter accounts for one-third of the hardware on the logic card which is approximately 7 x 9 inches in size. This logic card is a common field replaceable unit (FRU) used by device adapters to perform device control functions and to connect the device adapter to the system channel. All I/O, with the exception of some magnetic media devices, connect to the system via the IOC. Read only storage (ROS) control store that personalizes the IOC function to a specific device is packaged with the adapter unique to that device. Operation of each IOC is initiated by the SCA. At power-on time, all IOCs, with the exception of the IOC used by the SCA, are in a stopped, reset state. The SCA can then start the IOCs one at a time via SCA control as the system is brought up.

Controller characteristics

The IOC is an 8-bit processor with parity checking. It includes an internal 512 x 9 data store (DS). An additional 512 x 9 array is optional. Thirty-two local store registers (LSRs) are implemented as the first 32 DS locations in the first 512 x 9 DS array. The IOC has two program levels, one interrupt and one background. I/O instructions passing data to or from the device adapter can be extended in increments of one IOC clock cycle (670 ns) by the adapter. The I/O extend function is used to extend I/O instructions when more time is required in the adapter to respond to the data on the I/O interface. This makes the IOC easy to connect to adapters implemented in slower logic technologies. All instructions, with the exception of extended I/O and BRN (Branch Register Indirect) instructions, execute in one controller clock cycle. Thirty instructions are implemented. The IOC generates a 13-bit control

store address (CSA) with parity added, making the complete address 14 bits.

Data flow

A data flow diagram of the logic contained on the System/38 I/O controller card is shown in Figure 1. It is divided into three areas: SCA control decode, the IOC, and the channel adapter.

The IOC is a two-address machine. During one instruction execution (670 ns), it loads two operands into the ALU operand registers, combines them, and stores the result. In parallel with the execution of an instruction, the next instruction to be executed is being accessed from control store and the control store address is incremented via a hardware incrementer.

The IOC data flow is designed to execute four basic types of instructions. These are LSR to LSR, LSR to DS and DS to LSR, KI (control immediate), and I/O. KI instructions are used to pass data between control space (internal control registers, such as CSAR save registers for both program levels, check/status register, data store page register) and LSR space. I/O instructions pass data between LSR space and the device adapter.

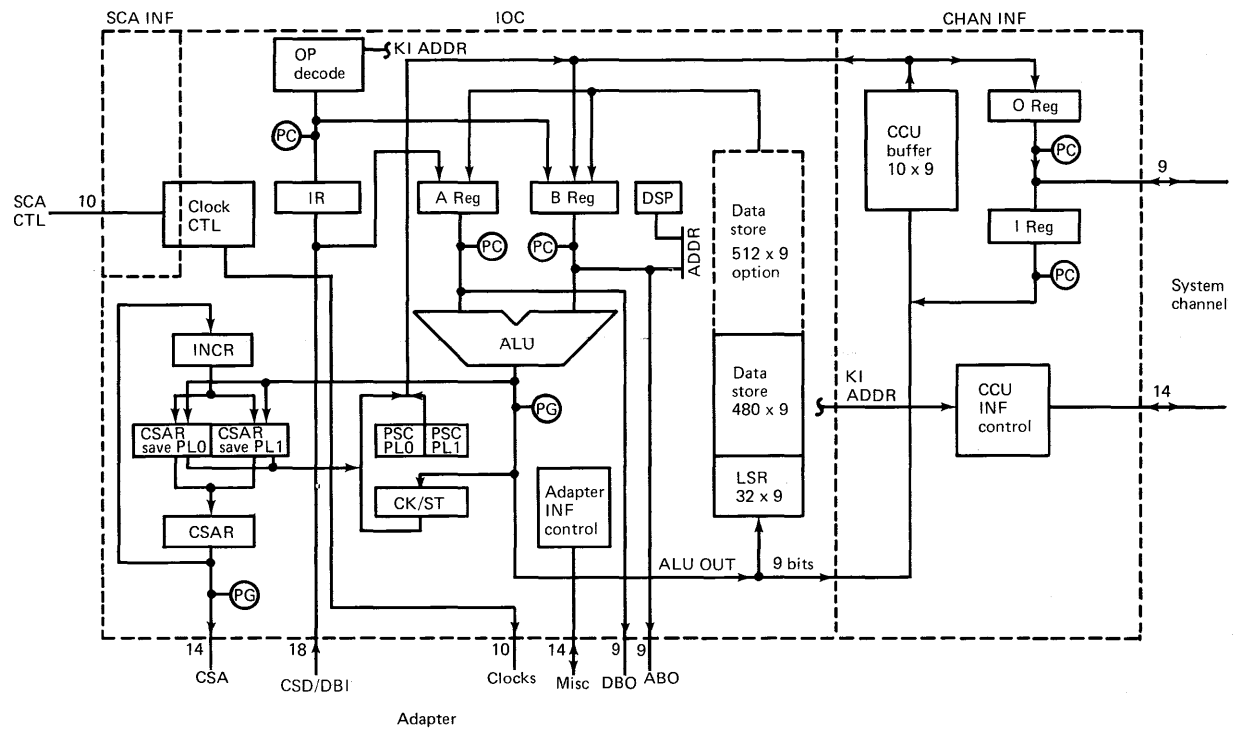


Figure 1 A data flow diagram of logic on the System/38 I/O controller card

The IOC is connected to three other logic areas in the system. These are the SCA, the system channel, and the device adapter areas. The SCA area generates ten control lines which are used to control various internal parts of the system. In the I/O subsystem, these control lines are used to start, stop, reset, and perform diagnostic functions on any IOC in the system concurrent with other IOCs operating the devices associated with them.

The channel adapter generates a connection one byte wide to the system channel. This connection, including control, consists of 27 lines. Data can be passed to or from the channel via the channel adapter at 480K bytes/sec. The channel adapter is controlled by the IOC via KI instructions. It is effectively a native adapter attached to and packaged with the IOC. This is important in that it makes the controller easy to adapt to other system environments.

The device adapter interface consists of 72 lines. It includes the control store interface and I/O interface for transferring data to and from the adapter. For I/O write operations, ABO (9-bit address bus out) and DBO (9-bit data bus out) are sourced directly from the arithmetic-logic unit (ALU) operand registers. This permits an I/O write operation, where address and data originate in LSR space, in one instruction cycle (670 ns). When operating with adapters requiring more time to respond to this interface, I/O instructions can be extended in increments of 670 ns. This is done by the adapter conditioning the "I/O extend" line on the adapter interface at the beginning of the I/O instruction and keeping it conditioned until the required number of instruction cycles (670 ns each) have occurred.

Instruction timing

Figure 2 shows the timing associated with an IOC clock cycle. Each clock cycle is divided into ten clocks, each 67 ns in length. The resulting ten clock signals are made available to the adapter for use as desired.

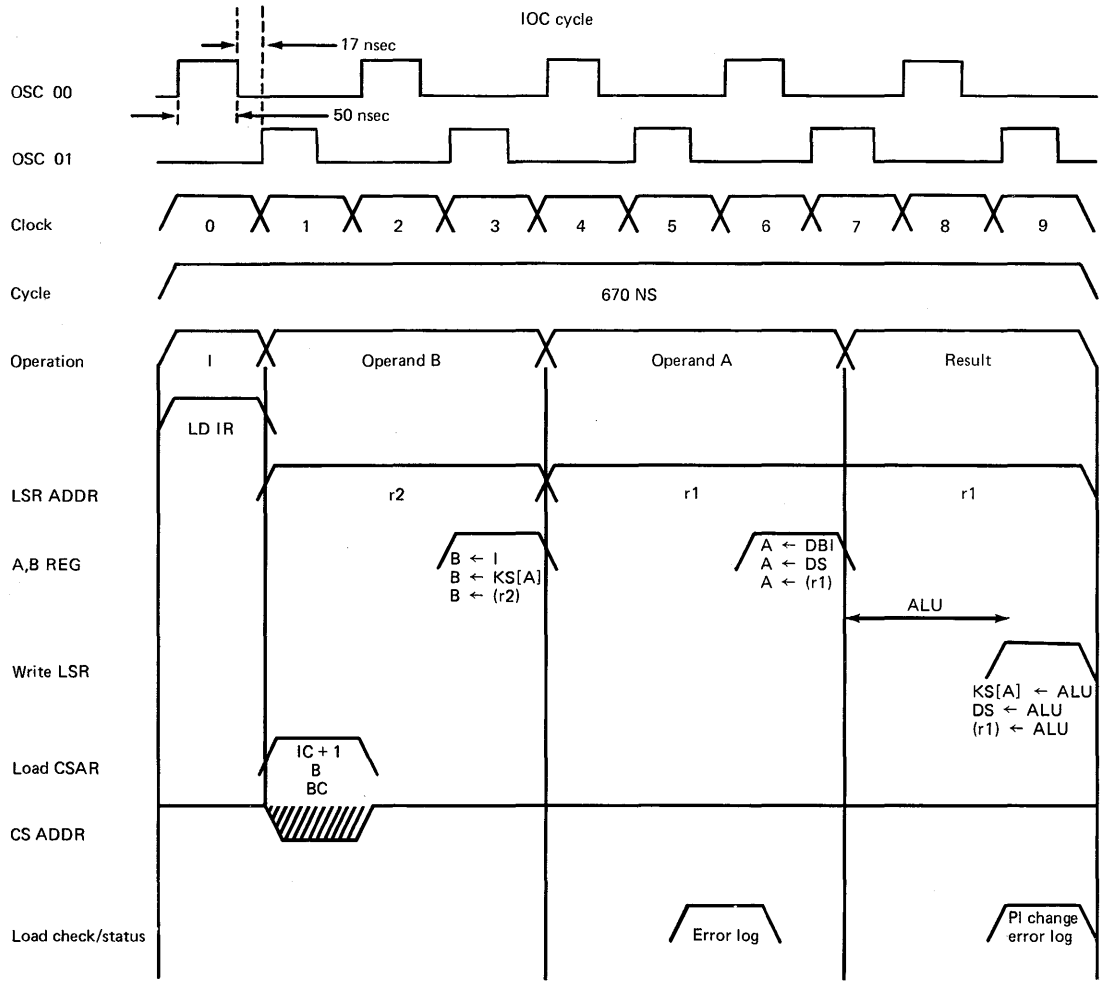


Figure 2 Operand register/destination sources

Four basic operations are performed during each IOC cycle. First, the instruction register is loaded during I time. Second, operand B is loaded from the source specified by the current instruction. Third, operand A is loaded similar to operand B. Fourth, the result is generated and stored in the memory space specified by the instruction.

The possible sources for the operand registers and the destinations for the result are shown in Figure 2. This information, along with the instruction description shown in Figure 3, can be used to better understand the IOC data flow for the various instructions.

Controller instruction set

A list of the IOC instructions implemented is shown in Figure 3. Each instruction contains 17 data bits plus one parity bit. The function performed by each instruction is described in Figure 3.

Conclusion

I/O control functions in the system environment are well suited for application of microprocessors. The structure chosen for the IBM System/38 I/O subsystem depends heavily on the use of the microprocessor described in this paper. This approach was chosen based on three primary advantages microprocessor-based I/O structures offer over conventional hardware designs.

In general, more device control function was moved from the processor to the I/O subsystem. This left more processor cycles to improve the overall system performance. The number of engineering changes and unique part numbers required in the I/O subsystem were reduced by approximately 30% from a conventional design approach. As the system was developed, a considerable number of functional changes were included in the various devices late in the development process by making ROS control store changes. Without the flexibility the microprocessor approach offers, many of these changes would not have been possible.

As technology continues to advance, microprocessors will continue to be used extensively in the system environment.

References

1. D.O. Lewis, J.W. Reed, and T.S. Robinson, "System/38 I/O structure," page 25.
2. R.L. Hoffman and F.G. Soltis, "Hardware organization of the System/38," page 19.

Instr	Format	PSC	Description
	1111111 01234567890123456P		
KIR	0000 r1 0 A P	L	(r1) ← KS[A]
2 KIW (BRN)	0000 1	-	1 KS[A] ← (r1)
NRI	0001 P	L	(r1) ← (r1) . I
ARI	0010	A	(r1) ← (r1) + I
XRI	0011 I	L	(r1) ← (r1) V I
LRI	0100	-	(r1) ← I
CRI	0101	A	PSC ← (r1) + I + 1
ORI	0110	L	(r1) ← (r1) V I
TBNI	0111	L	PSC ← (r1) V I
XR	1000 r2 001 P	L	(r1) ← (r1) V (r2)
AYR	1000 010	C	(r1) ← (r1) + (r2) + C
AR	1000 011	A	(r1) ← (r1) + (r2)
SYR	1000 100	C	(r1) ← (r1) + $\overline{(r2)}$ + C
SR	1000 101	A	(r1) ← (r1) + $\overline{(r2)}$ + 1
NR	1000 110	L	(r1) ← (r1) . (r2)
OR	1000 111	L	(r1) ← (r1) V (r2)
RR	1001 000 P	R	(r1) ← (r2) ROT RT 1
LR	1001 001	R	(r1) ← (r2)
CR	1001 011	A	PSC ← (r1) + $\overline{(r2)}$ + 1
LN	1010 000 P	-	(r1) ← DS[DSP:(r2)]
STN	1010 001	-	DS[DSP:(r2)] ← (r1)
IOR	1010 010	L	(r1) ← IOS[(r2)]
IOW	1010 011	-	IOS[(r2)] ← (r1)
IORI	1010 110	L	(r1) ← IOS[I:110]
IOWI	1010 111	-	IOS[I:111] ← (r1)
LND	1011 D r DL P	-	(r1) ← DS[(r2):D]
STND	1100 H 2	-	DS[(r2):D] ← (r1)
B	1101 B P' P	-	IC ← P':B
BC	1110 M	-	IC ← IC HI:B
DBC	1111 M	-	IC ← IC HI:B IF DIAG M

1 KS refers to control space (i.e., internal control registers)

2 A KIW instruction to a particular address is decoded as a BRN (Branch Register Indirect) which causes the controller to enter a 2-cycle sequence resulting in the control store address to be sourced from the register address by (r1).

3

A MASK OF → TESTS FOR CONDITION CODE →	100 CC0	010 CC1	001 CC2	000 CC3
Arithmetic result	0	-	+	Carry
Compare result	(r1) = (r2)	(r1) (r2)	(r1) (r2)	Carry
Logical result	All 0s	All 1s	Mixed	No branch
Rotate result	All 0s	High order 1	Mixed & (+)	Low order

Figure 3 Micro instruction format

Microprocessor-based communications subsystem

F.X. Roellinger, Jr. and D.J. Horn

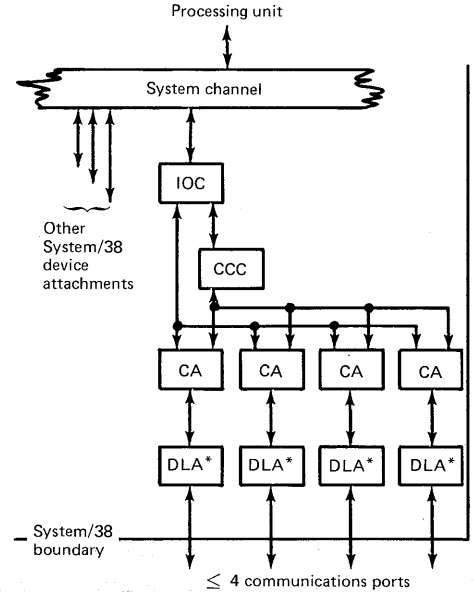
Discusses the building block approach used to implement the communications subsystem on the System/38.

This article highlights the main features of the communications subsystem on the IBM System/38. This subsystem employs a microprocessor to multiplex up to four synchronous data link control (SDLC) teleprocessing lines through one port on the system channel. The microprocessor operates under control of the processing unit, which presents various sequences of commands (start-up, read, write, etc.) via the system channel to perform data exchanges on one or more TP lines. The operation of the system channel is described by Lewis, et al [1].

Organization of the communications subsystem

The System/38 communications subsystem employs a building-block approach to offer diverse and flexible line-type attachment possibilities. Four building-block types are used as shown in the subsystem diagram (Figure 1).

The I/O controller (IOC) is a vertical type microprocessor utilizing 17-bit-plus-parity control words, 8-bit-plus-parity data paths and a 670 ns instruction cycle time. Further details of the IOC structure and operation are discussed by Dumstorff [2].



1. 1200 bps integrated modem
   - Nonswitched
   - Switched with auto answer
   - Switched with manual answer
   - Nonswitched, switched back-up with auto answer
2. 2400 bps and 4800 bps integrated modems
   - Nonswitched
   - Switched with auto answer
3. External modem adapter (switched or nonswitched use)
   - For use at data rates ⩽ 9600 bps
   - EIA RS232C/V.24 interface
4. External autocall unit adapter (must be used in combination with an adjacent external modem adapter)
5. Digital service adapter—connects to a channel service unit (CSU) of the nonswitched AT&T Data-Phone (registered trademark of American Telephone and Telegraph Company) digital data service network at data rates of 2.4 kb/sec., 4.8 kb/sec., and 9.6 kb/sec.
6. Loop system control adapter—available on an RPQ basis at 9.6 kb/sec. and 19.2 kb/sec. data rates.

***DLA types available**

Figure 1 Communications subsystem hardware structure

A communication adapter (CA) is used for each port installed. The CA contains hardware function which is personalized and controlled by the IOC microcode to support the specific port application.

The communication common control (CCC) provides function which is common to all CA line appearances (thus reducing hardware duplication cost) and contains the IOC ROS control store.

The data link adapters (DLA) provide capability for several forms of communication attachments (Figure 1). Each of the different DLA designs provides for an identical physical connection to the CA. As a result, it is possible to use the single CA design as part of each of the various communications attachments.

The partitioning of function within the communications subsystem is shown in Table 1.

Features of the subsystem

The subsystem provides for half-duplex SDLC operation over a maximum of four communications ports. Attachment flexibility is achieved by permitting installation of any combination of four DLAs which

Table 1 Communications subsystem function

Communications adapter (CA) hardware

- Driving and sensing registers for DLA control
- Serialization/deserialization of external SDLC data stream
- SDLC flag detection, abort and idle detection
- SDLC zero bit insert/delete
- NRZI encoding/decoding of data stream (when enabled)
- Internal clock correction for asynchronous modem applications
- Diagnostic capabilities for communications subsystem verification
- Parity checking and generation on subsystem address and data paths

Communication common control (CCC) hardware

- SDLC ROS control store, accessing control, and address parity checking
- CA line selection/multiplexing control
- Control of functions requested by the system control adapter (SCA)
- Internal clock source
- Subsystem reset and register clocking control
- TP line analysis diagnostic support

IOC function as implemented in microcode

- Insertion and deletion of SDLC flag characters
- Frame check sequence (FCS) generation and checking
- Data buffering between system channel and CA
- Operation of timer counters for SDLC timeouts (e.g., idle state detect)
- Automatic polling of remote multipoint tributary stations
- Handling of errors detected by system channel
- Switched line connection via the DLA
- Presentation to SCA of DLA status
- I/O command fetch and interpretation
- Synchronization of read/write command sequence
- Diagnostic checkout of CA functions
- Handling of parity checks detected by IOC, CCC, or CA

Data link adapter (DLA) hardware (possible types of function)

- Modulation/demodulation
- Voltage level (electrical characteristic) conversion
- External network connection control circuitry

is consistent with the subsystem maximum aggregate data rate of 57.6 Kb/sec.

DLA choices provide the capability to operate point-to-point on analog switched facilities, and point-to-point or multipoint on analog or digital nonswitched facilities. Switched connection options include manual or automatic calling, and manual or automatic answering. Automatic switched network disconnect is provided under microcode control. Both primary and secondary station operation are supported. In addition, primary station control of a local loop system is available on an RPQ (request for price quotation) basis.

The microcode provides an automatic polling function for up to eight multipoint tributary stations (per communications port). This function removes from the processing unit the burden of regularly polling inactive terminals.

Data integrity within the subsystem is ensured by both hardware and microcode function. Hardware parity checking is provided on all address and data paths between the IOC, CCC, and CA building blocks; external SDLC data stream checking is provided through microcode frame check sequence (FCS) generation and checking. Error recovery or reporting is under microcode control.

Microcode organization

The subsystem includes 1024 bytes of data storage, accessible by the IOC. The first 32 bytes of data storage are used by the IOC as local store registers (LSRs). The LSRs are used primarily as temporary work areas, but several are reserved for flags and status indicators to effect communication between the two program levels.

Data storage is divided into five general areas: (1) data buffers for storage of data between the system channel and the communications lines; (2) tables indicating which lines are installed and their relative service priorities; (3) queuing areas of channel

service requests; (4) common work areas, which are extensions of the LSRs; (5) line parameter tables, each containing status and personalization data for one communications line.

The microcode as organized is reusable, so that one copy may service any of the communications lines. Before a particular service routine is invoked, a line-selecting routine is used which sets a base register according to the line selected for service. This base register then becomes a line-parameter-table selector for the line to be serviced. In addition to status and personalization data, the line parameter table also contains the data buffer pointers and information necessary for communication with the input/output managers in the processing unit. The line parameter table is the heart of the line multiplexing capability in the microcode.

Processor command execution

The same queuing structure used throughout the system is also used between the processor and the communications subsystem. When the input/output manager for communications wishes to activate a particular function in the subsystem, an I/O command is placed on a queue in main storage and the subsystem is notified via a hardware signal on the system channel. The microcode then fetches the command from main storage, performs the requested function, and issues a status reponse to the processing unit. The command/response queue is known as an Operation Unit (OU) queue.

Each communications line employs two OU queues, one for transmitting and the other for receiving. The communications subsystem does not have a full duplex implementation, but at times commands may be outstanding on both queues. A typical example is a read/write sequence in which the subsystem transmits an SDLC frame with the poll/final bit on in the control byte, allowing a response from the remote station. Explicit synchronization of the transmit/receive sequence is not necessary by the processing unit; upon receiving either command, the subsystem will wait for the other before transmitting. At the end of the frame, response status is transferred to the processor for the write command, line turnaround takes place, and the receive command is activated to receive the response from the remote station.

Interrupt structure

The microcode employs two program levels, an interrupt level (0) and a background level (1). A pseudo-interrupt level is available for hardware parity checks, which force a trap to error-handling code. After the startup, data store initialization, and line priority specification have been completed by the system control adapter (SCA) and CPU, the microcode enables program level 0 interrupts and remains in the program level 1 idle loop until an interrupt occurs. The following events will cause an interrupt: a 13.3 ms timer pulse on the CCC, a byte service request from a CA, or a device-address-ready signal from the system channel that an I/O command is outstanding.

Byte service requests are handled directly by program level 0 routines, while timer and device-address-ready interrupts are enqueued for handling by program level 1. Other program level 1 tasks are: fetching commands and issuing responses via the system channel, issuing data-buffer transfer requests to and from main storage via the system channel, providing signals for the DLA status display, and execution of the I/O commands.

As previously stated, program level 0/1 communications are effected through the LSRs. When program level 0 requires processing by program level 1, it merely sets a bit corresponding to the line being serviced in the appropriate LSR. Service requests include data transfers for both transmit and receive data, and logical-operation-end (end of an SDLC transmitted frame or an SDLC receive frame sequence). Program level 1 is the supervising level in that its execution of I/O commands causes the activation of program level 0 code, and because it may disable interrupts and deactivate the operation of the interrupt service routines.

Concurrency features

The subsystem allows a considerable amount of flexibility in line operation and diagnosis without interaction with other operating lines. Most hardware errors associated with one DLA or CA will be handled by the microcode or reported to the processing unit without affecting the operation of other lines. A failing line may be removed from the line service priority table, diagnosed concurrently with other line operations, and replaced in the service table; if desired, the service priority may be altered at any time. This is all in keeping with the concurrent maintenance philosophy of the entire system.

Diagnostic functions

Various components have been built into the system to facilitate problem determination by the customer engineer. These components issue various diagnostic I/O commands to the subsystem which write and sense the state of the DLA and exercise the CA to isolate a failing hardware component. One single I/O command is available which, as implemented in microcode, checks out all of the SDLC functions (except zero bit insert/delete) and many of the other functions performed by the CA.

Other hardware diagnostic capabilities include an internal trap which allows the customer engineer to

record and display both transmit and receive data, and to display the activity of the DLA dynamically on the system console for any installed line.

References

1. D.O. Lewis, J.W. Reed, and T.S. Robinson, "System/38 I/O structure," page 25.
2. E.F. Dumstorff, "Application of a microprocessor for I/O control," page 28.

Microprocessor-based work station controller

Presents the method used to directly attach multiple work stations to the System/38.

J.N. Tietjen and W.E. Hammer

The 5250 series terminals can be attached to the System/38 using a work station controller or by the remote communications adapter [1]. The work station controller provides a more responsive path from terminal to user programs. The communications line-related overhead is eliminated, thereby improving system response time to a terminal transaction.

The work station controller as shown in Figure 1 is a microprocessor-based control unit designed to allow the attachment of IBM 5250 series of terminals to the IBM System/38. The System/38 I/O controller, a dedicated, high-speed microprocessor having a 670 ns instruction time, is used in conjunction with optional data store cards to allow up to 20 keyboard/displays and/or printers to be controlled by a single work station controller. The microcode has been designed to support both keyboard/displays and printer terminals. Alternate language keyboards are supported by loading an appropriate translate table in the microprocessor data storage. The microprocessor design allows operation of the supported terminals to be compatible with remotely located terminals attached to the IBM 5251 Model 12.

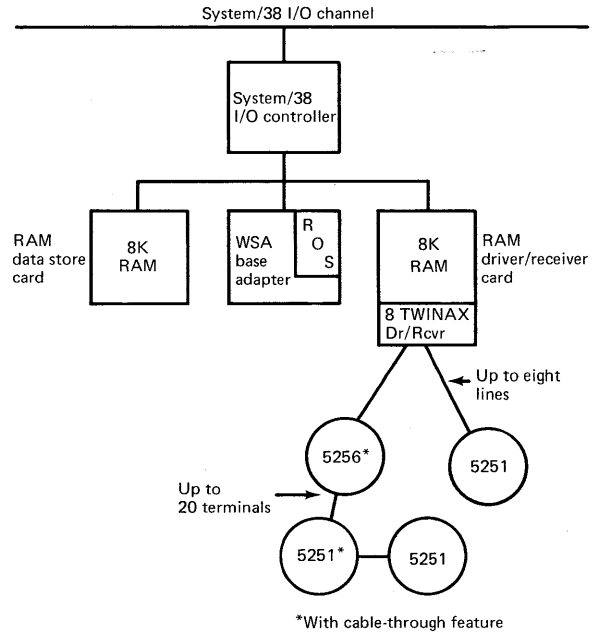


Figure 1 System/38 work station controller

Hardware overview

The hardware consists of the following elements:

- Microprocessor–System/38 I/O controller [2]
- Serial-to-parallel converter and control logic
- Twinax cable driver receivers
- Optional read/write data store
- ROS control store

Microprogram overview

The microcode for the work station controller is assigned to the two program levels within the microprocessor. Program level 0, the interrupt level, is assigned the major task of keystroke management. Interrupts are caused either by the expiration of a program-loadable timer or by the system channel bus logic, on the microprocessor card, indicating a processing unit command is available for processing (device address ready). Recognition of device-address-ready by the microprogram will cause a flag to be set to request program level 1 command servicing.

Program level 1 performs all control functions to the system channel. These tasks include controlling the transfer of data to/from main storage, interpreting the controller-defined processing unit commands, interpreting and executing the (user) data streams for display control, generating responses, and generating necessary status information.

As a result of interpreting the (user) data stream, a format table is built in the data store defining the location, length, and edit characteristics of all input fields (i.e., the field control words) on a display. Data keying is allowed only in screen locations defined within the format table; that is, before a keystroke is written to a display, the cursor location is checked to determine that the cursor is in a user-defined field.

Keystroke processing

The program-loadable timer will normally be set for an interval of approximately 32 ms. At the completion of this interval, each of the attached terminals is polled by the microcode for keystroke activity. A poll list, loaded at controller initialization time, is indexed sequentially by the microcode and used to control the polling function. The terminal responds to a poll with a status byte and a scan code byte, which are processed by the microcode. When the microprogram determines that a new keystroke is being presented, the scan code is used to access a translate table, and the required function is performed. In the case of a data key, the display code is read from the translate table and sent to the display after all field edit checks are satisfied. Keys such as enter, which send data to the host, cause the microprogram to post the data transfer request to a first-in first-out (FIFO) stack for handling by the program level 1 microprogram. In general, all keystrokes or error conditions requiring data transfer to main storage are posted to FIFO stacks for handling by the program level 1 microprogram.

Data stream processing

The data stream received from the user program is processed for a single terminal at a time. The actual data stream is in the same format as the data stream used by a remotely attached terminal (IBM 5251 Model 12). The data stream contains commands and orders which tell the adapter how to write the screen and which define the input field edit characteristics. The terminal operator can enter data only into valid input fields whose characteristics are defined by the data stream. The following field-level edit functions are supported by the work station adapter:

- Alpha shift
- Alpha only
- Numeric shift
- Numeric only
- Signed numeric
- Bypass
- Dup enable
- Auto enter
- Field exit required
- Monocase
- Mandatory enter required
- Mandatory fill
- Right adjust zero fill
- Right adjust blank fill

As part of interpreting the data stream, a table of input field characteristics, called a format table, is either built or modified. The format table has the starting address of the input field on the screen, the ending address, and the field edit characteristics. With these field format entries, edit checking and control are moved outboard of the user program. This provides immediate feedback to the terminal operator. With edit checking performed by the station controller, the user application program is guaranteed that all data received meets the requirements specified by the program. For example, only numeric characters will be accepted in a numeric-only field, or only alpha characters will be accepted in an alpha-only field. The program does not have to check for data not meeting this criterion.

For the 5256 printers, the commands and orders are contained in the data stream and are interpreted by the printer. The commands and orders perform various printer control functions, such as formatting the data and printing on a new line or new page, for example. Blocks of data up to 256 bytes long are transferred to the printer. The printer has two 256-byte buffers, and the number of buffers in the work station controller is specified by the Control Program Facility [3] along with the pacing count as described next. When the printer has a buffer available, the controller sends a block of data to the printer. The controller checks its inventory of available buffers and, if the inventory is equal to or greater than the pacing count, it requests that more data blocks be sent by the CPF. With this technique, the printer should be kept busy and all other resources can be shared.

Concluding remarks

The work station controller provides for high-performance attachment of display terminals and printers. The microprocessor attachment provides field-level processing, that gives immediate feedback to the terminal operator and permits program independence between terminals attached to the work station controller and terminals attached to communications lines.

References

1. F.X. Roellinger, Jr. and D.J. Horn, "Microprocessor-based communications subsystem," page 32.
2. E.F. Dumstorff, "Application of a microprocessor for I/O control," page 28.
3. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.

Microprocessor control of impact line printers for printing character-string data

Describes the System/38 printer subsystem design and presents the characteristics of the printer attachment.

D.T. Brunsvold

Microprocessors are used to control the impact line printers on the IBM System/38. The design of the controller provides the flexibility to print character-string data redirected from terminal devices as well as line-formatted data intended for line printers. This differs from conventional approaches in that system output can be directed to alternate output devices with a minimum of reformatting, resulting in a saving of system resources and a reduction in processing unit contention.

Printer subsystem features

The printer controller for the System/38 has been designed to control printers with speeds of 300 and 650 lines per minute. The microprocessor used in the printer controller is the System/38 I/O controller (IOC), which is described by Dumstorff [1]. One controller is needed for each printer attached to the system, but when a printer with the lower speed is upgraded to one with the higher speed, the controller need not be changed. The system identifies the printer attributes to the controller, which then makes appropriate control decisions.

The controller accepts data that has been blocked either by line or by form. Printing using line-by-line blocking requires one command from the system for each line printed. Printing data that has been blocked by form requires only one command from the system to print an entire form. In this mode of printing, character-string data intended for a terminal device can be redirected to a line printer and result in a similar visual display, subject to the restriction that some terminal control functions embedded in the character string are not meaningful on a line printer and must be ignored or have a similar control function substituted for them.

Carriage and forms control, including the detection of end of forms and the prevention of printing beyond the end of the last form, are achieved in the IOC microprogram using information supplied by the system when it issues initializing commands to the printer controller. Printing either 6 or 8 lines per inch is achieved in the same way, giving the system operator program control of forms length and the selection of 6/8 LPI.

Additional flexibility is achieved with interchangeable print belts. The system operator has the option of mounting print belts containing specialized or extended sets of graphics. The operator then issues system commands containing image information about the particular belt to the printer controller which controls the printing using the most recent belt information supplied.

Printer subsystem configuration

The printer subsystem consists of three parts—the printing unit, the hardware portion of the controller, and the microcode portion of the controller including the microprocessor (IOC). The functions performed in the various parts of the subsystem are listed in Table 1.

Data store description and organization

The controller hardware includes 1024 bytes of random access memory data store which can be accessed by the microprocessor. It is divided into four 256-byte pages. The first page of the printer controller data store contains local storage registers, control flags and pointers, status information, and space reserved for logging critical parameters and information about failing hammers, in the event of an error in the subsystem.

The remaining three pages contain (1) the current print belt information, (2) data store page control and print hammer optioning control constants, and (3) buffers which hold the current print line data, print hammer addresses to be strobed to the printing unit, and failing hammer information that is obtained after an echo check or any hammer-on check.

Table 1 Printer subsystem functions

Printer hardware functions
Print hammer address decoding
Parity checking
Carriage emitter generation
Print subscan emitter generation
Print hammer echo pulse generation
Carriage, belt, and ribbon motor controls and drivers
Ribbon speed checking
Diagnostic functions
Print hammer and paper clamp drivers
Controller hardware functions
Address bus decoding
Data bus funneling and latching
Parity generation and checking
Run control and status latching
Fire tier generation
Print hammer echo checking
Print hammer address strobe generation
Any hammer-on detection
Carriage timing generation and checking
Controller microcode functions
Command and data transfer and decoding
Printer status monitoring
Printer switch and interlock recognition and indicator control
Print belt synchronization and idle control
Controller status generation and error recovery
System data channel error detection and recovery
Interrupt handling
Hardware and microcode timer control
Carriage control and checking
Paper clamp control
Failing hammer logging for echo check and any hammer-on errors
Unprintable character detection
Print hammer optioning and limiting
Character string data analysis
Execution of imbedded character string control functions

Controller operation

Interrupt structure

Microcode instructions are issued in the microprocessor in two different interrupt levels. Program level 0 is the interrupt level that performs time- and synchronization-dependent tasks; program level 1 is the background level that performs supervisory and housekeeping tasks. See Table 2 for a description of the tasks performed in each program level.

Command processing

Most system commands sent to the printer controller are decoded, verified, and immediately executed in program level 1. Upon completion of the command, or if an error occurs, response information containing the completion status of the command is sent to the system. Additional detailed status information and a log of essential registers and parameters will be placed in data store if command execution was other than successful, and this information can be retrieved with other system commands. System commands that require printing and/or carriage motion are decoded and verified immediately, but the actual printing or carriage motion is executed in synchronism with the printing unit upon receipt of the proper timing pulses. The response containing the completion status of the command is sent to the system after the printing portion and all but the final line of carriage motion of the command are complete. The next system command and its associated data will be fetched and processed during the final line of carriage motion. This overlap improves throughput to the extent that double buffering of print data does not require the additional buffering hardware.

Table 2 Tasks performed in program levels 0 and 1

Program level 0 tasks (interrupt level)
Determine interrupt causes
Control hardware and microcode timing facilities
Synchronize the print belt with the microcode
Control the paper clamp and carriage
Detect and log errors
Perform print hammer optioning and limiting
Detect unprintable characters
Program level 1 tasks (background level)
Control communications and information transfer between controller and host system
Initialize hardware latches, registers, and data store after power on reset sequencing
Decode, verify, and execute system commands
Detect, analyse, and execute imbedded character string control functions
Monitor printer status, switches, and interlocks
Generate controller status
Detect and recover from controller and system channel error conditions

When operating on character-string data, the controller receives one system command per printed form, rather than one per printed line. This results in a savings of system resources and a reduction in processing unit contention caused by task switching. Spooling applications are simplified because line-by-line reblocking is eliminated.

The reduction in processing contention is dependent upon system configuration and the particular application program, ranging as high as 10% or more. When printing in character-string mode, data fields containing embedded control functions are continuously fetched from the time the system command is received until a form is completed with the control function which causes advancement to line 1 of the next form. At this time, response information is sent to the system. Each byte in the data field is analyzed and determined to be one of the following: a graphic character, which will be placed into the print data buffer; a control function supported by the controller, which will be interpreted much like a

system command and executed; or a control function not supported by the controller, which will result in immediate termination of the current system command and the returning of error response status to the system. As with discrete system commands requiring printing or carriage motion, the character string control functions which require printing or carriage motion are executed synchronously with the printing unit; however, response is not returned to the system until the current form is completed.

Print hammer optioning

Optioning, that is, the selection of which print hammers are to be fired based on print belt positioning and the data to be printed, is under IOC microprogram control. Optioning is performed synchronously with the printing unit and occurs as a result of the controller receiving a print subscan emitter pulse (PSS emitter) from the printing unit. When this emitter pulse is detected, the microprocessor switches to program level 0 and a number of events occur. The print belt position is updated and new pointers are calculated for the print belt image in data store. The PSS emitter count is updated, causing new echo checking information to be sent to the hardware portion of the controller, and new pointers are calculated for the print data buffer. Also, as a result of these updates, a series of checks is performed to confirm that the print belt, the fire tiers, and the microcode are still synchronized. When this checking has been completed, the actual selection of print hammers to be fired takes place.

Error detecting and logging

Errors are detected in the printer subsystem by several methods: hardware detected errors which the IOC microcode discovers by periodic polling of interface lines and controller latches, hardware detected errors which cause the microprocessor to switch to program level 0, IOC microcode-initiated error timeouts which cause the microprocessor to switch to program level 0, and errors detected by the IOC microcode during its normal execution.

At the time of detection of an error condition, the IOC microcode logs critical parameters into a reserved location in data store, recovers from the error by resetting controller latches and the printing unit, and then returns I/O error response to the system. If the error involved a print hammer failure, a poll is taken of the current status of all print hammers and compared to the history of recent print hammer firings maintained in data store by the IOC microcode, and information about hammers detected to be failing either on or off is also logged.

References

1. E.F. Dumstorff, "Application of a microprocessor for I/O control," page 28.

Presents a description of the magnetic media controller used on the System/38 to attach high-data-rate, magnetic-media devices.

System/38 magnetic media controller

J.W. Froemke, N.N. Heise, and J.J. Pertzborn

During the early development stages of IBM System/38, it was necessary to develop an alternative to the conventional attachment techniques to facilitate the control of high data-rate, magnetic media, input/output devices. Engineering design trade-offs were evaluated based on a need for a general solution that would minimize both product cost and development cost while retaining performance capability and shortening the product development cycle.

Concept

Conventional I/O device attachment design techniques generally fall into one of two categories:

1. Hardwired controller, where most of the device control function is personalized at the AND-OR level, requires the development of many new chip part numbers for the attachment of each new I/O device. Low product cost and high data-rate performance are obtainable but at a relatively high development cost and long product development cycle.
2. Microprogrammed controller, where most of the device control function is personalized at a high level in microprogram-ROS (read only storage). This reduces the number of new chip part numbers, resulting in a savings of development cost and a shorter product development cycle for low to medium data-rate I/O devices.

The IBM System/38 magnetic media controller (MMC) architectural concept has evolved from these two I/O attachment design techniques. The channel function is hardwired in a set of chips while most of the device control function is personalized at a high level in programmable logic array-read only storage (PLA-ROS). This results in a performance capability for the channel attachment and direct control of high-data-rate magnetic media devices and a common usage set (menu) of chip part numbers. A common data flow was developed to interconnect the functions within the MMC in a consistent manner. The use of MMCs to attach the disk storage devices to the System/38 channel is discussed by Peterson [1].

Description

Figure 1 shows both the microprogrammed I/O controller (IOC) and the magnetic media controller techniques used for the attachment of I/O devices to IBM System/38. Figure 2 shows the functional data flow within the MMC. Overall, five similar functions exist in both cases:

1. A hardwired channel function provided for responding to the internal System/38 channel handshaking requirements on a real-time basis. This is implemented in bipolar technology as a set of channel chips designed for common usage with all magnetic media device attachments. Block transfer of data over the channel is provided in multiples of eight bytes to accommodate high-data-rate devices. A more detailed description of the channel operation is given by Lewis, et al [2].
2. A buffer function is provided for storing channel commands and status, data, and device control information. It is implemented as either 256 or 512 bytes of bipolar technology random access memory depending on the individual device requirements. The buffer operation is time sliced so that both the channel and the device controller have access to the buffer during each MMC 400 ns cycle time as shown in Figure 3.
3. A set of register chips is provided to send control information and synchronize the transfer of status and data to and from the device. Common usage of these bipolar register chips reduces the proliferation of new chip part numbers. The number of register chips used by an MMC depends on the number of signal lines to the device and its actual data rate.
4. A common set of driver/receiver modules is provided to satisfy the electrical characteristics of the internal System/38 channel.
5. A controller function is provided to control the channel, buffer, and device operations (e.g., read, write, seek, block check, error detection and multiplexing). This function is personalized at a high level in PLA-ROS (field-effect-transistor technology) as compared to microprogram-ROS in microprogrammed controllers or custom-designed logic chips in hardwired controllers. The number of PLA-ROS

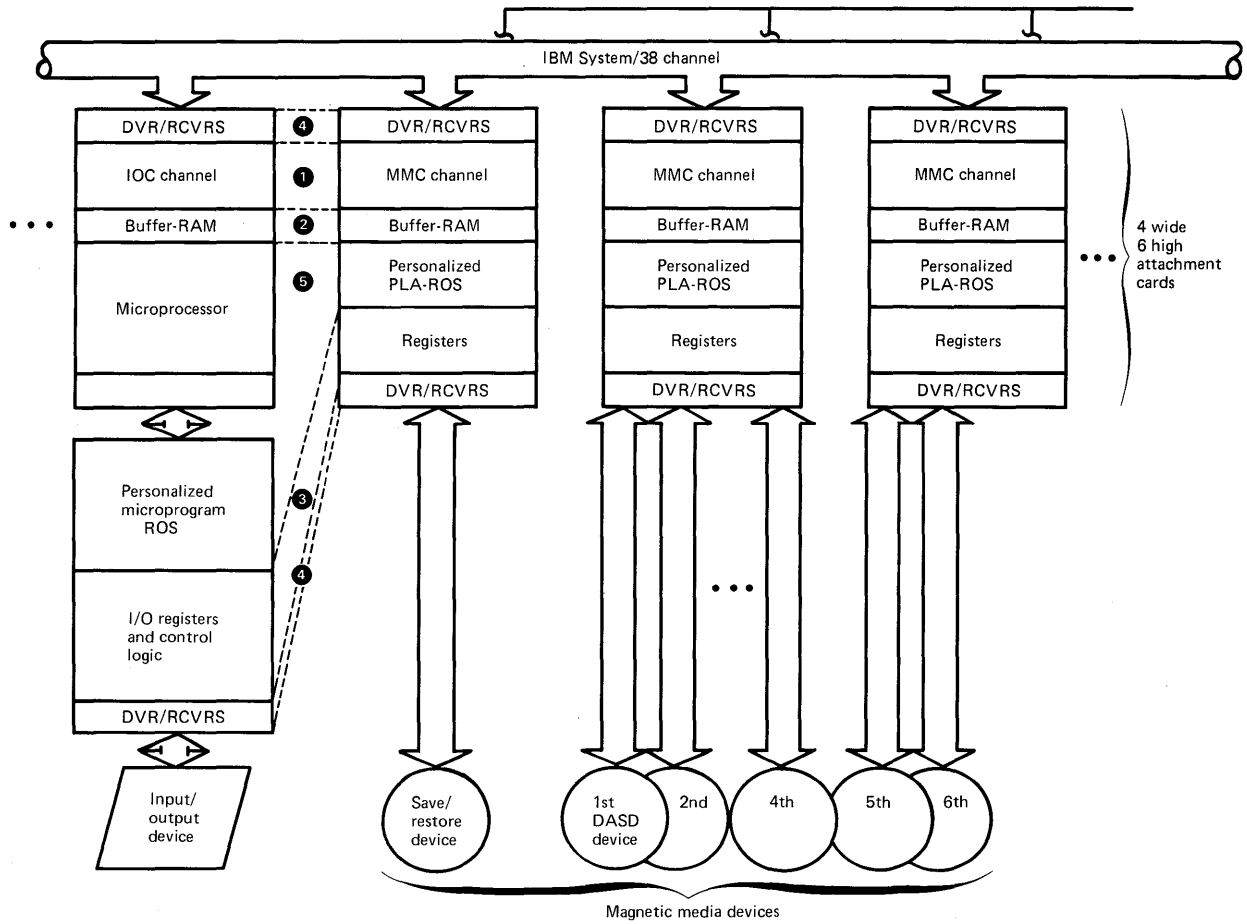


Figure 1 System/38 device attachments

chips used by an MMC is dependent on the number of devices and their complexity.

Performance

Performance requirements for the MMC were based on the combined characteristics of several, relatively high-data-rate magnetic media devices and the internal System/38 channel. Typical hard disk file, flexible diskette, tape and other magnetic-media-based devices have instantaneous data rates in excess of 100KB, serial by bit or by byte. Concurrent operation of multiple attachments on the system channel requires that the MMC provide adequate data buffering and timely transfer of data blocks over the channel. System commands must be transferred, decoded, and executed with orders given to the device. Appropriate responses with status data must be assembled, encoded, and transferred over the channel. During these channel operations, the controller must also be able to provide a response in less than 10 μs to multiple asynchronous signals from the device.

The MMC meets these performance requirements through the use of a hardwired channel function and a PLA-ROS based controller with a buffer time-sliced between them, as shown in Figure 3. During the 400 ns PLA cycle time, inputs are sampled, logical responses are generated, and outputs are activated. The logical power (analogous to the function and width of an ALU) of each PLA cycle is flexible. The number of operations (channel commands, data transfers, device control, etc.) being controlled simultaneously is also flexible. Overall, the PLAs within the MMC can be personalized to provide a high level of performance for the direct control of magnetic media devices having stringent characteristics. The on-chip combination of registers and logical function within personalized ROS provides PLA-based controllers with an inherent performance advantage over separately packaged configurations within the same technology. The ability to execute multiple, simultaneous operations is critical for the cost-effective control of magnetic media devices attached to the System/38.

Test and service

A common test and service approach has been developed for use with the MMC in the System/38 environment. That is, the same level sensitive scan design (LSSD) test patterns used by manufacturing are also used in the field for system fault isolation and repair verification. The application of LSSD test patterns has resulted in improvements in the test coverage of densely populated controller cards and their service costs without increasing product cost or lengthening the product development cycle. Use of LSSD throughout the MMC has also led to improvements in test generation and coverage at the chip level for channel and register chips.

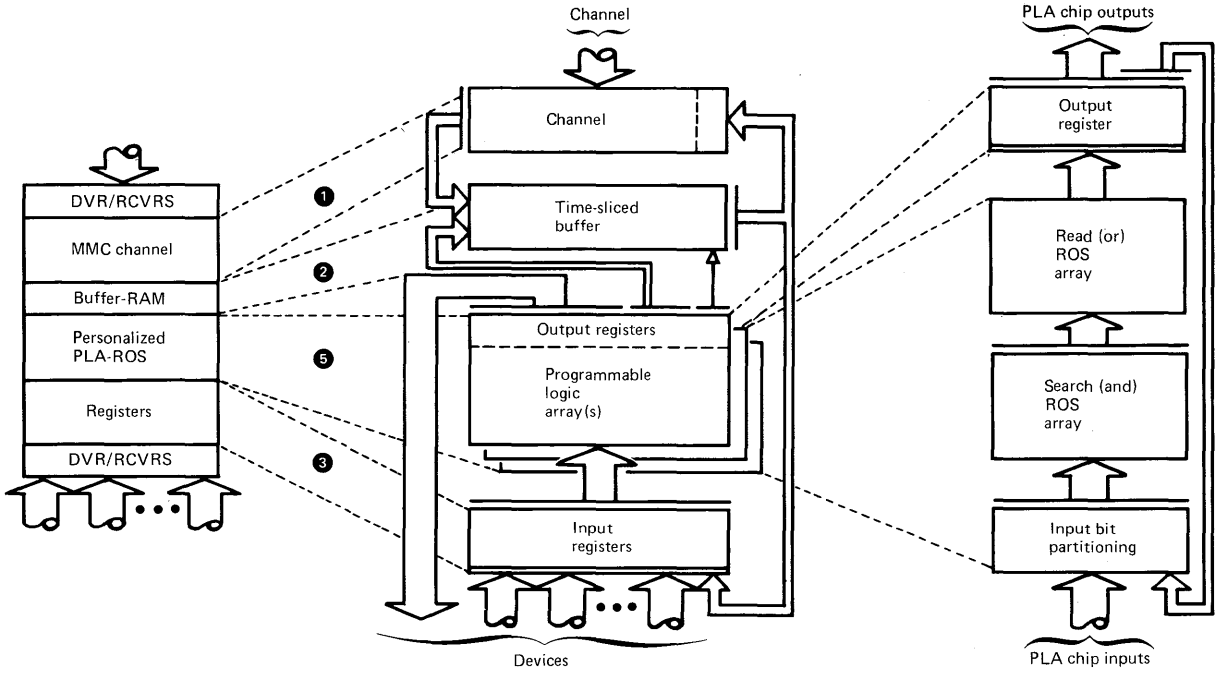


Figure 2 Magnetic media controller—functional data flow

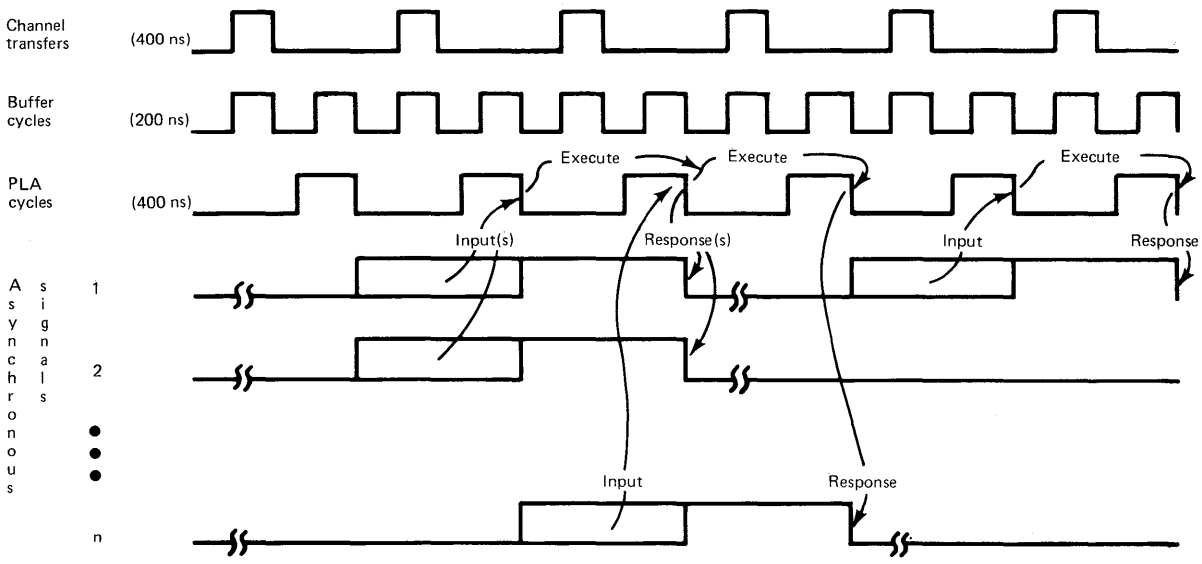


Figure 3 MMC simultaneous channel/device(s) operations

Summary

Overall, the design objectives of the System/38 magnetic media controller were met in the following ways:

1. The menu of bipolar chips (i.e., channel and register) is available for common usage with new device attachments. Personalization of logical function in PLA-ROS removes the designer from the physical design and test generation activities at the chip level. The result is a savings of development cost and a shorter product development cycle.
2. Extensive use of LSSD and a consistent data flow within the MMC have resulted in a savings of service cost and a shorter product development cycle.
3. Single card packaging (i.e., one field replaceable unit) of the entire attachment for one or more devices is made possible through the use of fewer high-density components. This results in a savings of both product and service costs as well as space for packaging additional attachments within the system.
4. Overlapped operation of the time sliced random access memory between the channel and device, together with the fast (400 ns) PLA cycle time, provides sufficient performance capability for the attachment of high-data-rate magnetic media devices.

References

1. R.A. Peterson, "Shared function controller design," page 44.
2. D.O. Lewis, J.W. Reed, and T.S. Robinson, "System/38 I/O structure," page 25.

Shared function controller design

Describes how the control of disk storage devices is accomplished on the System/38 by sharing multiple magnetic media controllers.

R.A. Peterson

IBM System/38 is a virtual storage machine with auxiliary storage consisting of one to six spindles of integrated, nonremovable disk storage. Data, in 512-byte blocks, is paged in and out of main storage across the system I/O channel, which is described by Lewis, et al [1].

The design approach for attaching the virtual storage subsystem to the I/O channel is described in this article. The high data rate of the disk and the hardware cost of the attachment were the prime reasons for developing a non-microprocessor design. The disk storage attachment on System/38 handles the function necessary to attach up to four spindles to the system I/O channel. The necessary hardware is contained on one 4 wide by 6 high card with separate clocking logic.

The attachment incorporates the concept of a shared function controller, whereby each major function has its own separate sequence controller. The term "controller," as used, is defined to mean a sequential state machine designed with logic circuits as opposed to a microprocessor. These controllers are built using high-density programmable logic array (PLA) technology [2]. The ability to subdivide the function into manageable pieces capable of being shared by all attached spindles made this approach feasible.

The attachment controls the disk storage for System/38, performing read, write, and diagnostic functions. The operation for each spindle is specified by an eight-byte command element containing the command, a starting address, and the number of blocks of data to be transferred. Six registers in the virtual address translator (VAT) are allocated for each spindle to point to command and data locations in main storage. The storage management function initializes these registers and issues a channel operation to inform the attachment that the command is ready to be obtained and operated on.

The connection between the attachment and the spindle consists of a bidirectional byte bus used for access and diagnostic sensing operations and a serial data bus. The attachment initiates the access, determines successful completion, and performs the serialization and deserialization functions and the cyclic redundancy check (CRC) function.

The attachment is divided into function controllers as shown in Figure 1. The channel controller, access controller, rotational position sensing (RPS) controller, read/write controller, and serializer/deserializer (SERDES) controller each control a portion of the data path and have access to the random access memory.

All data passes through the random access memory, allowing access to the necessary information by each function controller. The random access memory is divided into sections which contain command blocks, status blocks, and channel control blocks for each spindle. There is also an 8-byte ID block, an 8-byte header data block, and a 256-byte data block which are shared by all four spindles.

The five function controllers and random access memory are shared among four sequence controllers, one for each spindle. These sequence controllers are responsible for determining the proper action of each function controller and locking out each other while using a function controller. The four sequence controllers are located in one PLA. No data flow is associated with these controllers. A set of defined states exists within each sequence controller to represent the allowable command states as defined for the attachment.

Lockout technique

The lockout mechanism is important because the function controllers share a common data path and random access memory and are themselves shared by more than one sequence controller. Not all operations defined for the attachment can be serial with respect to each other. An example of parallel functions is the transfer of data across the channel while data is also being transferred to and from the device through the random access memory.

There are three levels of function controller lockout within the attachment. The first one involves the

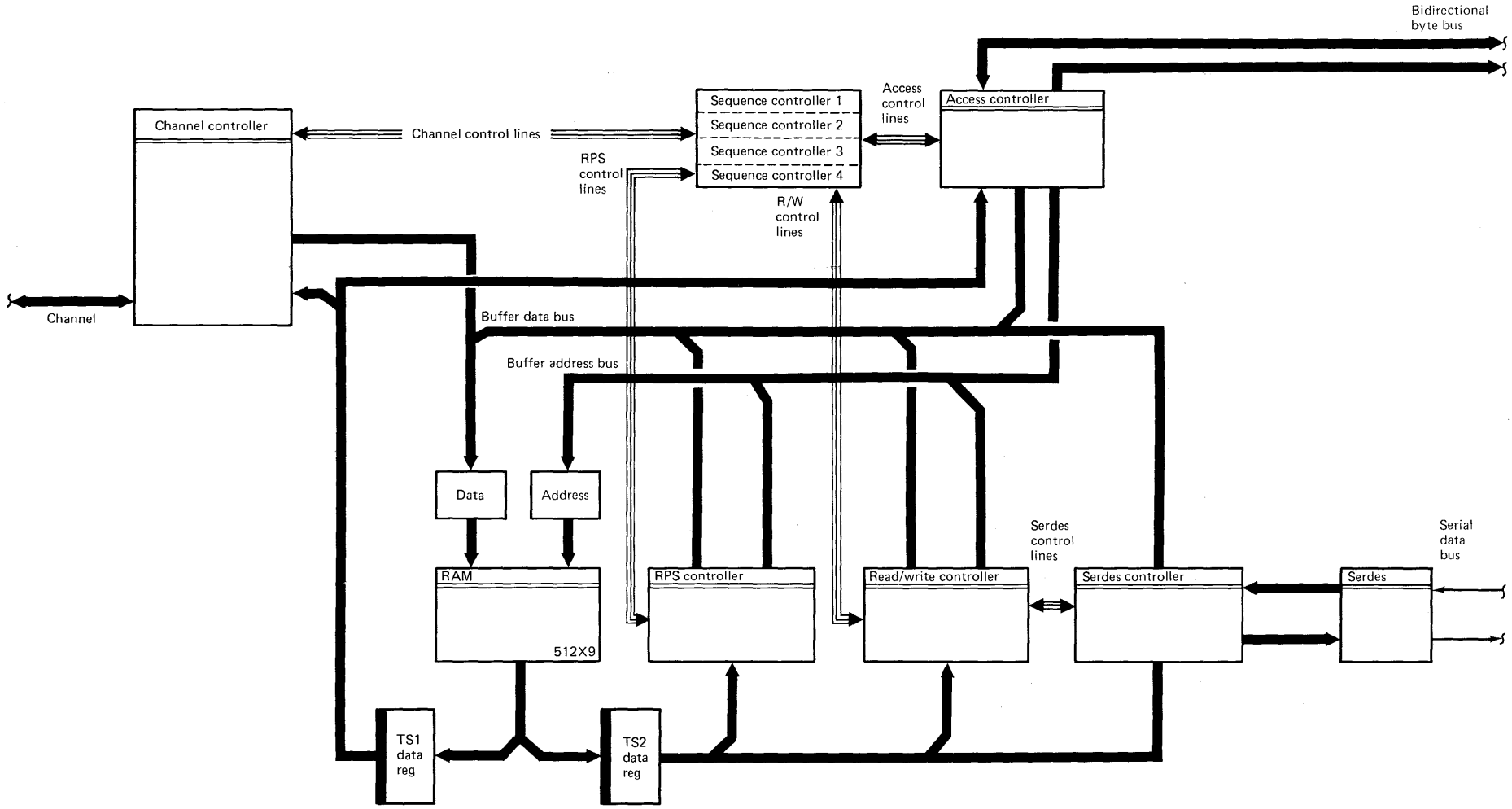


Figure 1 Disk storage attachment

timeslicing of the random access memory. The attachment runs on a 400 ns clock while the random access memory runs on a 200 ns clock allowing for two accesses per attachment cycle. The channel controller and access controller use the first time slice while the RPS, read/write, and SERDES controllers use the second time slice. This enables parallel operation between the channel and SERDES controllers.

The second level of lockout exists between the channel, access, and RPS controllers. No time-critical function or long operations which would adversely affect the channel interface controller are contained within the access and RPS controllers. The four sequence controllers each use a three-step operation to maintain this lockout. When one of the three function controllers is to be requested, the sequence controller goes to a request state (step 1). On the next attachment cycle, the status of the other three sequence controllers is checked for a busy state, and the status of the higher priority controllers is checked for a request state. The sequence controller for the first spindle is arbitrarily defined to have the top priority. If none of the three function controllers is busy and none is being requested by sequence

controllers of higher priority, the sequence controller goes to a busy state (step 2) and proceeds to operate the desired function controller. When the operation is complete, the controller goes to a wait state (step 3) for a minimum of one attachment cycle before requesting another function controller. The sequence controller for spindle 1 must stay in the wait state until the sequence controller for spindle 3 is in a wait state. The sequence controller for spindle 2 must stay in the wait state until the sequence controller for spindle 4 is in a wait state. This balances out the priority and ensures a window in which all sequence controllers can get the use of a function controller in a timely manner.

The third level of lockout involves the use of the function controllers on the second time slice. When the RPS controller determines that the head is now located one sector prior to the requested sector, the sequence controller is allowed to use the read/write controller which in turn determines what read/write operation is to take place. The read/write controller tells the SERDES controller what operation to perform and, by way of the sequence controller, requests data transfers from the channel controller. At the time the RPS controller determines the correct location of the head, a lockout is issued to all the other sequence controllers preventing the use of the RPS controller (and indirectly the read/write controller) since a determination of the correct location of the head on another spindle would be of no significance as long as the SERDES is busy. This lockout also extends out from the attachment for those commands which transfer data across the channel. This lockout signal is used to keep the channel from being overloaded by two high-data-rate attachments. The disk storage attachment monitors for a lockout from other attachments to inhibit the use of the RPS controller in the same manner as the internal lockout.

The design cycle was highly dependent on software modeling and simulation. The major spindle and functional changes were absorbed during the modeling phase. Most of the sequences were simulated at the high level, module level, and card level to ensure a degree of confidence in the design prior to embedding in a PLA module. This modeling effort proved to be a necessary step for a hardwired sequencer design in an LSI environment.

References

1. D.O. Lewis, J.W. Reed, and T.S. Robinson, "System/38 I/O structure," page 25.
2. J.W. Froemke, N.N. Heise, and J.J. Pertzborn, "System/38 magnetic media controller," page 41.

Characterizes the System/38 high-level machine instruction interface. Describes microcoded functions and the rationale for providing them.

System/38—A high-level machine

S.H. Dahlby, G.G. Henry, D.N. Reynolds, and 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 [1] 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 [2], auxiliary storage allocation and addressing [3], internal data structures and relationships [4], channel and I/O interface details, and internal microprogramming details [5].

This hardware independence characteristic of the System/38 instruction interface is due in large measure to the use of an object-oriented interface [4] 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 [6].

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 [2,4]. 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 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 [3], 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.

Integrity and authorization

A natural consequence of the object-oriented approach is improved system integrity and authorization mechanisms [2]. 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.

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.

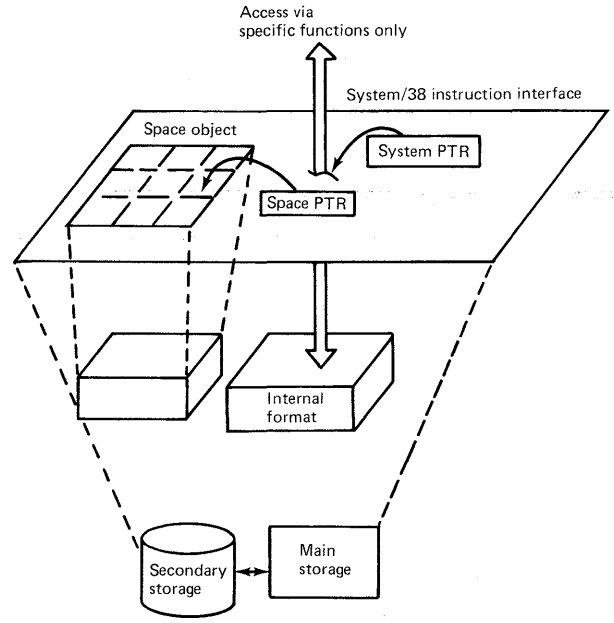


Figure 1 System/38 object-oriented structure

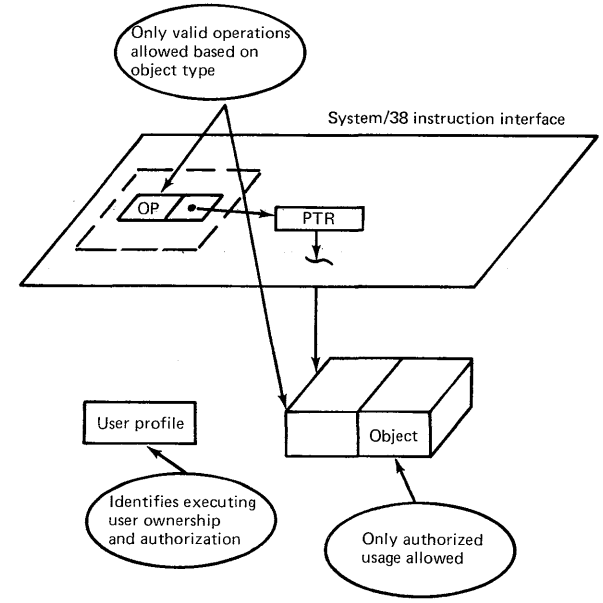


Figure 2 System/38 instruction interface integrity and authorization scheme

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 [6] and Howard [7].

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

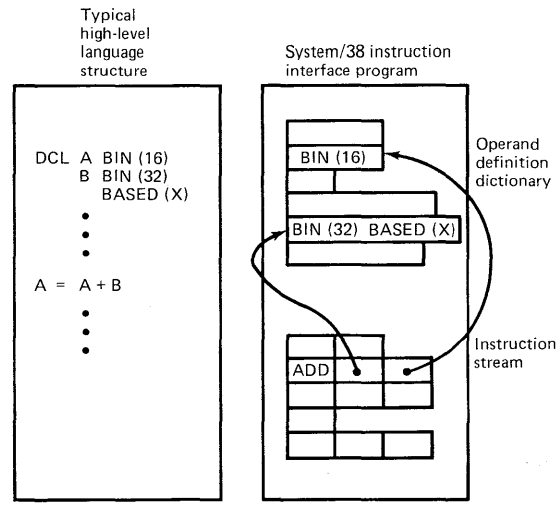


Figure 3 System/38 instruction interface program structure

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 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.

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.

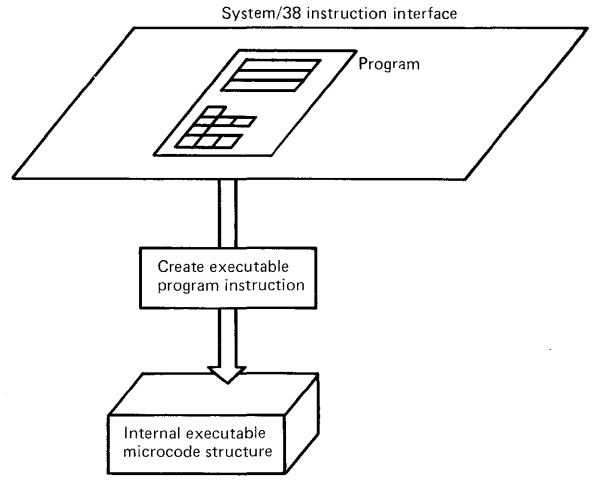


Figure 4 System/38 executable program creation

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 discussed 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

1. G.G. Henry, "Introduction to IBM System/38 architecture," page 3.
2. V. Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.
3. R.E. French, R.W. Collins, and L.W. Loen, "System/38 machine storage management," page 63.
4. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.
5. R.L. Hoffman and F.G. Soltis, "Hardware organization of the System/38," page 19.
6. C.T. Watson and G.F. Aberle, "System/38 machine data base support," page 59.
7. P.H. Howard and K.W. Borgendale, "System/38 machine indexing support," page 67.

Discusses addressing, authorization, locking, and synchronization in System/38.

System/38 addressing and authorization

V. Berstis, C.D. Truxal, and 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 [1].

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 Figure 1), which consists of a functional portion and an associated space (see Pinnow, et al [2]). 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.

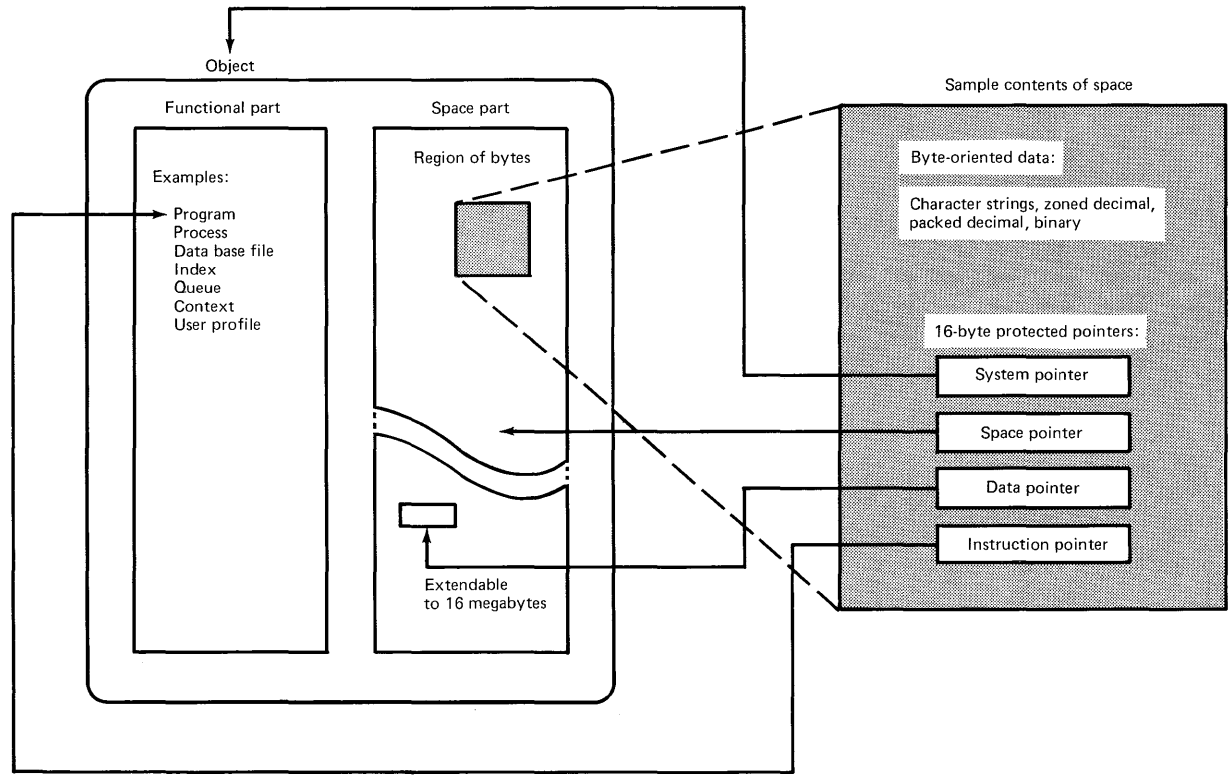


Figure 1 System/38 objects and pointers

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* [3].

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 Figure 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.

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.

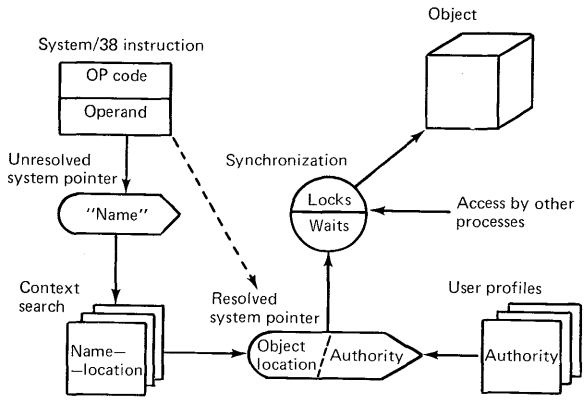


Figure 2 The object access path

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

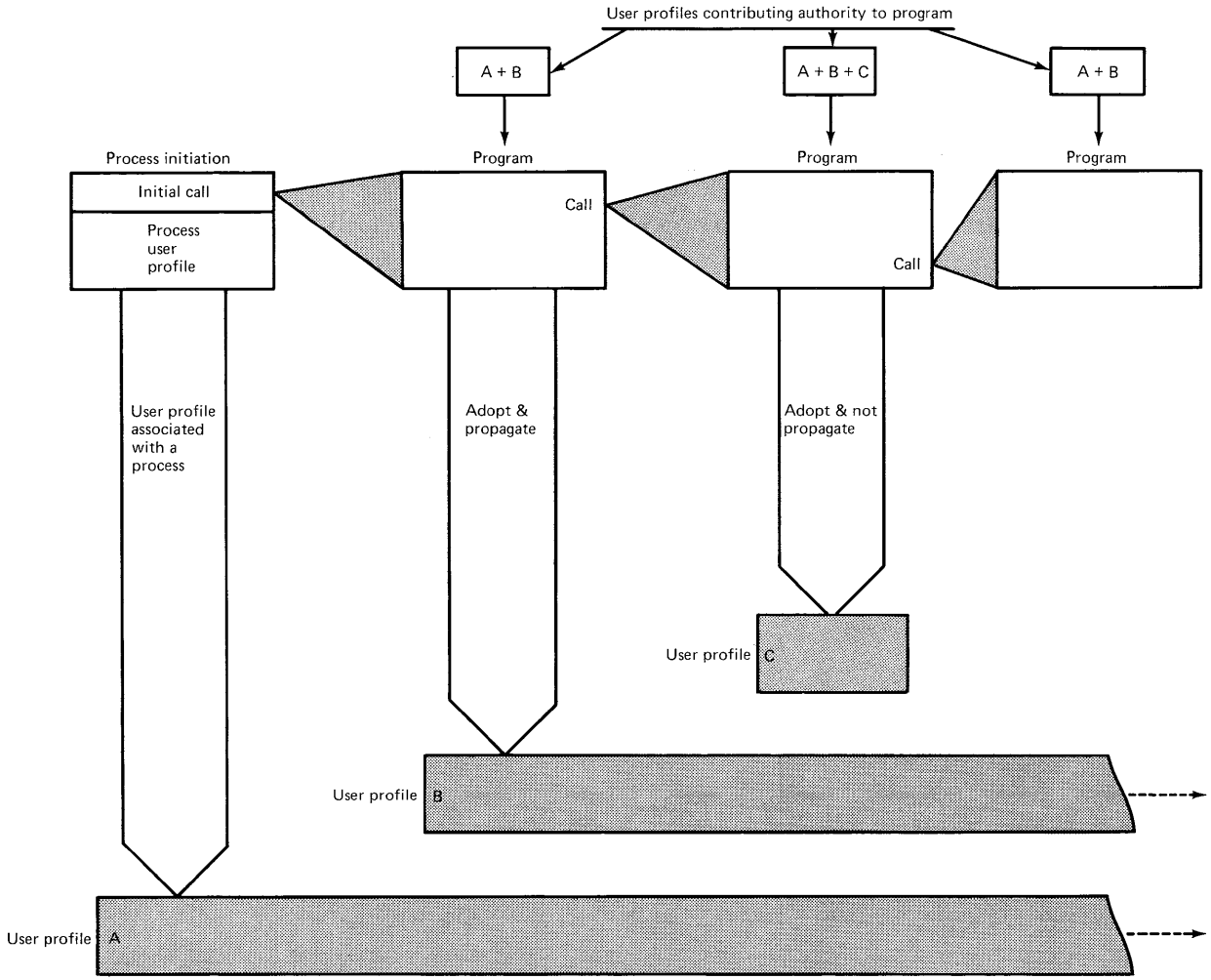


Figure 3 User profiles as sources of authority

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 user 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 [1]. 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 [2]) 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

1. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
2. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.
3. Theodore A. Linden, "Operating system structures to support security and reliable software," *Computing Surveys*, Vol. 8, No. 4, 409 (1976).

States that the construct, "object," is of central significance in System/38. Discusses the concepts, purpose, and characteristics of System/38 machine objects.

System/38 object-oriented architecture

K.W. Pinnow, J.G. Ranweiler, and 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 [1] 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 Figure 1.

The three examples of attributes illustrated in Figure 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 [2]. 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 Figure 1.

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

Figure 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 [3]. 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 [4].

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 sub-type, 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.

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.

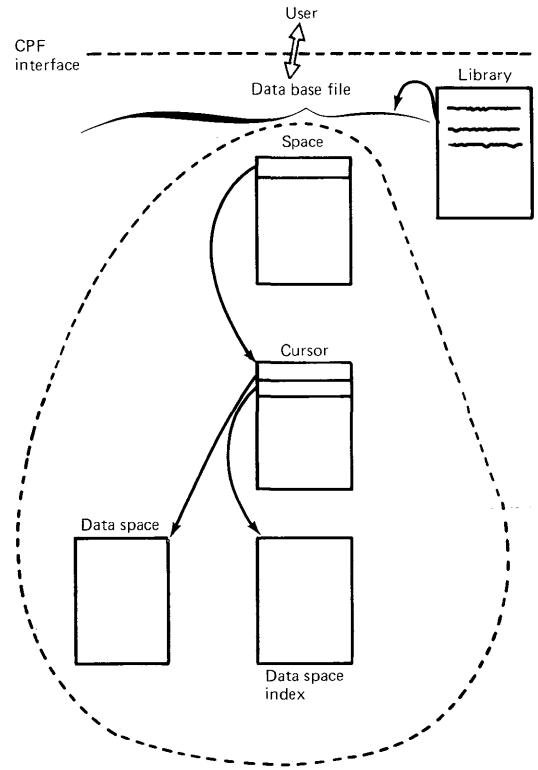


Figure 2 An example of how one kind of Control Program Facility object is constructed

The key advantage 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 aspects 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 for-

mat 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

1. J.K. Allsen, "System/38 common code generation," page 100.
2. V. Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.
3. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
4. R. A. Demers, "The generalized message handler in System/38," page 97.

States that much of the data base function in System/38 is provided by the machine instruction set. Describes the machine support and discusses performance, security, integrity, and ease-of-use considerations.

System/38 machine data base support

C.T. Watson and G.F. Aberle

One of the advanced features of System/38 is its data base facility. The data base facility is a primary component of the Control Program Facility (CPF) [1,2] and much of the data base function is supported by the machine instruction set. This data base support is used by the CPF data base component, which adds ease-of-use characteristics. The intent of this article is to describe this unique microcoded data base support and to discuss the factors considered in determining the level of support to be provided. A short description of the microcoded function is followed by a discussion of how performance, security, integrity, and ease-of-use considerations affected the determination of the level of support to be provided by the machine.

Figure 1 shows the comparative level of function provided in the microcode support for the various divisions of data base function. In general, for the "data transfer" instructions of the CPF data base component, as much function as possible was placed in the machine to improve performance. For the other divisions identified in Figure 1, the distribution was dictated by function and the desire to have as little logic in microcode as possible. In Figure 1, each column indicates the proportion of the function that is provided by the CPF code or by the machine-provided support. For "file definition" all the function is provided by CPF, while for "data transfer," the function is almost entirely provided by the machine support.

The microcoded data base support

System/38 is an object-oriented machine and its machine-instruction interface is an object-oriented interface [3]. The objects provided by the microcoded data base support are the data space, the data space index, and the cursor.

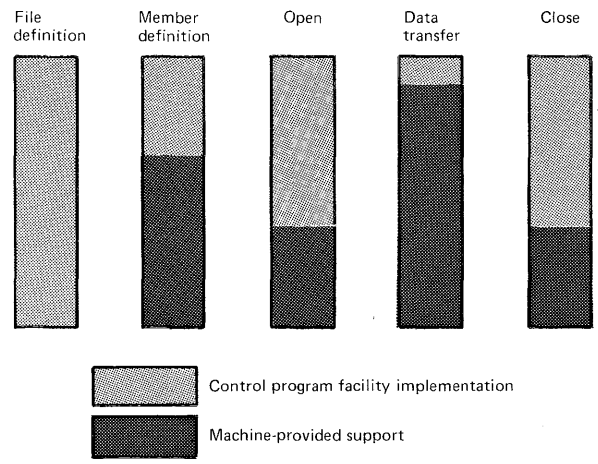


Figure 1 System/38 data base facility functional distribution

The data space is the data storage object. It is an arrival sequence file containing records of a single format. The microcode supports record lengths up to 32K bytes and files up to 256M bytes.

The data space index is used to provide access paths other than arrival sequence. The data space index provides a logical reordering of the records in one or more data spaces, based on keys made up of field values in the records and constants called "fork characters." The data space index can provide a logical reordering of records in one data space, a logical merge of like format records from several data spaces, or a logical hierarchical ordering of records of different record types from different data spaces. The key definition provides this extensive ordering capability. The key is made up of fields from the record, and fork characters, in any order. For each key field, ordering attributes can be specified, such as ascending, descending, absolute value, alternate collating sequence, etc. The fork characters are one-character constants that provide very powerful ordering functions for duplicate keys or duplicate portions of keys.

Figure 2 shows the forced hierarchical ordering of duplicate supplier numbers in different record types through use of fork characters.

Assumed key definitions:

Supplier file	Supplier #	1	
Part file	Supplier #	2	Part #

where "1" and "2" are fork characters.

Figure 2 also shows how records from one or more files can be logically reordered through key definitions

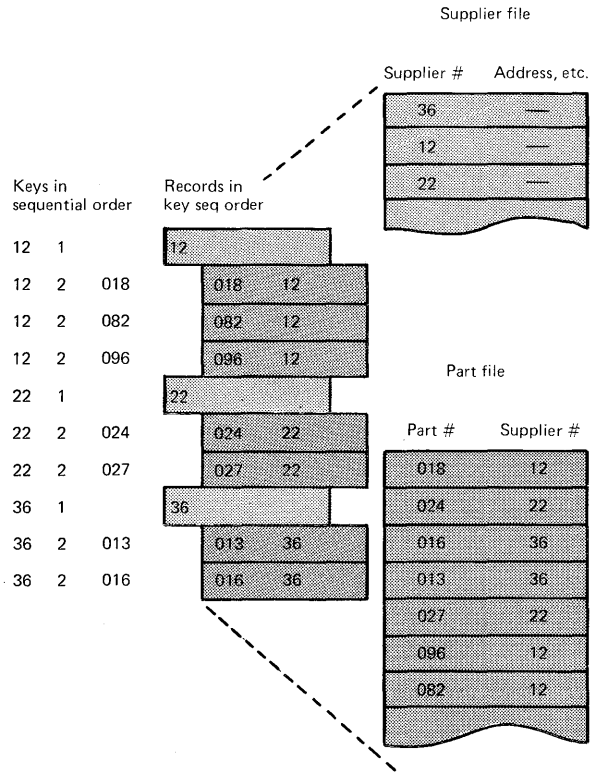


Figure 2 Use of keys in forcing logical hierarchical ordering

supplied with a data space index. The use of fork characters forces the record from the supplier's file to be returned before the records from the part file supplied by that supplier. Note that the part number field (or the supplier field) can be defined to be ordered in ascending, descending, absolute value, or other sequence, independent of the fork characters.

By definition, a key exists for each record contained in each file covered by the data space index. A feature is provided to allow a subset of the available records to be addressed based on values in the record. This facility allows the records to be partitioned into logical files entirely differently than actually stored. Through one set of data space indexes the data appears to be separated by days of the week, while through another set, the same data appears to be separated by sales department. The actual partitioning of the data may be done by either of these or by an entirely different method. The data space index is immediately updated to reflect changes in the data spaces it covers.

The data space index is implemented through use of a general machine index function that is also used by many other components and objects in System/38. It is based on a binary search algorithm and is optimized and balanced for performance among modification, random searches, and sequential searches [4].

The cursor connects the data spaces and optionally a data space index to the process. It contains current position information for use in "next" type operations. If the data space index is not specified, the access path is either direct or sequential by arrival sequence over one or more data spaces where the data spaces appear to be concatenated. If the data space index is provided, the cursor provides a keyed access path for either random or sequential retrieval based on the key values or a direct access path via the arrival sequence relative record number. A very extensive set of search operations is supported.

The cursor is the facility through which records are locked between retrieval and modification. The cursor also contains the logic necessary to map the physical record to and from the logical record defined by the user. This mapping provides for reordering the fields, subsetting the fields, and conversion of the fields to new types and lengths, thus supporting the logical file format capability.

As shown in Figure 3, physical and logical files are implemented through use of the three machine data base objects. The data space in the physical file contains the actual data. A data space index is used where a keyed access path is specified. A prototype cursor is duplicated for each shared usage of the file

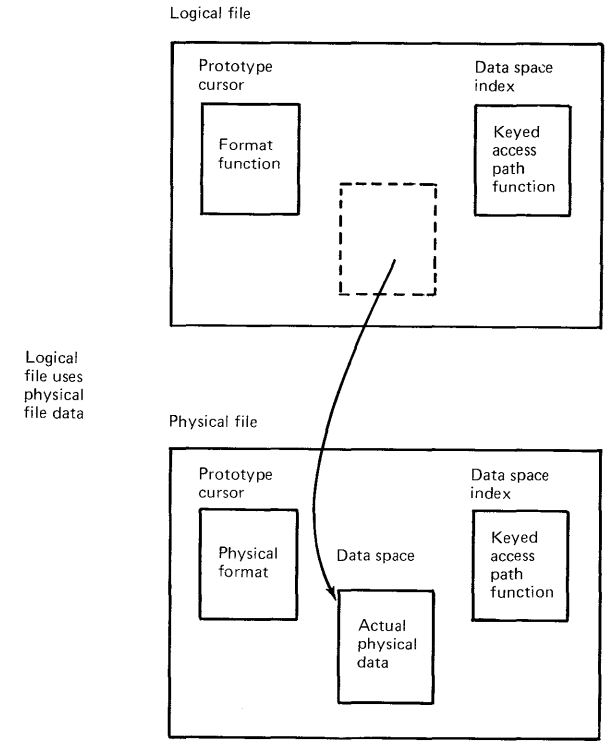


Figure 3 Machine object usage in implementing physical and logical files

and contains the format function. The cursor, data space, and data space index are used by the CPF data base component to implement logical and physical files.

Performance

Because System/38 is a data base machine and all of the user's file data is maintained by the data base facility, performance has been a very important consideration in the design. A very good way to improve performance is to move closer to the hardware. In System/38, this was done by providing the data base search and modify operations in the machine. This can be seen in Figure 1 in the column called "data transfer." Machine instructions have been tailored to do, almost entirely, the work of the Control Program Facility GET, PUT, UPDATE, and DELETE functions. First, last, next, previous, generic next, generic previous, and direct search functions are provided through a SET CURSOR instruction which supports the positioning portion of GET. The record, as defined in the logical format, is returned by a RETRIEVE instruction for the data portion of the GET. The UPDATE, DELETE, and PUT functions are handled by UPDATE, DELETE, and INSERT machine instructions which automatically update all the data space indexes to reflect the change.

The machine does all space management for the data base facility, utilizing the machine storage management component. The microcode has been thoroughly optimized to shorten "normal" path lengths and to reduce the number of pages touched, whenever possible.

Authorization

A primary objective of System/38 is to improve security of the system, and the data base facility in particular, over previous systems. To effect this, object authorization was implemented in microcode below the lowest user-available interface. The objects are stored on auxiliary storage, but there is no interface available to the user for reading from, or writing to, this medium. All operations are made through well-defined, object-sensitive, microcode instruction interfaces which enforce object authorizations [5]. In systems implementing security and data base functions above a user-available interface, there have been ways to bypass the security of the data base. With System/38, the authorization and most data base functions are implemented below the lowest user-available interface, thus providing greatly improved security capabilities. It is for these reasons that measures like file keyword locks and data encryption are not considered to be necessary for the System/38 data base facility.

Integrity

The attributes of integrity are very similar to those for security. Because of a desire to improve data integrity in the data base facility, enough of the data base function was implemented in the machine to ensure that changes to objects must be made through the correct interfaces as defined by the architecture. In past data base systems, the user was able to bypass the data base support code and access the files through the file management interface, thus defeating integrity functions.

In System/38, all data base operations are done in move mode. The data base user never has access to, or the address of, the actual physical record. The most "physical" address that is available to a record is the object pointer and the arrival sequence number of the record in question. The only way this information can be used is through the appropriate data base support instructions.

In moving data in and out of the data base, some data checking is done under certain conditions, but for the most part, actual field data values are not checked. The data base facility does not explicitly do any range checking or guarantee any data-type validation. These are perceived to be user defined requirements. On the other hand, usage of the centralized data definition capabilities will greatly reduce the chance of data-type errors [2].

Individual records are locked so that another process cannot change the record between the time a program reads it and the time the program is ready to write back the updated fields.

Because of the high level of data base support provided by the machine, many recovery-type integrity questions are solved. On process termination, files are automatically closed. Logical access paths will always represent the actual stored records, or will be marked as unusable and needing rebuilding. Sequentiality of inserts is guaranteed. The facility to protect random updates and deletes is also provided.

Ease of use

Ease of use is one of the predominant design considerations of System/38 and the data base facility. Where not impacted by security, integrity, or performance, the ease-of-use functions lie entirely above the machine interface. File definition is a good example.

The actual file definition, building of control blocks, and system definition of data formats and field attributes are done entirely outside the machine-provided data base support. The file member definition has some machine support due to the creation of the data base objects involved. When the file is created, the format and access path definitions are stored with no machine data base support. When a file member is created, the machine data base support is used to create the data space, data space index, and cursor that internally make up the physical and the logical files.

Summary

A unique feature of System/38 is its high level of machine function. In the data base facility, this is even more apparent than elsewhere.

Because of the efforts to provide good performance, the search and data transfer operations were implemented in the machine. Implementing security below the lowest user-available interface, and providing data integrity at that interface, resulted in objects being defined at the machine level and part of the OPEN and CLOSE functions to be implemented in the machine. The result is that, for the first time, optimum data base support has been provided at the machine instruction interface.

References

1. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
2. C.T. Watson, F.E. Benson, and P.T. Taylor, "System/38 data base concepts," page 78.
3. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.
4. P.H. Howard and K.W. Borgendale, "System/38 machine indexing support," page 67.
5. V. Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.

States that primary and secondary storage in System/38 is managed by microcode. Describes key implementation concepts.

System/38 machine storage management

R.E. French, R.W. Collins, and L.W. Loen

In all current IBM systems utilizing virtual storage management techniques, a job executes in a large virtual address space containing job-related structures and programs. A storage management component manages the transfer of portions of this address space to and from main storage as required. Separate data management components using different disk interfaces and mapping techniques manage the transfer of data between disks and buffers in the address space. One of the major innovative features of System/38 is that, during normal operations, the storage management component, which is part of the microcode, provides the *only* interface to disk storage, and all programs, files, and work spaces are managed as address spaces. All System/38 components thus address data on disk uniformly through this component, greatly simplifying the design of the system. For example, the data base component on System/38 is not concerned with buffers and disk I/O programs, but simply addresses a desired record in a file by its virtual address, relying on storage management to actually bring the data into main storage.

Figure 1 illustrates the relationship of storage management to other System/38 components.

This paper will describe key implementation concepts of System/38 storage management.

Basic addressing structure

All data on System/38 is addressed, whether by microcode or by System/38 encapsulated programs,[1] through six-byte virtual addresses, each consisting of a 3-byte segment ID and a 3-byte offset within the segment.[2] Storage management maintains directories relating all valid virtual addresses to disk locations. When a required piece of data is not present in main storage, the System/38 instruction processor generates an address translation exception. Storage management receives control, determines from its directories the location on disk of the data, and transfers the data (one or more blocks, called pages) to main storage where it may now be accessed by the processor.

When an object is created (i.e., a System/38 instruction-interface create-object instruction is issued), the microcode component associated with that object requests storage management to allocate one or more

[1] Encapsulation is the process, similar to compilation, which a System/38 program undergoes before it is executable. An encapsulated program addresses data in the same fashion as microcode components.

[2] System/38 hardware employs an addressing scheme with 4-byte segment IDs and 2-byte offsets. To facilitate management of large objects, storage management assigns 256 contiguous 64K segments at a time, so that each segment potentially addresses 16MB. Only the amount of space actually required is allocated, however.

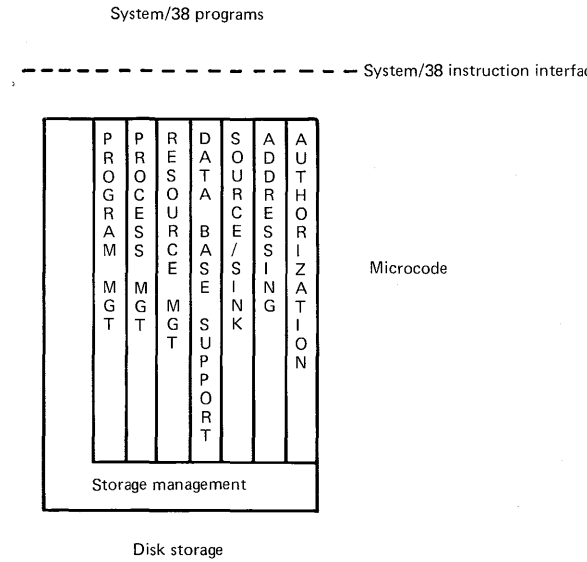


Figure 1 Relationship of storage management to other System/38 components

segments. For each such request, storage management assigns a unique segment ID, allocates the amount of space requested on disk, and updates its directories accordingly. The requesting component then addresses data within the segment through the segment ID and offset as described above. This address is not directly seen by the System/38 creating program. Instead, the 6-byte address becomes part of the 16-byte system pointer through which the program subsequently references the object. Storage management is also invoked when a microcode component requires a non-object related segment, for example, a work area. Thus, storage management manages segments, not objects. The segments thus allocated may subsequently be extended, truncated, and destroyed. Once a segment is assigned a location on disk, it remains at this location until destroyed.

Directory structure

At a given time, many thousands of objects, implying many thousands of segments, will exist on the system. These segments will range in size from a few hundred bytes to millions of bytes. Some will exist only for a few seconds, while others will exist for the life of the machine. Some objects, such as data space indexes, will be accessed one page at a time; others, such as programs, will generally be required in main storage in their entirety; still others may be required in main storage in conjunction with a number of other objects. The directory structure is designed to satisfy these requirements with minimal overhead.

Table 1 summarizes the storage management directories. The *temporary directory* describes the allocation of segments that are generally required only for the duration of a job and need not exist across IPLs. There are generally relatively few such segments, but at a given time most of them are in use. The *permanent directory* describes segments which must exist across IPLs, for example, data spaces and programs. There are generally many permanent segments, relatively few of which are in use at any given time. Use of separate directories thus enables faster look-up and allocation operations on the more frequently used temporary segments. (The indication of which directory to search is encoded in the virtual address itself.)

Together, the permanent, temporary, and free-space directories describe the allocation of all disk space. These directories are implemented as pageable indexes.[3] Contiguous blocks of space on disk are called extents, and are always a power of 2 in size, ranging from 512 bytes (one page) to 16MB. Each such extent is described by a 4-byte extent descriptor containing the extent size (a 4-bit code which is the log of the number of pages) and the disk address. The free-space directory consists of extent descriptors mapping all unallocated space, while the permanent and temporary directories each contain entries consisting of a segment ID followed by one or more extent descriptors mapping the disk allocation of the segment.

Allocating space in blocks rather than as single pages facilitates multiple page transfers to and from disk and reduces directory size. The power of 2 scheme enables the concise encoding of a vast range of sizes, further reducing directory size. Additionally, it makes possible the use of a "buddy" scheme[4] to partition or recombine extents when allocation and deallocation requests are received. The buddy scheme, in conjunction with a best fit algorithm for allocating space, is used to minimize disk fragmentation.

To reduce the number of permanent and temporary directory searches (which could cause additional page faults since the directories are pageable), a main storage resident *lookaside directory* is employed. This directory consists of an array of entries, similar to those in the permanent and temporary directories, describing the location on disk of the most recently referenced segments. It is managed on a least-recently-used basis and is accessed by hashing the address to be resolved and using the result as an offset into the directory. Typically, 90% of all directory lookups are satisfied in the lookaside directory without additionally searching the permanent or temporary directory.

Often, several objects are required simultaneously in main storage. For example, the arrival of a transaction to the system may require that a number of work areas and control blocks needed to process the transaction be placed in main storage. To reduce the number of separate directory searches and disk accesses, a number of objects may be packaged together in an *access group*. Space for the access group itself is described in the temporary directory. Space for the individual object is then suballocated from within this space. This suballocation is described in an access group table of contents. The table of contents is a linear list of the contents of each page of the access group, enabling storage management to transfer the set of objects to or from main storage in a single disk operation. An access group thus facilitates a roll-in/roll-out or swap type of operation. However, the concept is generalized to include any set of objects referenced more or less simultaneously. From a performance standpoint, access groups are a virtual extension of the object-oriented interface [2].

It must also be possible, although not desirable from a performance standpoint, to reference a segment when its containing access group is not in main storage. This is accomplished through the access group member directory, an index which relates segments to their containing access group.

Recovery

Since the directories describe the location of all data on disk, they must be recoverable in the event of a

[3] The System/38 microcode includes a component which supports a variety of search, insert, and remove operations on indexes. This function is used by storage management as well as by a number of other microcode components [1].

[4] Associated with each block of space is an *adjacent* block of space of the *same size*, called a buddy. Since the buddy's size is known, its address may be determined. When a block of space is freed, a single lookup operation is performed to determine if the buddy is also free, in which case the two may be recombined.

Table 1 Storage management directories

Name	Organization	Entry size	Entry format	Number of entries	Number in system	Accessibility	Typical size	Typical number of entries
Permanent Directory	Index	10-22 bytes	virtual address + 1-4 extent descriptors	1 or more per segment	1	Pageable	128K-1M	5000-40,000
Temporary Directory	Index	10-22 bytes	virtual address + 1-4 extent descriptors	1 or more per segment	1	Pageable	5K-50K	200-2000
Lookaside Directory	Array	10 bytes	virtual address + 1 extent descriptor	1 per extent	1	Resident	4K-8K	400-800
Free Space Directory	Index	4 bytes	1 extent descriptor	1 per extent	1	Pageable	6K-48K	500-4000
Access Group Member Directory	Index	14 bytes	virtual address of segment, virtual address of access group, size of segment	1 per segment in an access group	1	Pageable	2.5K-25K	100-1000
Access Group Table of Contents	List	8 bytes	virtual address, disk address	1 per page in an access group	1 per access group	Pageable	1.5K	150

system failure. Instead of using a recovery scheme that adds overhead to normal system operation, such as journaling changes or maintaining two copies of the directory, all pages on disk are made self-defining; that is, each page on disk is preceded by an 8-byte header containing the virtual address of the page. The header is always read and written along with the page. In the event of a failure, the directory can be rebuilt by reading all records on disk. (Since the disks may be read sequentially, a track at a time, and since large blocks of allocated space may be skipped over, the time to recover the directories is not excessive.)

A natural fallout of the self-defining page concept is that a first reference to a page of a segment is detected as a mismatch between the header that is read in with the page and the virtual address being read. When such a mismatch occurs, storage management clears the page before returning to the user.

This enables storage management to provide a page of zeros when a page is first referenced, and without the overhead of maintaining a bit map of never-referenced pages or of zeroing space on disk when a segment is created or destroyed.

Summary

All disk storage on System/38 is managed by a single microcode component: storage management. The advantages of a simple virtual storage addressing scheme are thus extended to all other components of the system. Performance considerations are met primarily through a flexible, multilevel directory structure.

References

1. P.H. Howard and K.W. Borgendale, "System/38 machine indexing support," page 67.
2. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.

States that one of the novel features of System/38 is the integrated capability in the machine for creating, using, and maintaining indexes. Describes indexing features, implementation, and rationale.

System/38 machine indexing support

P.H. Howard and K.W. Borgendale

Indexes are one of the fundamental building blocks of system and application programming. An index is a logically ordered set of entries: Each entry consists of a key that is used to define the ordering and associated data. Typically, the key is the name of an object and the data is the address of that object. Indexes are used in many components, such as data management to access records in data files and in compilers to manage symbol tables.

One of the novel features of IBM System/38 is the integrated capability in the machine for creating, using, and maintaining indexes. This capability is used internally to support many of the basic machine functions, such as data base and storage management [1]. It is also made available to the users at the System/38 instruction interface via the independent index instructions. This provides the Control Program Facility (CPF) [2], compiler, and utility products a powerful and standardized index function for their many diverse uses.

In addition to being integrated into the machine, the System/38 index support has an advanced implementation approach utilizing a binary radix tree structure with special considerations for paging. This approach was required due to the challenging requirements of generality, performance, and function that had to be met to allow the machine index support to be used as a building block for many diverse functions.

This paper discusses a few salient features of the basic indexes and the rationale for the implementation choice.

Applications

The IBM System/38 uses index operations for sorting and table searches and to maintain cross-reference lists. Indexes are used in storage management, data management, context management, symbol tables, message handling, and authority checking. This high usage requires an efficient implementation. Some indexes are small, and others may contain up to 16 million bytes.

In all of these uses, there is a need for accessing an entry by a key value. Many require access to a prior or subsequent entry as well. Fast access is another requirement. For fast access it is important to keep the index compact and minimize the number of pages referenced.

Several techniques help to keep the indexes compact. First, the uses are designed to minimize the number and length of entries. The ability to handle variable-length entries eliminates the need for including unused fields. The indexes are implemented to minimize the overhead per entry. Common text, the leading portion of a key that is common to two or more entries, is stored only once. Each entry that is added to the index requires only a few bytes of storage in addition to its unique text.

The second consideration in achieving fast access is to minimize the number of pages that are referenced on each access. This is important when the pages reside in slower secondary storage. For uses that involve sequential access to entries, the problem is alleviated by placing sequential entries on the same page.

Another requirement of most applications is that the index operations be convenient. For this reason, all space and pages required for the index operations are managed by the index function. Both the key portion and the data portion of an index entry may vary in length up to a combined length of 128 bytes.

Binary radix tree

The System/38 uses a form of a binary tree to implement indexing. The primary reason for this choice is that it permits fast access to the individual entries. It differs from the usual binary tree in that a single bit is used to make each decision.

A binary search is used to bisect the index sequence on each iteration. Normally, this is implemented so that the search key is first compared with a key near the middle of the index sequence. Then, if the search key value is higher (lower), it is compared with a key near the middle of the upper (lower) portion of the index sequence. This process is repeated until an exact match is found.

For the System/38 indexes, the concept remains but the details are changed. Instead of comparing the whole search key with the various index entry keys, a single bit of the search key is tested on each step and, thereby, it is determined which path will be followed on the next step. This process is faster and nearly as informative. Certain bits of the search key are not tested because all of the candidates (those index entries which have satisfied all of the prior tests) have those bits in the same state. Each node of the decision tree must identify one bit of the search key to be tested and, as a result of that test, select one of two possible successor paths. This process is repeated until there is only one candidate left. A termination text element is used to identify the unique text of the proper index entry.

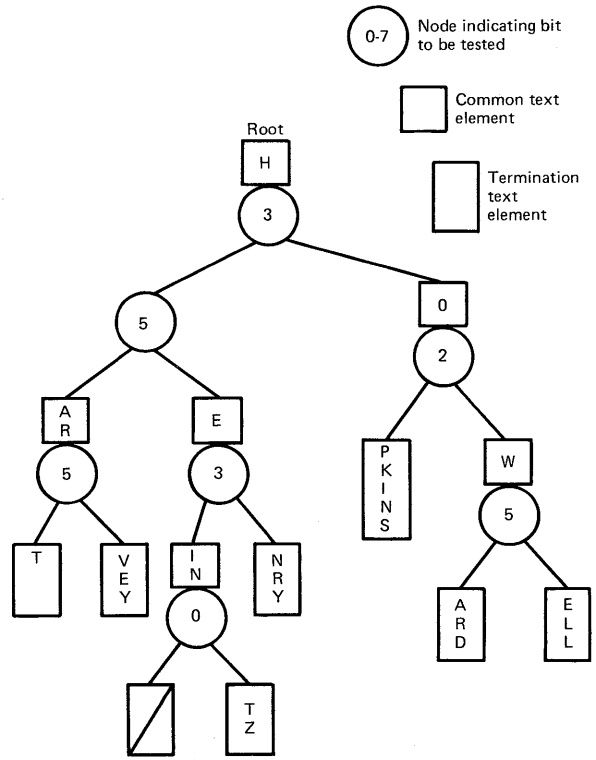
The tree is composed of nodes and text elements. Each entry is represented with a separate path from the root through a termination text element. The text elements define the location and length of text that logically belongs at each point along the path. Each node defines a test that will help select the path to be followed for a given search key. It identifies the search key bit to be tested. It also indicates the relative locations of successor and predecessor elements.

Figure 1 is a representation of these nodes and text elements for a tree with eight entries. The same type of text element is used for text that is common to several entries and for the termination text of a single entry.

Figure 2 shows an analysis of the same entries. It indicates that the entries have different lengths and that the keys are also of different lengths. The key for an entry is defined as that set of leading bytes which is sufficient to uniquely distinguish it from all others. Note that subsets are included by providing a null text element.

Data representation

Every entry in a System/38 index is treated as a variable-length bit string. The acceptable lengths are multiples of eight bits up to 1024 bits (128 bytes).

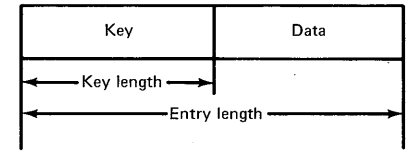


Note: The node values assume that text is represented in EBCDIC.

Figure 1 Binary tree with eight entries

The entry's length, in bytes, must be defined when it is inserted into the index. Its length is returned whenever the entry is retrieved.

Before a new entry is inserted into the index, its synonym entry is found. The synonym entry is that entry which is found when the new entry is used as a search argument with the existing decision tree. If the new entry is an exact match for the synonym entry, then it is rejected because it is a duplicate. Subset entries are allowed; the shorter entry is placed ahead of the longer one. Otherwise, the first bit that differs between the new entry and the synonym is used to discriminate between them. The decision tree is modified along the path leading to the synonym so as to include the new test. The new test node is positioned along that path so that the search key bits are always tested in a strict left to right sequence; this keeps the entries in order.



Entry	Bits tested on search	Entry length	Key length
HART	3	4	4
HARVEY	3	6	4
HEIN	4	4	4
HEINTZ	4	6	5
HENRY	3	5	3
HOPKINS	2	7	3
HOWARD	3	6	4
HOWELL	3	6	4
TOTAL	25	44	31
AVERAGE	3.1	5.5	3.9

Figure 2 Analysis of entries

Front-end compression

The binary radix search technique described above will select one index entry for any given search key. However, the correspondence between the search key and the selected entry key is guaranteed only for those bits that were tested. To verify that the selected entry matches the search key, a full comparison must be made. This is better than the normal binary search because only one full comparison need be done.

The full comparison can be performed as a single step at the end of the search, or it can be performed in a piecewise fashion as the search progresses. The latter method was chosen because it provided other benefits as noted below.

At various stages along the search path, the next bit to be tested lies in a different byte than the prior one. At this point, all of the candidates have the same value for the completed bytes. By comparing each completed byte with the corresponding search key byte at the earliest possible stage, it is possible to terminate a search early when none of the candidates can match the search key. Terminating the search early can eliminate unnecessary page faults. Also, by storing the common text byte at the earliest stage in the decision tree, it can be eliminated from the termination text of each of the candidates. This can represent a substantial reduction in the amount of text required for many applications. This reduction is referred to as front-end compression because leading bytes of key fields need to be stored only once.

Page splitting

Even with front-end compression and a compact format for the structural elements, most decision trees require more than one page to represent all of the entries. This raises the question of how a tree should be distributed over several pages.

It should be noted that the logical structure of the tree is dependent only on the data. The number of entries in the tree and the average number of nodes along each path are generally related as follows:

Entries	Nodes
200	8
1000	10
5000	13
30000	15
1000000	20

Differences in individual path lengths are unimportant because the time to process a few nodes more or less is minor. But the time for a page fault is of more concern.

Accordingly, much effort is expended to provide a physical balance in the number of pages required to complete each path. The page-splitting algorithm allows any node in the logical tree to be replaced with an index page pointer; that page must then resume with the equivalent subtree. Most of the pages with termination text are placed at the outermost (that is, branch) level of the tree. The first page of the tree is called the trunk and will contain mostly pointers to other pages. Pages that contain pointers to other pages and are beyond the trunk level are called limbs. Table 1 gives examples of the number of pages at each level.

The splitting algorithm causes the pages in the level adjacent to the trunk to grow gradually and then drop to two when an additional level is needed. This is done because in a virtual paging environment the small number of pages in the level adjacent to the trunk has a high probability of remaining in main storage just through heavy usage. The fanout from a single page is typically in the range from 50 to 90.

Summary

Because of its extensive usage, indexing is a key element of the System/38. The index instructions provide fast access to the index entries by employing a binary radix search. Only a small number of pages are touched, and only a small number of nodes are processed. The indexing capability is independent of, but used by, data base and storage management. Thus, the index instructions provide an efficient and standardized set of building-block functions that are used by many system components.

Table 1 Examples of page hierarchies

Trunk	Limbs	Limbs	Branches	Total
1	–	–	80	81
1	2	–	150	153
1	12	–	1000	1013
1	80	–	6000	6081
1	3	200	14000	14204
1	8	500	32258	32767

References

1. R.E. French, R.W. Collins, and L.W. Loen, "System/38 machine storage management," page 63.
2. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.

User-System/38 interface design considerations

D.G. Harvey

Provides an overview of System/38's primary user interfaces. Describes objectives, constraints, and rationale.

The IBM System/38 is a new system providing extensive functional capabilities. One of the major architecture and design problems encountered in the development of this type of system is how this advanced function should be presented to the end user. Heavy customer investment in program and skill development precludes a radical diversion from current interfaces and approaches. At the same time, constraints imposed by contemporary facilities prevent a user from realizing the full potential of the new system. It is also deemed undesirable to burden new users with vestigial data processing constructs.

This paper gives some insight into how these classically opposing forces were reconciled to provide the System/38 user with interfacing mechanisms which are at once familiar and progressive. In so doing, the paper also provides a simple overview of the system's primary interfaces and their relationships.

Language/interface influences

The decision on when to extend existing interfaces versus providing new ones was driven mainly by consideration of four factors:

- New technical concepts
- New function
- New users
- Business investment

For purposes of this article, words like *current* and *existing* refer to IBM predecessors of System/38: System/3, System/32, System/34.

New technical concepts

A number of important technical concepts in System/38 have had major influence on the architecture of the user interfaces. Program/data independence, program/system independence, binding flexibility, transaction processing, and modular application structures have forced a function-language taxonomy that not only categorizes the languages/interfaces themselves, but also classifies function by interface. These five concepts are more or less dependent on the basic notion that applications, and even systems, consist of three major entities:

- Objects
- Programs
- Processing environment

Objects are those pieces of an application that "get worked on." They are the ultimate targets of all programming operations. Devices and data base files are perhaps the most common objects. The concept of the object is treated in detail by Pinnow et al [1].

Programs are those parts of an application that "work on" the objects. A program is also one of the more common objects since at any point in time it may serve as a target rather than a processor.

Processing environment is the conditions under which the application will run, that is, which programs will process which objects under what circumstances. Many aspects of the environment are often system dependent.

All five of the previously mentioned technical concepts flow from the theory that these three major parts may be, to a large extent, treated as independent entities. That is, objects within an application may be established and modified (redefined) while programs and environment remain constant; or programs may be established and modified with no effect on the others, etc. Also, given separate sets of objects, programs, and environments, it should be possible to statically or dynamically bind elements from each into a synergistic unit (the executable instance). Figure 1 illustrates this concept.

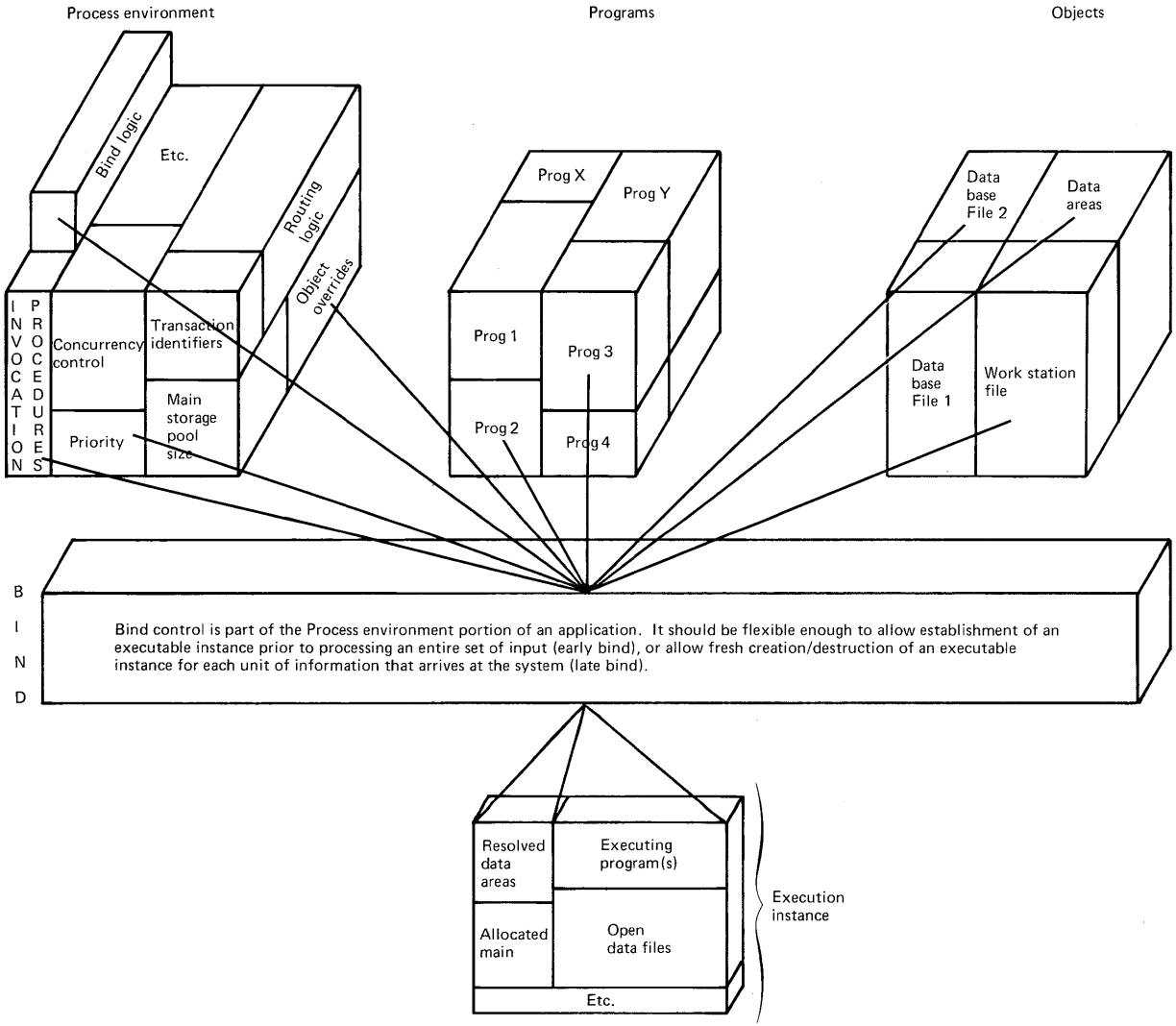


Figure 1 A method of combining elements from objects, programs, and environments

Given this fundamental trinity of application entities and an understanding of the benefits that can be gained by adhering to a clean separation of each, it became a major objective of System/38 to provide languages/interfaces that encourage the user to consciously delineate and define the separate parts. At the same time it was clearly recognized that there are enough instances of functional overlap among these parts that rigid enforcement of such a structure is neither desirable nor possible.

New function

Significant new function is provided on System/38, some of which is not readily adaptable to existing interfaces. The definition of data base and device files, for example, involves the building of structures and data relationships that are foreign to current user interfacing constructs. Other areas where new functions must be manifested through the user interfaces are:

- Work station I/O
- Data base I/O
- System display capabilities
- Object creation/manipulation facilities
- Test and service functions
- Transaction processing/system control functions
- Message handling facilities
- Save/restore
- Cross referencing/where used functions

It has been a primary objective of System/38 to not only give the end user access to these functions, but to also classify them such that they are available in the proper piece of an application. For example, some functions are clearly related only to the processing environment and need not be accessible from within programs. Many of them are system dependent in nature and should be rigidly isolated from the programming interface.

New users

Even a cursory analysis of this area reveals a growing set of users whose needs are fast outstripping current interfacing facilities. These users range from work station operators who need consistent mechanisms for effecting simple transactions to company managers who desire rapid access to their data base. Data entry, inquiry, and query functions (including hard copy generation) are required by users with little or no data processing training. Clearly, the final interface architecture had to take this user into account.

Business investment

Several straightforward business considerations affect the choice and definition of user interfaces on any new system. Gratuitous change from current approaches causes increased development costs for IBM, high reprogramming and education costs for our users, and, therefore, a justified resistance to the product. On the other hand, there are very real though less easily quantified costs resulting from strict adherence to dated practices. Unfortunately, the problem is further complicated by the fact that the optimum rate of innovation varies from business to business. Because of this, it was decided that System/38 interfaces must allow users to move onto new ground at their own rate. The user should be allowed to bring current applications to the new system without having to understand most of the new function available and without having to deal with radical new interfaces.

Results

After considering all the above factors, some conclusions were obvious. RPG was definitely the base on which to build the program definition interface. Customer investment in the language is high and, with the proper extensions, it provides a simple yet powerful means of manipulating System/38 objects. New file options and new calculation operation codes have been added which facilitate processing of work station and data base files and allow for much greater user control over logic flow within the program. This new RPG, designated RPG III, also allows the user to write a program such that objects like data base files, work station files, and external data areas are *not defined in the program.* I/O operations are specified in the program, but the object structures (down to the field level) are defined via a *separate interface.* Thus, RPG III meets the requirement that programs need not contain object definitions. It serves as the primary programming interface on System/38.

Again, looking at the major influences documented above, it was decided that no existing language could serve as the interface for establishing and manipulating the processing environment. Far too many new system functions are provided to even consider propagation and extension of the operator control language and utility interfaces now available. Most of this function is required interactively and is of an imperative "outside-in" nature, which fits nicely into a verb-noun command structure. Accordingly, a new Control Language (CL) interface is provided, which allows for the establishment and control of the processing environment and provides an interface to many system dependent functions.

The object definition interface could have been folded entirely into RPG III or CL. RPG III was rejected for this role because many of the objects are system dependent in nature and because the system would thus be architecturally constrained to a single high-level language programming interface. Also, for reasons previously stated, object definition should be isolated from the program. CL on the other hand is high-level language independent and is the intended interface for system dependencies. For simple objects (libraries, message queues, external data areas, etc.), it easily provides a simple and consistent definition/creation/deletion interface. Files, however, presented a problem. The definition of files, fields, structure, screens, etc., is largely a declarative process that historically has been supported through easy-to-use, fixed-form interfaces. Force fitting complex declarative structures into a language, which by definition is verb driven, would detract from the single statement-single action structure of the control language. It was decided that a separate forms interface should, therefore, be provided as a means of describing the more complex objects—work station and data base files. Appropriately, the interface is referred to as the data description specification, which is discussed by Truxal and Ridenour [2]. The data description specification serves only as a *descriptive* interface. In this regard, it is directly analogous to RPG III. Both are easy-to-use forms interfaces that describe complex objects (RPG III describes a program object). In all cases, actual object creation can only be effected via a CL command. While the description of simpler objects, such as libraries, is specified directly in create command parameters, the description of program and complex file objects is stored and referenced by name on the create command. Thus, the commands are simplified, and detailed data and program descriptions can be reused when necessary.

As illustrated in Figure 2, the three basic user interfaces on System/38 are the control language, RPG III, and data description specifications.

Users may choose to access System/38 function only through these interfaces; but in order to fully accommodate the non-DP types of users described previously, and those people requiring frequent use of control functions, it was decided that a higher level of interface was required—one that would allow these users to access data and major system functions without understanding any of the primary interfaces. As shown in Figure 3, this interactive ease-of-use facility consists of two major parts. One is simply a conversational means of entering CL commands; it

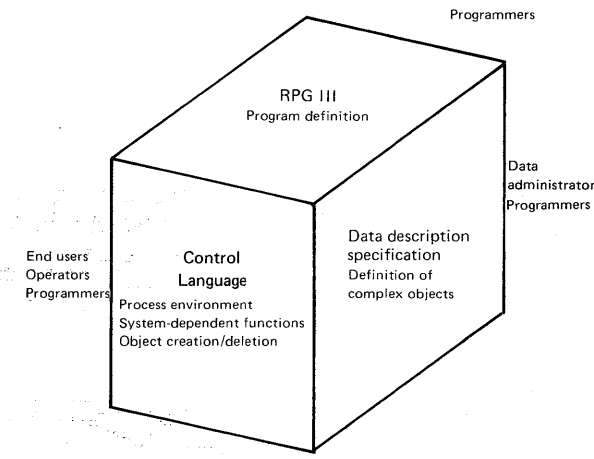


Figure 2 The three basic user interfaces in System/38

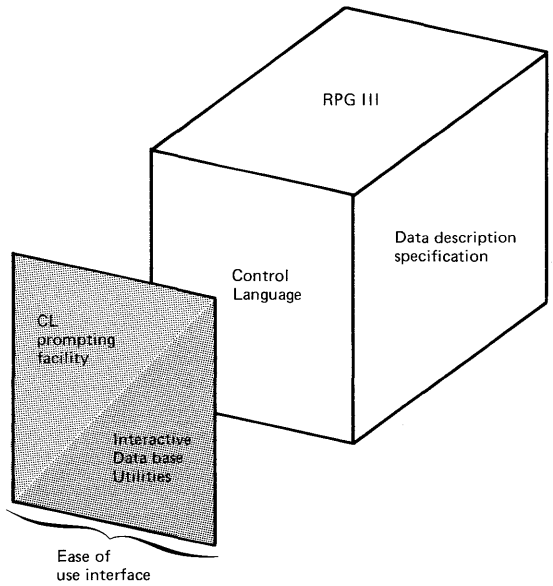


Figure 3 The high-level interface for accessing data and major system functions

interactively helps the user to enter the proper information for any chosen command function and then dynamically builds the command and causes it to be executed. Results are the same as if the user entered the command directly in CL syntax. The other is a powerful facility that allows the end user to define and execute typical work station applications without having to understand any of the primary interfaces. Source entry, data entry, inquiry, and query applications may be generated and executed with little or no DP training.

Summary

The user interfaces to System/38 reflect a balanced means of accessing function on a new generation of systems. User investment, new user roles, advanced technical concepts, and increased function were the primary factors influencing the interface architecture. They resulted in establishment of interfaces that support the new and advanced characteristics of the system while retaining a high degree of familiarity.

References

1. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.
2. C.D. Truxal and S.R. Ridenour, "File and data definition facilities in System/38," page 87.

Introduction to the System/38 Control Program Facility

D.G. Harvey

States that the System/38 Control Program Facility is a new high-function operating system product. Highlights key aspects of function, interfaces, and design objectives.

The System/38 Control Program Facility (CPF) is a totally new, high-function, operating system product designed to complement and extend the capabilities of the System/38 machine. As such, its functional offerings further emphasize the basic thrust of the System/38 architecture—fully integrated support for data base and work station-oriented applications. The advantages of real time, interactive access to functions and data are extended, through CPF facilities, not only to work station operators performing user business functions, but also to application programmers and system operations personnel as well. In addition, the CPF supports a wide range of batch processing options allowing increased control of batch work flow, full concurrency with interactive work, and submission of batch work from locally or remotely attached work stations.

To make its functions available in a consistent and easy-to-use manner, the CPF provides a single new control language and advanced new data definition and data manipulation interfaces.

This paper introduces the CPF by highlighting key aspects of its function, interfaces, and basic design objectives. It also provides the framework for related, but more detailed, CPF papers that follow.

Pervasive characteristics

In order to introduce significant advances in function, many previous systems have proliferated the number of specialized interfaces. This not only increased the apparent complexity of the system, it also greatly reduced the ease with which system users learned new functions and procedures.

To avoid such problems, the CPF has been designed to provide its wide range of functions through conceptually simple and consistent mechanisms. This simplicity and consistency is embodied in three basic CPF characteristics:

The interface to CPF functions is object-oriented. Each CPF "object" type can be characterized by its attributes, the operations that can be performed upon it, and the user data contained within it. Examples of CPF objects are: files, programs, message queues, user profiles, and libraries. Object concepts are discussed by Pinnow, et al [6].

Operations on objects are performed through a single new control language. The CPF Control Language (CL) consists of over 200 single-function commands. Command processing is available to work station and batch users alike. Commands can also be compiled to create control language programs. Key aspects of the Control Language are discussed by Botterill and Evans [2].

Function and object usage can be controlled through CPF security facilities. CPF supports both system level and user level security controls, as described by Berstis, et al [7]. System level controls can be used to prevent access to the system by unauthorized personnel. User level controls allow authorized users of the system to "own" objects and to grant and revoke object rights-of-use to other system users.

These simple but pervasive features make CPF functions potentially available to all CPF users through a single mechanism. The subset of functions needed by a particular user is not predefined or restricted by the CPF. It is determined solely by the needs and policies

of the System/38 installation. As users expand their use of CPF, they are not required to learn new or diverse interfacing techniques.

Workflow and resource management

One of the more novel aspects of the Control Program Facility is the support it provides for managing the flow of work and the usage of system resources [1]. Similar to some previous systems, CPF offers basic support for the concurrent execution of batch, interactive, and transaction-oriented applications. However, unlike previous systems, the CPF offers this support as an integrated feature of the base operating system product. This integrated design eliminates the need for unique resource management subsystems that impose their own operating overhead, functional restrictions, and special interfaces on system users. It replaces such subsystems with a subsystem concept of its own—the user-defined subsystem.

Integrating the subsystem concept into the basic operating system design allows the CPF to support any number of user-tailored operating environments through a single rule-driven mechanism. This technique not only makes the basic tuning functions of the System/38 machine available in all environments through a single interface, it also allows the user to isolate or intermix different types of work as appropriate without restricting in any way the options available to the application designer.

The rules for the operation of each subsystem are contained in an object referred to as a *subsystem description.* A subsystem description provides a means by which a user can prespecify sources from which new work is to be accepted (such as a job queue or a work station), programs to control the processing, and various resource usage parameters. Prespecification also provides a means for easy operational control since a subsystem can then be started with a single command.

In keeping with the terminology of previous systems, each "use" of the system is referred to as a *job.* Within a job, each processing *step* results in the initiation of a *process*—the basic unit of parallel execution within the System/38 machine [8]. The degree to which each processing step appears to be batch-like versus, for example, transaction-like is totally dependent upon the characteristics of the program(s) controlling the flow of execution within the step. The specifications in the subsystem description determine whether the stimulus for the next processing step causes control to be given to a designated user program or, for example, to the CPF command processor.

Command processing

The CPF command processor is designed to operate as a rule-driven subprogram [2]. As such, it can be invoked in virtually any environment to provide command services to CPF users. For work station users, basic command processing services have been extended to provide a powerful, interactive, command-entry facility complete with command selection aids, parameter prompting, and message reply capability. Through these facilities, CPF users have access to all commands and user applications (via the CALL command) to which they are authorized.

The rule for processing each command is contained in an object called a *command definition.*

Each command definition describes the validity checking, parameter defaults, and prompting text for its associated command as well as the program to be invoked to perform the command function. Because the command interface is maintained separately from the program that performs the function, new commands can be defined to tailor the interface to an existing program or to provide an interface to a new program (thereby extending the command set).

For often-repeated command sequences, a Control Language compiler is provided. Compiled CL offers not only the expected performance advantages but also a number of significant "programming language" functions not normally found in a control language, including:

- Declared program variables
- IF THEN ELSE and DO-group capabilities
- Work station I/O support
- Error monitoring facilities

These functional extensions (and others) make control language programs well-suited for performing the control flow portions of both batch and work station-oriented applications. The work station I/O support provided through compiled CL is specifically designed for simple menu and prompting usage.

Data management

The CPF control language provides a single interface through which object level functions can be performed. However, one type of object, the file, has special significance in that it is designed to provide a mechanism through which system users define and access large volumes of data.

As was the case on previous systems, the primary System/38 interfaces through which file processing is performed are provided by high level language compilers and utilities. CPF data management facilities are designed specifically to provide extensive file processing support for such products through a common file approach that encompasses both devices and data base. This support also includes a powerful new data description facility (*data description specification*) offering a fixed-form interface similar to RPG. This facility provides a system-wide mechanism for describing the format, relationships, and processing options for user data files, as described by Truxal and Ridenour [4]. Data description specification allows, for example, work station screen formats and logical views of physical data base records to be defined outside the programs that process the data. Much of the redundancy of record and field definition can be eliminated.

The file processing support offered by CPF extends to its users key System/38 machine features, such as concurrent data base access by multiple users and local/remote transparency for work station devices. In addition, it provides significant levels of both device and file independence including the ability to redirect sequential I/O between devices and the data base. Full input and output spooling facilities are also provided for printer, card, and diskette devices. File processing concepts are described by Fess and Benson [5].

Message handling

To facilitate communication throughout the system, the CPF supports the ability to send and receive messages via *message queues*, as discussed in detail by Demers [3]. A message queue is automatically created for each work station defined to the CPF. This allows work station users to communicate with each other by simply sending messages to the queue for the desired work station. Message queues can also be created for special uses such as a personalized "mailbox" or to allow program-to-program communication.

Individual messages can be predefined and stored in *message files* independent of any particular sender or receiver. This independence facilitates message text translation and allows CPF to provide support for default handling of conditions that cause messages to be sent. The CPF supports two levels of message text as well as data insertion within text.

The simple concepts of messages and message queues have also been integrated into other areas of the CPF architecture where generalized message handling functions are required. A single CPF component provides basic message-handling functions for all interface layers: machine to program, program to program, job to job, and job to user. Even the CPF's logging functions are supported through the use of this common facility.

Application development

To extend online services to programmers, the CPF control language provides a number of application development and maintenance functions:

- Interactive program testing support offers both a "test environment" capability that protects production data files against inadvertent modification and a dynamic debugging facility (e.g., no compile time specifications) that allows program breakpointing and tracing at *high level language* program statements and variable references.

- Library facilities provide those functions needed to group objects for convenient reference and to perform basic object operations such as locating, moving, renaming, and deleting.

- Extensive reporting facilities are also provided to allow work station or hard copy output of information such as object descriptions, file usage, spool file contents, and job status.

- A copy facility provides a means for copying data from one device or data base file to a new or existing file. Various record selection, record sequencing, and record formatting options are supported.

- The CL compiler and command definition facility provide the programmer with excellent tools for utilizing the full power of the control language and for extending or modifying the command set.

System management

A number of important functions are supported by CPF to aid in managing the overall operation of the system. Highlights of these functions include:

- A simple "no sysgen" installation procedure. CPF is shipped with a number of IBM-defined "user" objects including user profiles, subsystem descriptions, device files, and message queues. After installation, the system is ready to use. Any special tailoring of the system can be performed at any time through a variety of CL commands.

- Facilities for saving and restoring objects offline to or from diskettes. Save/restore functions can be used for producing backup copies of objects, for freeing auxiliary storage space, for application interchange between systems, or for removing data from the system so it can be physically secured. The CPF maintains a history of when and where each object was saved.

- A wide range of service functions intended primarily to aid IBM service personnel in analyzing, diagnosing, reporting, and fixing system problems. The majority of these functions are designed to operate concurrently with other system work to minimize impact on the overall system operation.

- Flexible console and system operator support. Full work station support is provided for users of the system's imbedded console. System operator functions are not, however, restricted to the console. Communication with the system operator is performed through a system operator message queue that is independent of any particular device. The system operator is, therefore, free to use any work station.

Summary

The System/38 Control Program Facility offers a wide range of function to both end-users of the system and other System/38 products. Its "integrated subsystem" approach to managing work flow coupled with the extensive resource management support of the System/38 machine provides system users with the flexibility and control needed to manage a variety of application environments. Its data management support extends and enhances machine device and data base support to provide a consistent and highly functional interface for data definition and file processing. Its single control language makes significant new function available that is not offered

through other interfaces. These functions and interfaces are provided through a design attuned to both the System/38 machine and the users of CPF.

References

1. H.T. Norton and T.R. Schwalen, "Table-driven work management interface in System/38," page 94.
2. J.H. Botterill and W.O. Evans, "The rule-driven Control Language in System/38," page 83.
3. R.A. Demers, "The generalized message handler in System/38," page 97.
4. C.D. Truxal and S.R. Ridenour, "File and data definition facilities in System/38," page 87.
5. R.O. Fess and F.E. Benson, "File processing in System/38," page 91.
6. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.
7. V.Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.
8. H.T. Norton, R.T. Turner, K.C. Hu, and D.G. Harvey, "System/38 work management concepts," page 81.

System/38 data base concepts

C.T. Watson, F.E. Benson, and P.T. Taylor

States that the System/38 data base support is unique in many respects. Describes major characteristics and their rationale.

The System/38 data base facility differs from existing data base packages in many basic ways. An associated article, "System/38 machine data base support" [1], explains that much of the System/38 data base function is supported by the machine and discusses the design decisions that were made regarding the level of function to be supported by the machine instruction set. The trade-offs covered in that article are performance, security, integrity, and ease of use. This article, on the other hand, describes some of the differences by giving an overview of the major functional characteristics. These characteristics include the System/38's design philosophy, its file structure, its sharing capabilities, its expandability, and its data manipulation capabilities.

Design philosophy

In past systems, full-function data base packages have been applications built on the machine's operating system and its file management component. This has always caused problems with security, integrity, and performance. System/38 is the first computer system to have a full-function data base facility designed as a part of the basic machine. The data base capability is a primary function of the Control Program Facility (CPF) and is comparable to data base systems previously available only as applications on more expensive machines. All online data in System/38 is stored, manipulated, and accessed through the data base component. The extensive capabilities of the data base facility are designed to be available to the user at whatever level of function and sophistication is needed. The security, integrity, and performance of the System/38 data base facility were enhanced by the consideration of the data base facility during the entire design of the system.

Files

All online data in System/38 is stored as records in data base files. A data base file has three primary attributes: the *format* of records within that file, the *access path* for the records within that file, and the *members* of that file [2]. A record has a fixed format defining each field and its attributes. The access path for a file defines the ordering of records within that file and provides for either random or sequential accessing of those records. The members of a file are different instances of data sharing the same file definition (the same format and access path definition).

The access path defines an ordering of records either by arrival sequence (order of insertion) or by key sequence. The keyed access path provides comprehensive ordering functions. A key exists for every record addressed by the file. A key is made up of fields from the record and system-generated character constants used to achieve hierarchical or duplicate key ordering. Each field can have ordering attributes applied to it: ascending, descending, absolute value, an alternate collating sequence, etc. Additionally, LIFO and FIFO duplicate-key ordering is specifiable for duplicate keys within a file. A selection feature is provided to allow the access path to address a subset of the records within a file based on field values within the record. Keyed access path maintenance represents some additional overhead on data base changes and, therefore, continuous or deferred maintenance options are provided. To ensure that the access path always represents the actual existing data, the keyed access paths are automatically recovered in case of system failure.

The two types of data base files on System/38 are the physical and logical files, as shown in Figure 1. Physical files represent the actual stored data. Logical files provide alternate user views of the stored data to support application and data independence and to avoid redundancy of data.

As shown in Figure 1, a physical file has one access path and one format. A logical file is over one or more physical files and has one access path and one or more formats. Members of physical/logical files (not shown) are different instances of data sharing the same access path and format definitions.

A physical file has one format; therefore, all records within members of that physical file have the same fields, attributes, and length. The records in each physical file member are ordered as defined in the access path for the file.

A logical file provides an alternate format and access path for one or more physical files. The logical file format allows a user to see a view of a physical record that subsets, reorders, or changes the attributes of the fields in the physical record. The logical file access path (1) allows the user to see a view of one or more physical file members, and (2) subsets, or reorders, the records in those physical members. This provides the user with an alternate ordering of the records for sequential retrieval, or, for random retrieval, a different key than that defined in the physical file. A different key definition can be specified for each physical file member addressed and any field in the format may be used. Figure 2 shows how a logical file can be used to provide a hierarchical view of one or more physical files. Notice that the three physical files, through use of a logical file, appear to be in one hierarchical file containing multiple record types.

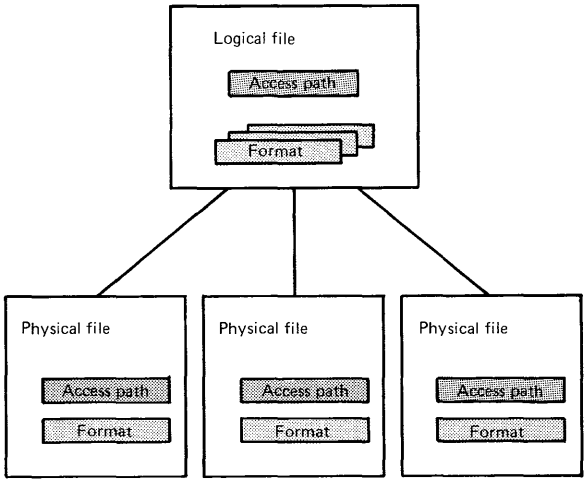


Figure 1 Physical/logical file structure

Another facet of this file-based design is security. Security in the data base facility is by file. When a user creates a file, he is the owner of that file and may specify public, private, or normal authorization. "Public" implies all users have all authorizations on the object. "Private" implies only the owner can currently use the object. "Normal" indicates that only the system default authorizations should be made public. The user may subsequently grant any authorities to selected (or all) users and may transfer ownership to another user [3]. To create a logical file, a user must have the correct authority on each of the physical files referenced. To use a logical file, sufficient authorization must be available for both the logical file and its physical files. Field-level security is supported through use of logical files.

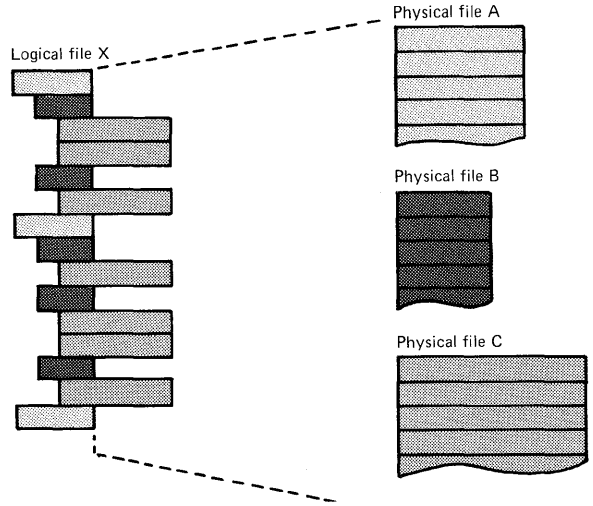


Figure 2 Hierarchical file organization achieved through use of a logical file

Sharing

Two of the main objectives in the System/38 data base facility design were to significantly increase sharability and decrease data redundancy. The main thrust of the physical/logical file structure is to achieve these ends. System/38 permits sharing on all levels. One of the major new features of System/38 is that file data definitions are specified to the system and shared. Programs may reference by name a central definition for file, format, and field attributes.

This means that the programmer supplies only the format name and then the compiler automatically retrieves from the system the record definition for use in the compiled program. The programmer may choose to ignore the defined format and may redefine field names or even redefine the entire record format, as was the standard procedure in all previous systems. The use of centralized data definition allows improved programmer productivity, fewer errors, improved file maintenance and growth capabilities, and provides the information needed for query, report generation operations, and field level prompting.

As noted in the "Files" section, the execution-time sharing of System/38 data is very powerful. Different user programs through different logical files see the same data in diverse ways. To achieve this, the System/38 data base facility makes changes visible to all users immediately, prevents users from updating the same record concurrently, and immediately reflects the change in all access paths.

Access paths themselves may also be shared. Another file type called a derived logical file allows different format definitions while sharing the access path with another file.

Expandability

Through use of the physical/logical file structure and the system-defined data capability, System/38 has made the user's data base flexible enough to meet the

changing needs of application programs and users. Below are several examples of this.

All file space expansion for additional record needs is handled automatically by the system unless the user specifies constraints.

New physical or logical files can be created at any time for use by new applications as shown in Figure 3. This has no effect on existing physical files, logical files, or application programs.

Modification of the attributes of a data file will not cause the program to need to be recompiled unless the new definition is not consistent with the program's usage. The affected programs are automatically notified at their next usage to indicate their need to be recompiled.

Data manipulation

The basic data base facility operations are OPEN, GET, PUT, UPDATE, DELETE, RELEASE, and CLOSE. OPEN and CLOSE connect and disconnect the file and the process. GET locates and/or reads a record from a file and has very powerful search operations. For arrival sequence access paths, the search can be sequential or direct. For a keyed access path, the search can be sequential, keyed, or direct.

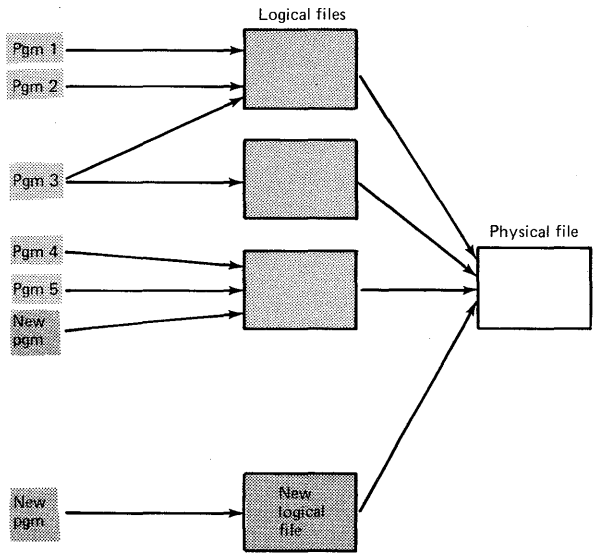


Figure 3 Meeting the users' changing needs

For sequential searches, the available options are first, last, next, previous, and same. Next and previous imply the record (as defined by the access path) next to the currently addressed record. Same is used when the previous GET was for position only, and now the record is desired.

For direct searches, the options are to find the n-th record in the arrival sequence or to find the record $\pm n$ records from the current position in the arrival sequence.

For keyed searches, GET can position to the record whose key is before, equal or before, equal, equal or after, or after the position indicated by the key supplied by the user. Or, GET can position to the record whose key is next or previous to the position indicated by a leading portion of the key for the record currently addressed, that is, a generic next or a generic previous.

On every GET operation, the arrival-sequence position number of the physical record addressed is returned so that the user can, later, quickly reposition to any previously addressed record. This is very useful for "navigating" through a data base structure and can be used to jump from the hierarchical structure defined by one logical file to a different hierarchical structure defined by another logical file, that is, parent to child and back to a different parent.

PUT inserts a record into a file. UPDATE modifies the contents of a record. DELETE removes a record from a file. All three operations immediately update all the keyed access paths to reflect the change. RELEASE unlocks a record that was locked by GET for modification or deletion.

Summary

The System/38 data base facility is unique because of its physical/logical file structure. It does not conform to the relational, hierarchical, or network data base model, but combines aspects of each. A physical file is similar to a relational file. The requirement that relationships between records (key values) must be stored values in the record is similar to that in the relational model. A logical file is a hierarchical file. The fact that multiple logical files can coexist and that the user can jump from one to the other provides function comparable with that provided by network models. In combining these features of the classical models in this unique way, System/38 has provided the user with an elegant, easy-to-understand data structure.

The System/38 data base facility has extensive capabilities and is a basic part of the design of the system. The data base facility was designed to greatly simplify the application programmer's job and to provide savings to the user in bringing new applications online and in maintaining existing ones. System/38 and its data base facility represent a significant step forward in bringing this level of function to the diverse System/38 customer set.

References

1. C.T. Watson and G.F. Aberle, "System/38 machine data base support," page 59.
2. C.D. Truxal and S.R. Ridenour, "File and data definition facilities in System/38," page 87.
3. V. Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.

States that the System/38 work management support is novel in many respects. Describes the objectives for work scheduling and how they were met.

System/38 work management concepts

H.T. Norton, R.T. Turner, K.C. Hu, and D.G. Harvey

The current trend of computer use places increasing emphasis on interactive processing and high levels of multiprogramming. In this environment, efficient management of the flow of work through and within the system is critical to the total capabilities provided and the level of performance. In addition, the work management functions must support a wide range of application approaches and usage environments. The combination of these requirements for function, flexibility, and efficiency presents many design problems in the area of work management. IBM System/38 addresses this critical area with a unique structure providing powerful and flexible work management functions with special design considerations for performance.

Management of the flow of work submitted through the System/38 Control Program Facility (CPF) licensed program [1] is provided by the high-level facilities of the System/38 instruction interface and specialized support provided with CPF. The System/38 machine provides a centralized mechanism for the dynamic management of processor and main storage contention. The specialized CPF support provides an interface [2] through which work can be submitted, presented to the System/38 for execution, and controlled by operators.

This paper describes the unique elements of the System/38 machine that relate to work flow management and the way these elements are used by the CPF work management support.

Jobs and processes

A *job* is the unit of work submitted by the user and recognized by the CPF work management support. A job may be a traditional batch job, an entire interactive session from sign-on to sign-off at a work station, or a spooling reader or writer. The existence of a job includes not only the time required to perform the processing but also any spooling operations accomplished in behalf of that job. For example, a batch job exists from the time it is read by the spooling reader, through its execution, and until any spooled output has been written (although the actual spooling is performed in other spooling jobs). The operator may issue job-related commands during the entire period of the existence of the job.

A *process* is the unit of work submitted to the System/38 machine and by which execution parallelism is managed. A process may be performing work as specified within a submitted job or may be a controlling process whose function is to manage the flow of submitted jobs. A job may be executed within a single process or may be executed within a series of processes whose initiation is managed by one or more of the controlling processes. In user terms, each new process initiation in behalf of a job is known as a new *routing step*. An executing job may also initiate asynchronous work, which in turn is processed as a separate job.

Process execution control parameters

As a process is initiated for execution within the system, a set of parameters is defined to control the execution of that process. Some of these parameters are similar to those supported on other systems; for example, the execution priority for the process and the processor time slice period. There are other parameters, however, that are unique to the System/38 and provide greater flexibility in resource management. These new control mechanisms are necessary for efficient operation in a dynamic multiprogramming environment.

A *storage pool* is a logical grouping of main storage. It is a quantity of main storage from which the dynamic requirements of processes assigned to that storage pool are satisfied. The user controls the number and

size of storage pools through CPF support. Inasmuch as the System/38 supports a uniformly addressable storage, there is much sharing of objects between executing processes (although facilities exist to minimize interference and handle contention between processes). If an object to be referenced exists anywhere in main storage, it may be referenced by a process without respect to its current storage pool assignment. However, if an object must be retrieved to main storage, it is brought into the storage pool to which the requesting process is assigned. In this way, processes attached to different storage pools do not interfere with one another in satisfying their requirements for objects not in main storage.

A *multiprogramming class* relates to the allocation of processor resource to various processes. It is a means of limiting the number of processes in a class that may be concurrently eligible for execution. This allows a general control over the extent of interference among processes that would otherwise compete for system resources. A process that would otherwise execute but exceeds the multiprogramming limit for its class is withheld from execution until some other process in that class completes to the end of a time slice or to some other execution wait point. For ease of use, CPF associates a multiprogramming class with each storage pool. The user views the multiprogramming level as belonging to the storage pool and can perceive the multiprogramming limit as relating to processes assigned to that storage pool.

Process execution objects

The "object orientation" [3] of the System/38 design requires several objects to support the actual execution of a process. These objects contain, for example, the static storage and automatic storage required by programs invoked in the process and other working storage areas used during the execution of the process. To make the initiation of a process more efficient, the design of the System/38 allows these objects to be precreated and reused for different processes.

Since several objects are required to support a process, a performance penalty would be incurred if each of the objects had to be separately accessed each time processing is initiated. The system, therefore, supports a special object known as an *access group.* An access group occupies one or more physical extents of auxiliary storage. Many objects may be created into the access group. Therefore, retrieval of the access group is more efficient than accessing each object independently.

A single access group may be associated with a process at the time a process is initiated. The transfer of such an access group between auxiliary storage and main storage is automatically accomplished by the system whenever a "long wait" in the execution of the process is encountered and another use of the main storage occupied thereby is anticipated. Examples of such long waits are waiting for a work station response, waiting for a message to arrive on a queue, and end of time slice.

CPF also creates other objects for each job that are used by CPF in the management of the processing for that job. The aggregate of all objects that are precreated for a job is called a *job structure.* This job structure is assigned to the job at the time it is selected for execution and is used for all processes that are serially initiated to perform the processing required for the job. By precreating this job structure, much of the overhead that would otherwise be involved in the initiation of processes is eliminated. In addition, continuity of job-related information from one process to the next is easily supported.

Summary

The work management facilities provided within the System/38 machine provide a strong base on which an effective work flow control can be built. The CPF capabilities complement the machine facilities, supporting a user interface that gives to the user the ability to apply meaningful, effective controls over the flow of work within the system. The controls are general enough to enable the system to efficiently manage the instant-by-instant flow of work, but are flexible enough to cover the broad range of operating environments envisioned for the System/38.

References

1. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
2. H.T. Norton and T.R. Schwalen, "Table-driven work management interface in System/38," page 94.
3. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.

States that System/38 has a single, flexible control language that works in conjunction with an authorization facility. Describes the implementation approach and rationale.

The rule-driven Control Language in System/38

J.H. Botterill and W.O. Evans

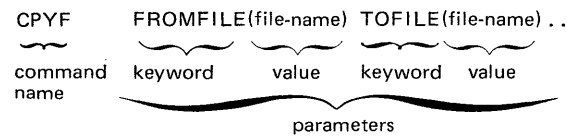
A work station-oriented system such as the IBM System/38 must make its functions available to different types of users in a manner that can be readily used. For example, users need to be able to request standard system functions from work stations, from the system console, and from programs without limiting commands to particular devices or environments [1].

A single, flexible, control language coupled with an authorization capability satisfies these requirements, but is difficult to design and implement. An associated article [2], discusses the way the authorization capability is supported in System/38.

This article describes how a rule-driven approach made this type of control language possible on System/38 and the significant benefits that result. This rule-driven approach produced a single set of commands with a common syntax, the same level of validity checking, and the same type of work station prompting. The restrictions and multiple sets of unlike commands found on previous systems were thus eliminated. In addition, the capability exists for easy modification and extension of the commands.

Control language structure

The basic syntax for the System/38 control language (CL) is a simple, free-form syntax, using blank as the separator. The command name and parameters can begin anywhere on the record, allowing indentation and parameter alignment. Each parameter has a keyword that can be used to identify the parameter value. The keywords may be omitted if the parameter values are entered in a fixed order. For example, a Copy File command could have the form:



A file A could be requested to be copied to file B in either of the following ways:

```
CPYF  FROMFILE(A)   TOFILE(B)
```

or

```
CPYF  A   B
```

The detailed description of each individual command and its supported parameters are stored in a command rule, which is the basis for the rule-driven approach.

Rule-driven approach

Basic to the rule-driven approach is the recognition that each command is simply an interface to a function. The processing of a command consists of

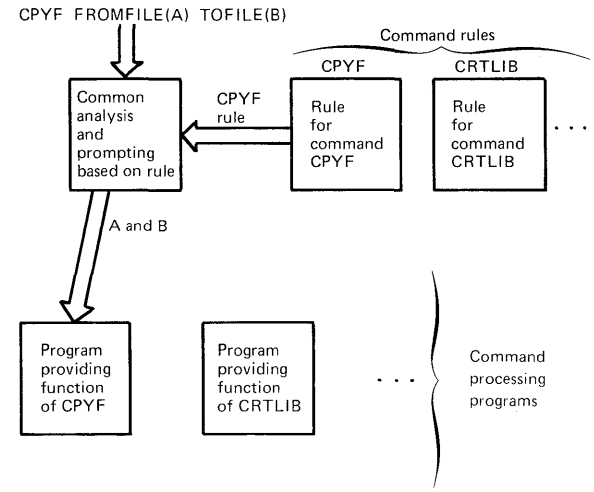


Figure 1 Processing a command in System/38

analyzing the command based on the rule and providing the function, as shown in Figure 1.

Even though the functions of two commands may be very different (e.g., the functions of the Copy File command and a Delete Program command), the user expects the manner of specification to be much the same. This includes things like command name specification, similar error messages for similar errors, and a standard defaulting technique for parameters. Such similarity is achieved by having the checking and analysis done by a common command analyzer.

Most values used as command parameters fall into a few well defined data types and subtypes such as:

Character: Any string, name, date, time
Decimal
Logical

Therefore, a function was implemented on System/38 to define a rule for a command. The rule, which is called a *command definition*, simply defines, for a given command:

Command name
Each parameter on the command
Program to process the command
Where valid: Interactive and/or compiled

For each parameter, the rule defines the type of input to accept and other attributes that can logically be centralized to the common command analysis. Specifically, the following can be specified for each parameter:

Keyword name
Type
Validity checking
Default to be used if parameter is not specified
Prompt text
Constant value in place of user-specified value

The command name is the identifier of the rule object that represents the command on the system. The rule is used to control the common control language validity checking and command prompting. The only unique specifications for a given command are the command's rule and the program to provide the function.

The following paragraphs discuss the major benefits achieved on System/38 through this rule-driven approach.

Validity checking

By specifying the desired validity checking in the command rule, the CL validity checker ensures that:

- Required parameter values are present
- Data type and length requirements are met
- Conflicting parameters are not entered
- Where appropriate, the value of a parameter:
  - is one of a list of valid values
  - falls within a numeric range
  - satisfies a required relationship with other parameter values on the same command

By separating the validity checking from the processing for the command, consistent validity checking is provided at all of the following key times:

Source entry into a data base file
Batch job spooling
Compilation of a CL program
Command execution
Interactive prompting

This means that on System/38 all values can be checked at entry into the system, greatly improving problem feedback and thus reducing processing steps based on erroneous input.

Parameter prompting

For work station entry of commands, a command prompter identifies parameters, defaults, and valid values for the user so that he can enter commands without frequent reference to publications. All the information necessary for this assistance is obtained from the command rule.

This prompting can be requested at any time during command entry, independent of how much of a command has been keyed or whether the command has been requested through a command selection menu. The assistance is designed to fully utilize the display work station by presenting multiple parameter text descriptions on the same display along with corresponding input areas. Figure 2 shows a display screen prompting for six parameters of the Copy File command.

The input areas are of the length of the values allowed and contain the meaningfully labeled defaults. The CRTFILE parameter is shown in the example with a default of *NO. That value can be accepted or keyed over with the other valid value of *YES. The user can quickly review the command parameters and their defaults and key only the values that are out of the ordinary. Values are entered without specifying parameter names or the need to adhere to special positional or syntactical requirements.

After the values have been keyed, they are validity-

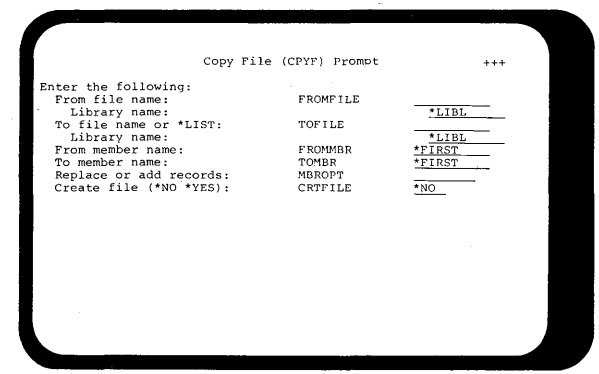


Figure 2 Example of command parameter prompting

checked and immediate feedback is given. This process continues until the user and the system agree that the command is ready for execution or entry into a source file.

Parameter defaulting

The System/38 control language utilizes a new highly visible defaulting approach. All parameters with a default have a corresponding specifiable word value that is descriptive of the default; for example, *NONE for an authorization default, or *NOMAX for a file size default. Such self-descriptive values are defined in the command rule, documented in the command reference publication, shown on the prompt display, and optionally coded when the command is entered. If the default is coded, the same action is performed as if a value were not coded. The specification of the value identifies to any person reviewing the program or command log what value is used. A value may be coded without concern for whether or not it is the default.

This approach to defaulting removes the ambiguity of defaults, thus allowing them to be freely used to simplify coding and keying. A default is simply a valid value for the parameter that is identified as a default. This default can be changed by simply changing the command's rule definition.

Advanced source entry support

The separation of validity checking, prompting, and defaulting from the command processing program allows these functions to be performed at source entry time. Thus the standard command validity checking validates all parameter values as they are entered rather than waiting until compile or execution time.

In addition, the same prompting support is available to the source entry user as to the operator entering commands interactively. This means that the user must learn only one interface and has the same advanced function for both types of command entry.

Customer tailorable command set

A rule-driven command set results in an easily tailored interface. In order to subset, simplify, or change the terminology of a command or commands, a user can change the command definition rule and create his own command to invoke the system-supplied function.

Besides changing keywords and defaults, the user can set parameters to a constant value. Thus, a parameter that otherwise would always be specified in the same way can be removed from the command. Therefore, the command prompter will not prompt for that parameter, nor can it be specified. For example, a PRINT parameter on a Copy File command that accepts the value *NONE for "do not print any of the records" or *COPIED for "print the copied records" could be fixed to the value *COPIED. Thus, the command prompter would no longer prompt for the PRINT parameter and it could no longer be specified. The following command would no longer be valid:

```
CPYF FROMFILE(file-name)    TOFILE(file-name) +
     PRINT(*NONE)
```

but instead

```
CPYF FROMFILE(A) TOFILE(B)
```

would always mean copy the file and print the data copied (i.e., *COPIED). The print option would not be offered during prompting and could not be specified.

User command definition

A command is provided to allow a user to define commands to invoke his own programs. This allows the user to have the full benefit of the system-parameter validity-checking, prompting, and defaulting facilities when invoking installation-provided functions. Such user-defined commands can invoke application-oriented functions or user-defined system-related functions. In this way, the rule-driven approach results in the customer being able to extend the command set.

Development benefits

The handling of all validity checking during the common command analysis results in reduced development costs. It removes the need for validity-checking code within each of the command processing programs. This results in more single-function, command-processing programs that can be more easily and thoroughly tested. The validity-checking code only needs to be tested in one place. These factors result in greater system reliability and consistency of system operation.

In addition to this development economy, the rule-driven approach gives the flexibility to improve the ease of use and consistency of the command set during development without affecting the programs that provide the functions. This allows the development of a well designed command set that has been improved through use.

Summary

The rule-driven approach to command processing on System/38 utilizes a command rule that defines a command to a centralized set of system programs. These programs support common command-processing functions.

The approach made possible a single control language for work station, compiled program, and batch

environments. In so doing, it results in many benefits to its varied users. These benefits include:

- Consistency of validity checking, parameter prompting, parameter defaulting
- Extendability
- Tailorability
- Reliability

All this was accomplished with greater economy of development cost because of the centralization.

References

1. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
2. V. Berstis, C.D. Truxal, and J.G. Ranweiler, "System/38 addressing and authorization," page 51.

Discusses System/38 files, their definition, and the advantages realized from system-wide use of a central data description facility. States that the System/38 central data description facility is a significant innovation.

File and data definition facilities in System/38

C.D. Truxal and S.R. Ridenour

The IBM System/38 provides a high level of functional capability and incorporates many novel technical approaches. One of its most significant innovations is the unified and cohesive approach taken in providing function. This is particularly true in the area of data management support in the Control Program Facility (CPF).

The principal purpose of a computer system is to process data. The data enters and leaves the system through devices and is stored on a variety of media, both online and offline. In the past, each program (utility, data base, etc.) has required a separate definition of the data, and the definitions frequently differed. This produced redundant data descriptions and sometimes created program incompatibilities making inter-language file transfer impractical.

System/38 has an online data base, extensive device support, and a wide range of utilities and program products. To meet system design objectives for ease of use and file independence, one consistent system-wide method of data definition was required. The result is the *data description specification* facility and the supporting data base and command functions. The data description specification provides the user interface for both data base and device file creation. This paper discusses System/38 files, their definition, and the advantages realized from system-wide use of a central data description facility. An associated paper, "File processing in System/38" [1], describes some of the data management concepts used in processing files defined by the data description specification.

Files

Data base files reside in the system's uniformly addressable storage and provide to the high-level language user either keyed or arrival-sequence access to online files. *Device files* are also resident in the system and define the user's interface to I/O devices, such as work stations (terminals) and printers. Throughout System/38, files are treated in a consistent way. That is, all files are created through the data description specification, file definitions are displayed by the same commands, and programs which are compiled referencing one file can be dynamically redirected at execution time to use another. The level of device independence in the system even permits the interchange of data base and device files.

Data description

All data base, printer, and work station data on System/38 can be described at a *field* level, with field attributes such as data type and length. Options exist which allow specification of such things as descriptive text and field validation parameters for each field.

The normal unit of data transferred by a program is a *record*, which is made up of one or more associated fields. To define a record on the system, descriptions of the fields comprising the record are grouped and are called a *record format.* Logical data base files and printer and work station files can have more than one record format and, to create a file, a command is executed with record formats and file attributes specified as *file description* input. This input is specified for both data base and device files using a common syntax. A data description specification coding form and an interactive utility are provided for creating the file description input. The syntax provides fixed columns for frequently specified or required information and keyword specifications for less frequently specified options.

The data description specification is input to a compiler-like function, operating on file description information and creating a file. This file not only provides access to data but can also maintain the description of that data. This description is available to all subsequent users of the file, as shown in Figure 1, and

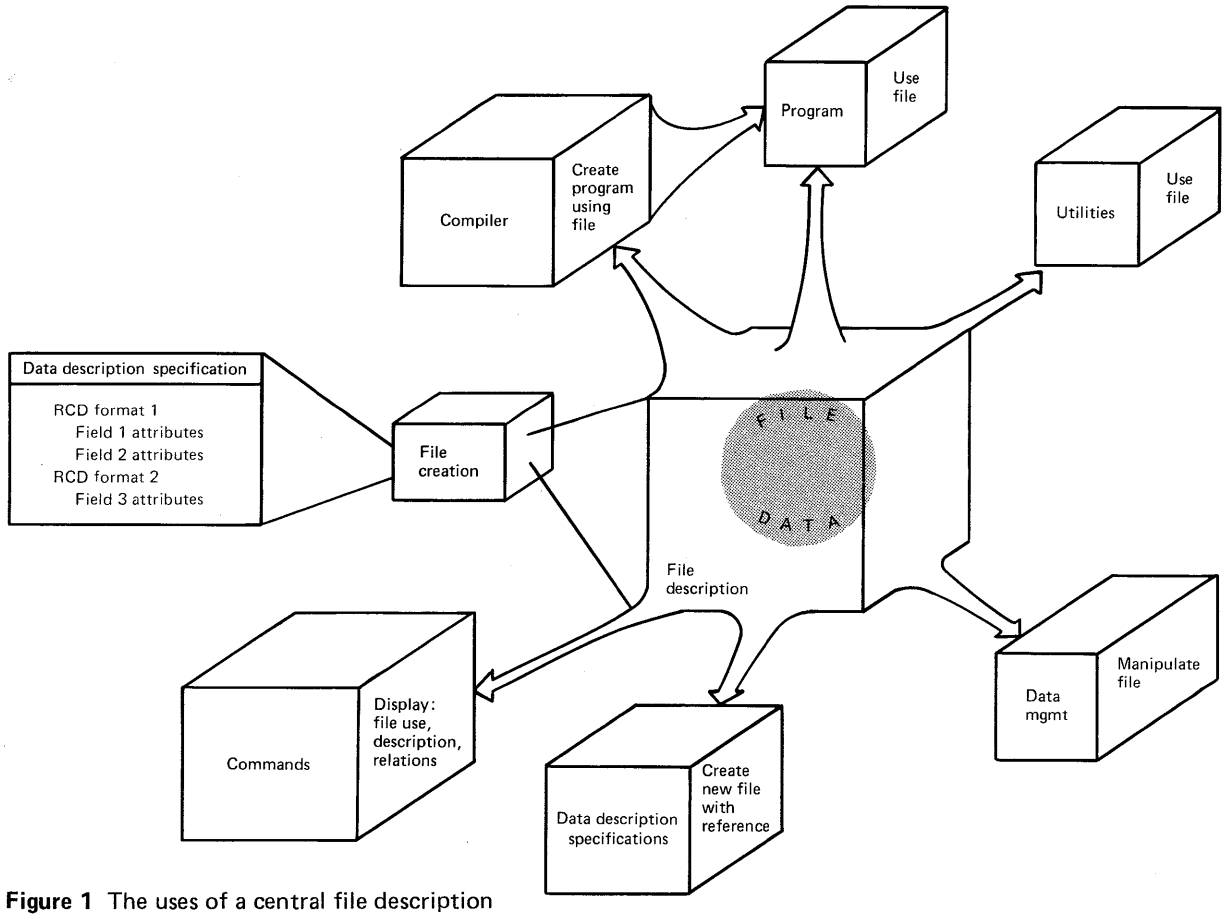


Figure 1 The uses of a central file description

to anyone defining similar files. It is used by compilers, utilities, commands, user programs, data management file processing interfaces, and the data description specification facility. When defining a new file, existing data base files can be referenced for field attributes, and data base record formats can be shared.

Once a file is created, it may be manipulated through the use of system commands, utilities, program products, or user programs.

Programming considerations

For programming on System/38, two approaches are available to describe file data to be processed by the program. The first is the conventional approach, where the structure of the data is described in the program itself. In this case, the file is created with its records described as one data string. Because the file definition occurs within the program, this is termed *program-described* data. Since no field attributes are defined, the system provides no validity checking, transformational functions, or device formatting. This type of function must be performed by the program.

This approach to data definition is provided mainly to facilitate program conversion to System/38.

The alternative approach can be used to take advantage of full data management function, to increase programmer productivity, and to decrease the probability of error in describing data. Files are specified as *externally described*, and the centralized data description specification file description is used by the compiler. This not only increases programming productivity but also significantly improves program/data integrity. This is because there is a "level" associated with the system file description which changes when the description is changed. If the record format of a file changes and a program accesses it with an "old" format, a warning is automatically issued that the program is down level.

Data documentation and interactive ease of use are two more programming areas enhanced by a centralized file description. When a file is externally described, a text description can be associated with each field. The compiler uses each field's text description to generate program comments. Interactive applications can use similar descriptive text at the work station to provide meaningful prompts on a field basis. The System/38 interactive utilities do this on file maintenance and inquiry functions.

The System/38 data management facility performs functions for the programmer which are requested at file creation through the data description specification. The functions available vary depending on the type of file. There are data base and device file-specific parameters to match these functions. A brief description of some of the device and data base functions follows.

Device file functions

There are many parameters which simplify programming interfaces and facilitate work station usage.

These attributes are used to specify the exact field location on a screen or page, automatically display prompt information, underline/highlight specific fields, specify automatic input field validation, indicate whether an input field is required or not, etc. When a program is interacting with a work station using field-level definitions, an entire screen (or multiple screens) can be input or output with one I/O request. Similarly, an entire page of constant and variable information can be formatted and printed with one output request. If desired, only changed fields are entered from a work station, with indicators set to show field status. These indicators and related conditions are options specified as data description specification input at device file creation.

Data base file functions

Both *physical* and *logical* data base files are created using the data description specification. The structural difference between these files is that physical files actually contain data and are limited to one record format per file. Logical files provide alternate logical views and ordering of the data contained in physical files. Related records are grouped into *members* within a file, providing a partitioned or generation-like data structure. For example, a file containing annual business orders might consist of twelve members, each containing orders for one month.

Through the data description specification, the user can define several record formats in one logical file and different key field specifications for each format. Key fields can have a "unique" requirement or not, and, if duplicate key values are allowed, the retrieval ordering of duplicates can be specified. A logical file can "interleave" records from several physical files, giving the illusion of variable-length records ordered by key in one file.

For example, a logical file might be defined over three physical files: a header, a detail, and a trailer record file. The logical file can be processed randomly by key or sequentially, and will be ordered with each header record followed by all associated detail records and the associated trailer record, as indicated in Figure 2.

File data is shared by physical and logical files. Record sequencing information, called *access paths*, and record formats can also be shared to improve both space utilization and execution performance. Each of these sharing relationships is specified through the data description specification at file creation.

File information

Data base file relationships, device and data base file attributes, and record format specifications are file-related information that is available through the use of a System/38 command. A report on program use of files and record formats is similarly available, on display, printed, or in a data base file for program analysis or cross-referencing. All of this information is maintained dynamically on a file basis and always reflects the current file status.

Summary

The System/38 data description facility in CPF provides a centralized file definition capability for both the system's integrated data base files and device files. A file's definition can be at the field level, and contains field attributes, descriptive text, and other field level specifications.

This definition is associated directly with the file for its life, and is used by compilers, utilities, and data management file processing functions.

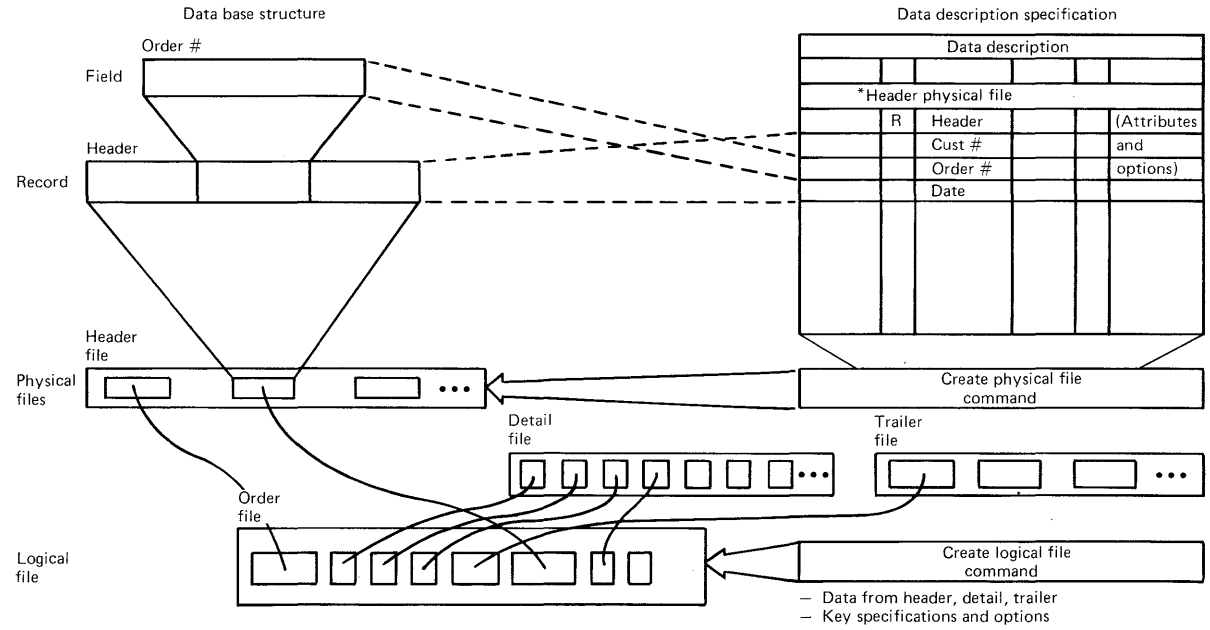


Figure 2 Relationship of the data base structure to the data description specification and creation interface

This facility, when combined with the functions provided by data base and device support, provides the user with a consistent and powerful approach to file processing.

References

1. R.O. Fess and F.E. Benson, "File processing in System/38," page 91.

Describes several System/38 file processing functions. Included are opening files, binding files to programs, and file independence.

File processing in System/38

R.O. Fess and F.E. Benson

The main function of data management on the IBM System/38 is to assist users in managing the data processed by their programs. Data management is that part of the Control Program Facility (CPF) licensed program that provides the means for formatting data into records, for organizing data records into files, and for transferring the data records of a file between a program and the file.

The file is the central structure used to access data from the data base or devices attached to the system. All files created through the CPF data description specification facility consist of two parts: the file description and the file data. An associated paper, "File and data definition facilities in System/38" [1], describes data base and device file creation. This article describes several key functions provided by the data management file processing interfaces. The functions discussed are file open (*open data paths*), late binding of files to the program (*file overrides*), and file independent operations.

Open data paths

Before any file can be manipulated by a program, it must be identified by the program and the intended use specified. This is done through the file open interface. Identifying a file also identifies the open data path (ODP).

An ODP is the internal structure that data management uses to connect a program to a file such that the file data can be accessed through the various data management interfaces.

The ODP is used by the system to contain all the file status information necessary for the use of the file by a given user. Through the implementation of ODPs, the system has achieved a high level of file independence, the capability for users to share files concurrently, and the ability for multiple programs in a single job step to share a file's status, position, and buffers. The implementation of ODPs also minimizes the execution time processing required when a file is opened. To understand how this is achieved, it is necessary to understand the ODP.

There are three types of ODPs: prototype, active, and inactive. A prototype ODP is created when a device file or data base file member is created. Thus, for each group of data records there is a prototype ODP that is part of the permanent file structure. The inputs used to construct the prototype ODP come from parameters on the Create/Add commands, the file specifications, and internal data management information based on file type. The active ODP is created when a file is opened. The inputs to creating an active ODP are: the prototype ODP, open parameters specified in the program, and parameters specified on a File Override command. Active ODPs are temporary objects and exist only while the file is open. Active ODPs become inactive ODPs when a file is temporarily closed.

The prototype ODP is a pre-open path to the data that contains the base set of attributes to be processed when the file is opened. It contains the initialized linkage to the data management I/O interfaces supported for this file type (data base, card, display, printer, etc.), the allocation list for the file, where the data is to be retrieved or output, and initial open and I/O feedback information. The prototype ODP also contains device-initialization parameters such as print image and lines per inch; common file parameters such as share ODP and secure ODP from overrides, and file-dependent parameters such as overflow line number, force ratio, and hopper number. These parameters provide all the required information to open the file.

Prototype ODPs are created to provide a fast open for the file, since they are created when the file is defined. The repeated processing of the same param-

eters, initialization of allocation lists, linkages to data management I/O interfaces, etc., are eliminated. The prototype ODP also allows a program to open a file by specifying only the file name.

File processing

As is true for other systems, OPEN is the process of connecting the program to the file such that the data can be processed. OPEN processes the file name specified in the program against any *file overrides* that the user may have entered external to the program. A file override allows execution time alteration of file attributes and the late binding capability of a file to a program. This capability will be discussed.

When OPEN has determined the file to be processed, a copy of the prototype ODP is made and parameters from the program and file overrides are merged with the file attributes residing in the ODP. The order of the merge is (1) program-open parameters override the parameters in the prototype ODP, and (2) any file-override parameters override both the program OPEN and prototype ODP parameters. This allows the most recent request to take precedence over older parameters and file attributes. After the copy of the prototype ODP is updated, OPEN:

- Allocates the file to assure the users access to the file description and file data. For data base files, the file description, data space index, and data spaces are allocated. For device files, the file description and the device are allocated.
- Creates the buffers for the user to receive the input data or for the user to place the output data.
- Constructs the I/O request blocks used by the machine to communicate with the data base or device.
- Activates the cursor or device so that data can be transmitted between the program and the data base or device.
- Initializes the device to process the data.

On completion of OPEN, there is an active ODP and the user program can address the open feedback information and input and/or output buffers. This is through a program object called the user file control block (UFCB). The open processing for both data base and device files is similar in that a prototype ODP is copied and updated with file overrides, the file is allocated, and the connection to the program is the same as shown in Figure 1.

The data from the file is retrieved via a GET interface and output to the file via a PUT interface. There are also other I/O interfaces provided by the CPF to take advantage of the unique operations allowed on each file type or device. Examples of this are the UPDATE and DELETE interfaces for data base files.

As is the case in other systems, CLOSE disconnects the file from the program: CLOSE deallocates the file, deactivates the device or cursor, and destroys the active ODP.

File overrides

File overrides allow the late binding of files and file attributes to a program. They are applied when the file is opened and permit the file attributes and/or the file to be changed at program execution time. With file overrides, the user can:

- Change, at program execution, the file-open parameters specified in the program, the attributes specified in the file description, and/or the file to be processed.
- Apply a single file override to a file that is used in one or more programs.
- Enter file overrides in batch jobs, interactive jobs, and CL programs, or invoke them from a high-level language interface.
- Control the files in the program to which file overrides can be applied.

There are two points of view for file overrides as they apply to a program. One is the programmer's (designer/coder) and the other is the user's (caller/invoker). The programmer must know what files are

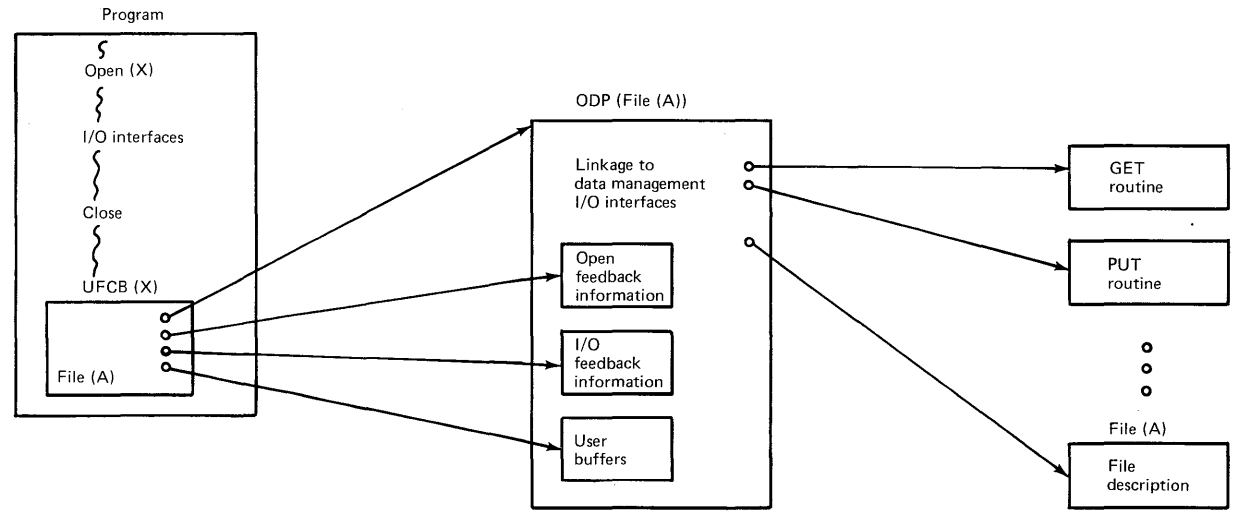


Figure 1 Opened file structure

to be processed and whether overrides are to be allowed for the files. The user must know the files accessed by the program and which files can be changed via a file override. Each program is treated as a "black box," and the user does not know or care if the program contains a file override. All the user of a program knows is that the program processes 'File A' with attributes a(1), a(2), . . ., a(n). These attributes could be specified in the file, in the program, or on one or more file-override commands.

The file overrides are applied at OPEN and can be explicitly or implicitly deleted. A file override exists for the duration of the program that invoked the command unless explicitly deleted. At open time, a file name specified in the program is used to search for any file overrides that apply. The file overrides are applied from the innermost file override to the outermost file override as shown in Figure 2.

```
1  OVRPRTF    FILE (REPORTS)      COPIES (6)
2  CALL PGM (WRTRPTS)
       Program-WRTRPTS
3  OVRPRTF    FILE (OUTR) TO FILE (REPORTS) +
                 COPIES (2)  LPI (6)  FORM TYPE (3760)
     •
     •
     •
     CALL PGM (RPTW)
        Program-RPTW
          •
          •
          •
4       OPEN FILE (OUTR)
          •
          •
          •
        End program-RPTW
     End Program-WRTRPTS
```

Figure 2 An example of how a file override is applied

In the example shown in Figure 2, the file overrides from Statements 1 and 3 are applied when the file OUTR is opened in Statement 4. The result is that the file REPORTS is opened with attributes COPIES(6) from Statement 1, and LPI(6) (lines per inch) and FORMTYPE (3760) from Statement 3; the rest of the parameters will be taken from the file.

Through the file-override command, data management supports changing file parameters at program execution time and file independence.

File independence

File independence is provided because the file and/or file type can be changed but the program still receives or outputs the desired data in the proper record format. The highest level of file independence is when the file and file type can be changed with no impact to the inputs received or outputs produced by the program. A program can achieve this level of file independence by using the file-independent operations that are common to all file types. These are the normal OPEN, CLOSE, GET, and PUT interfaces with no file-dependent parameters specified. The processing performed by the program is to read sequentially through the file or write sequentially to the file.

The second level of file independence is when the file can be changed but the file type cannot. In this case, the program specifies file-dependent parameters on the GET and PUT interfaces or uses a file-dependent interface such as UPDATE (a record). The desired input records or output records will not be produced unless a particular file type is used and the record formats that are specified in the program match those of the file.

The third level of file independence is when the file and file type can be changed and the user is willing to accept slight differences in the output produced. An example of this is overriding a printer output file to a card output file. The program will function correctly so long as a file-dependent interface is not used.

Summary

On System/38, the file is the central structure used to access the data from a data base or device file. The prototype ODP is created along with a data base or device file. The prototype ODP is the pre-open structure for the file that is used by OPEN to connect the program to a file. When a file is opened, file overrides are applied to alter parameters specified in the program or in the file.

Different levels of file independence are provided by file overrides in conjunction with file-independent operations.

References

1. C.D. Truxal and S.R. Ridenour, "File and data definition facilities in System/38," page 87.

Table-driven work management interface in System/38

H.T. Norton and T.R. Schwalen

States that work flow management is table-driven by system values and definitional objects, and that default settings may be used or new ones entered.

The IBM System/38 is designed to satisfy a wide range of application environments ranging from purely batch work to predominantly interactive work, from fast response time requirements to high throughput requirements, from program-driven application approaches to transaction-driven approaches, and so forth. This wide range of environments requires powerful and flexible work management functions. However, any particular user will normally deal with only a narrow set of the work management capabilities. Thus, the challenge of the work management design is to provide an interface through the control language that is straightforward and easy to use, yet allows access to a wide range of function in an easily modifiable fashion.

This paper discusses how these goals for managing work are met in System/38 through a table-driven interface approach. An associated paper, "System/38 work management concepts" [1], describes the actual work management and functions invoked through this interface.

Figure 1 illustrates the structured interface that is described in this paper.

System-level controls

The system level is the highest level of control for managing work flow. There are few operator controls at this level. The system may be started through the operator console and may be terminated by means of a command. Some global options are available to the system operator during the starting of the system through *system value* parameters. Using these system values, the installation may prescribe parameters such as the maximum system-wide level of multiprogramming, a switch value indicating whether the system is to be brought up in an unattended mode if the console is not operational, and the format to be used in presenting the date on displays.

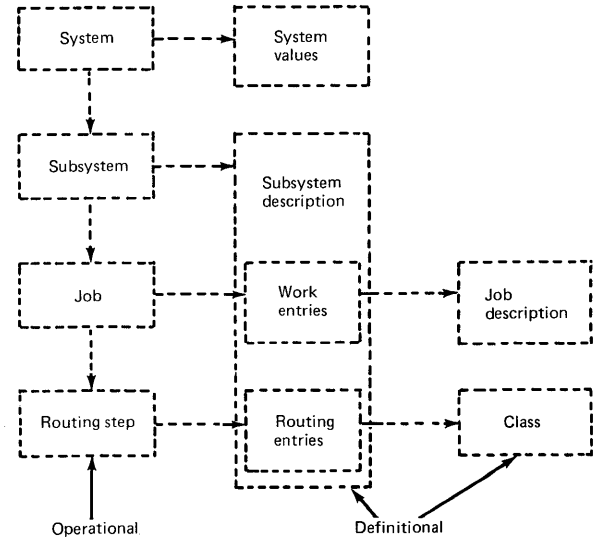


Figure 1 Structure of CPF interface for work flow management

Subsystem-level controls

The key to the capability, flexibility, and ease of use of the work flow controls provided through the Control Program Facility (CPF) [2] is the *subsystem.* A subsystem is the means by which an operational environment can be defined and controlled. By allowing multiple subsystems to be active concurrently, the installation has extensive flexibility in defining and controlling the work being done on the system.

The definition of the operational environment for a subsystem is contained in a CPF object called a *subsystem description.* (The concept of "object" is discussed by Pinnow, et al [3].) This makes it possible for a single CPF program to control all types of environments, although a separate invocation of this program is established for each subsystem that is

started. It is through the starting and terminating of subsystems that the subsystem-defined environments become active or inactive. Thus, the operational management of these environments is simply to start and terminate the desired subsystems at the appropriate times.

The number of subsystems that an installation chooses to define or to have concurrently active depends on a number of factors, such as:

- The number of unique processing environments that are needed to effectively manage the usage of resources by various applications
- The degree of operational control that is desired over the set of applications defined within the various subsystem descriptions
- The extent of isolation the installation chooses to impose between the sets of applications defined within the different subsystem descriptions

The advantages provided by the implementation of subsystems in this manner include:

- The entire processing environment for a set of applications can be prespecified and easily modified as needed to meet the changing needs of the installation.
- Operational control of the entire operating environment of a subsystem is easily accomplished.
- The use of main storage and the allowed level of execution concurrency for jobs executing within the subsystem can be dynamically tuned to adapt to the changing work load on the system.
- The degree of predictability for performance within a subsystem can be achieved because the usage of resources by that subsystem is isolated from the resources used by other subsystems.

Subsystem description

The subsystem description is the CPF object containing the definition of the operational environment for a subsystem. It includes a definition of the main storage allowed for work to be executed within the subsystem, the sources from which work is to be accepted for execution, the identification of the programs to be invoked to perform the work, and the environment within which those programs are to execute.

This environment description includes the number of jobs that may be concurrently active within the subsystem; the storage pools, a logical grouping of storage, to be allocated for use by this subsystem when it becomes active; and the activity level (multiprogramming level) to be used by the system in managing work within each storage pool.

Work entries defined within the subsystem description identify the sources from which jobs may be accepted for execution.

These sources include work stations, a job queue on which batch jobs have been placed, and automatically started jobs whose definition is contained within the subsystem description. Jobs from all of these sources may coexist within a single subsystem. A single subsystem monitor program that is common to all active subsystems accepts jobs for execution within a subsystem on the basis of the work entries contained in the subsystem description. It is the ability to accept work from a variety of sources and manage those jobs within a single environment that provides the installation with the flexibility it needs in defining its own subsystems.

Routing entries within the subsystem description provide the means of relating the requests for work to be done with the program to perform the work and the environment within which the program is to execute. A *routing data* field is provided each time the subsystem monitor handles a work request. The routing data field is compared with values contained in the routing entries to identify the particular routing entry to be used for initiating execution in behalf of that work request. The routing entry identifies the program to be invoked, the storage pool to which the processing is to be assigned, and a *class* object that contains a collection of predefined execution parameters to be used. A *process* is then initiated to perform the required processing. Each time a new process is thus initiated, the user may view this as another *routing step* within the job.

Figure 2 illustrates the flow of work within an active subsystem.

The ability to predefine an entire subsystem description provides the programmer or application designer with the opportunity to prescribe an entire operational environment for a subsystem. The activation of that environment is then as simple as starting a subsystem that uses that subsystem description as its basis for control.

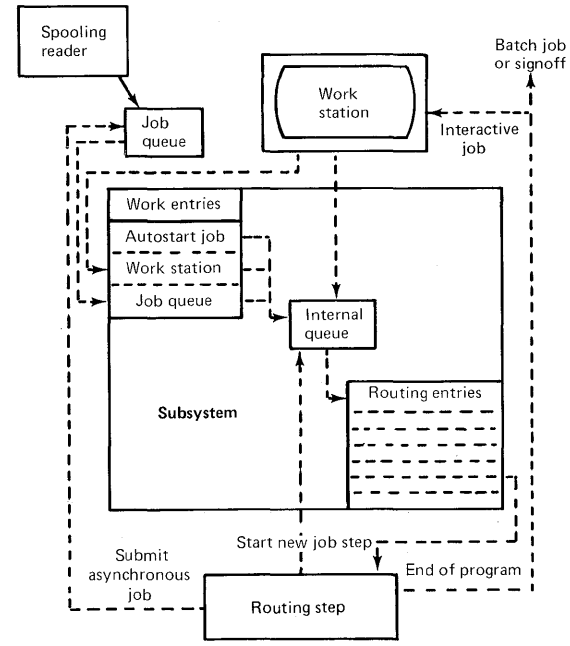


Figure 2 Work flow within an active subsystem

Jobs and job descriptions

All work submitted through CPF work management is processed as *jobs.* Each job must execute within the jurisdiction of a subsystem. In fact, a *controlling subsystem,* whose identity can be specified by the installation through a system value, is automatically started when the system is started and may not be totally deactivated until CPF operation is totally terminated. The first work requested of CPF after it is started is requested through the controlling subsystem. Other subsystems may be started and terminated as the needs of the installation dictate.

There are many parameters for a job, including such things as scheduling priority and message level. To ease the burden of specifying these parameters when a job is submitted, a CPF object called a *job description* is identified. This object contains a complete set of job parameters. A job description is referenced when a job is submitted to the system. Individual parameters may be overridden at the time the job description is referenced. The resulting parameters are used during the execution of the job.

A job is the lowest level of work for which external controls are provided. A job may be held, released, or canceled. Certain of its parameters may be changed dynamically during the execution of the job. The attributes of a job may be displayed. These controls afford adequate operational management of the individual units of work on the system.

Summary

The table-driven interface to CPF work flow management gives the programmer or application designer both power and flexibility in defining the work load to be processed by the system. This definition is contained in the subsystem description, job description, and class objects, as well as in the system values that are supported. Default settings for the system values and usable versions of the definitional objects are shipped with CPF, enabling the installation to operate without defining these objects. On the other hand, the approach offers the installation the opportunity to define its own definitional objects that better reflect its unique requirements.

The operational aspects of work flow management are simple, requiring primarily that the appropriate subsystems be started and terminated at the appropriate times. Lower level controls for managing jobs are available but their use should seldom be required. The total interface thus supported meets the objective of providing a powerful, easy-to-use means of controlling work flow for a broad range of application environments.

References

1. H.T. Norton, R.T. Turner, K.C. Hu, and D.G. Harvey, "System/38 work management concepts," page 81.
2. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.
3. K.W. Pinnow, J.G. Ranweiler, and J.F. Miller, "System/38 object-oriented architecture," page 55.

States that the System/38 message handler provides a single, system-wide capability for defining, sending, and receiving messages. Messages are formally described to the system and can be received by either human users or programs.

The generalized message handler in System/38

R.A. Demers

One of the pervasive problems in operating system design is the communication of data in the form of messages between the system operator, users, programs, system logs, and the machine. The problem has classically been solved by having separate facilities for each type of communication to be done: write-to-operator, write-to-programmer, specialized work station facilities, messages written to output files, machine interrupts, special queues and events, etc. Each of these facilities had unique features not applicable to the other destinations of messages. As a result, the operating environment of a program was in part defined by how and to whom it sent messages. This prevented the development of programs that could be used as off-the-shelf components by many applications. It also required the user and programmer to learn multiple techniques for communication and, in effect, limited certain kinds of communication, such as program-to-program or job-to-job, to system programs. The generalized message handler of the System/38 is an integrated facility of the Control Program Facility (CPF) that provides Control Language (CL) commands and display screens for defining, sending, and receiving messages. The CPF is described by Harvey and Conway [1]. This paper describes the key concepts of the CPF *message handler.*

Messages

Messages are data, but, unlike the records of a data base file, each message is unique, having its own format, content, and meaning. A message can be sent to either a program or a user. When sent to a user, the message must consist of text that can be read and understood by the user. But when sent to a program, text is inconvenient to process. Programs can more easily process data that is organized into fields with predefined attributes.

When work station users send messages to each other, or to the system operator, the text provided by the sender is the whole message. And when a program sends messages to a user, the message must also be in the form of text that can be read and understood. It is desirable to store this text external to the sending programs so that it can be easily modified or translated into other natural languages. The System/38 supports objects called *message files* in which *message descriptions* can be stored. Message descriptions contain text and other attributes of a message.

A program sending a message can provide a set of data fields that can be substituted into the stored text. The attributes of these fields are stored with the text in a message description. The same message can be sent to either a user or a program. The user receives the text, with data substitutions; a program can receive either the text or the data fields. Thus, any message can be handled by a program or presented to a user. This allows messages to be reinterpreted by other programs or handled in different ways by different programs. The sending program is therefore independent of the message recipient, which can be either a user or a program.

Message queues

The CPF supports the ability to send and receive messages via objects called *message queues.* Message queues are associated with particular users, programs, or jobs through ownership, authorization, allocation, and other conventions of use. There are system-created message queues for the system operator, for each work station, for each of the system logs, and for each job. Users can also create message queues to meet specific application needs or for use as personal mail boxes. Figure 1 illustrates the set of message queues that can be used by a job.

Message queues support a wide range of applications, from system operation and logging to user mail

boxes. The methods of delivering and receiving messages are comprehensive and are controlled by the user or program that is associated with and using the message queue. These facilities allow the system operator to be at any work station to operate the system, and they also provide for an unattended mode of operation. In the System/38, these facilities are not limited to the system operator. They are applicable to any message queue, including user message queues, thereby extending the range of application development techniques available to the system and user programmer.

Message queue delivery modes

The user or program associated with a message queue determines the method by which messages are to be delivered by message queue. There are four delivery modes:

1. ***Break.*** When a message arrives at the message queue, the user's job is interrupted and a program is invoked. For interactive jobs, if no other program is specified, a CPF message-handler program is invoked to display the message at the work station. Alternatively, a user program can be specified that presents a formatted display of the message, or performs some function requested by the message. Thus, a job can be interrupted to perform some more important function.
2. ***Notify.*** For interactive jobs, when a message arrives, the work station operator can be notified of its arrival by an indicator light and optional alarm. The messages can be displayed when it is appropriate to the particular job.
3. ***Hold.*** Arriving messages can be held in the message queue for later delivery. These messages can be displayed by issuing a CL command, or they can be received by a program. Programs can poll multiple message queues, or they can wait either indefinitely or for a specific period of time for the arrival of a message. This allows a programmer to coordinate the activities of different jobs of an application.
4. ***Default.*** Arriving messages are rejected and those requiring a reply are answered with a default reply stored in the message description. This delivery mode provides for an unattended mode of operation. A message queue used for the control of an application can be set to default mode for unattended operation of that specific application. The system operator's message queue can be set to default mode for unattended operation of the system itself.

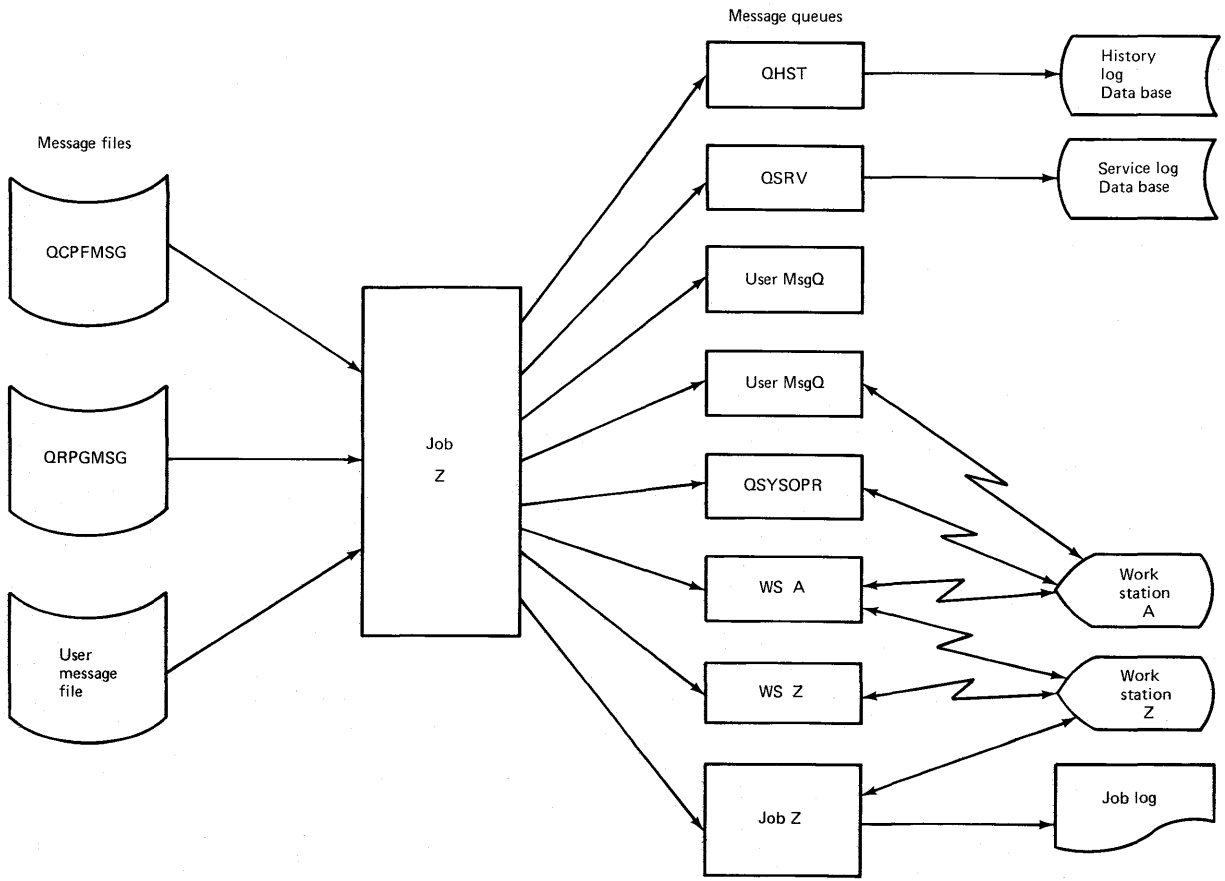


Figure 1 The message files and message queues used by a job

Receiving and displaying messages

Messages are usually received on a first-in-first-out (FIFO) basis. But when a message is received, it is not automatically removed from the message queue, although this is an option. Instead it becomes an old message. The message handler assigns a key to each message on a message queue. This key can be received at the time that a message is received in FIFO order. To receive an old message, the key must be specified. Alternatively, all messages can be reset to the new message status for FIFO access. By leaving messages on a message queue, a program can retain a log of the messages it has processed. All of the messages can be written to a printer file at the end of the job. It can also batch certain types of messages on the message queue and process them all at one time.

The distinction between old and new messages is simplified when messages are displayed at a work station. The user can request the display of messages starting with either the oldest or the newest messages on the message queue. The roll keys of the work station can then be used to view any of the messages on the message queue. Other display options are also available to allow the work station user to reply to messages and to remove them from the message queue.

The user can retain them on the message queue to keep track of the messages received and replies to them. Also, by leaving messages in the queue, the user can delay taking action on particular messages until it is appropriate to do so.

Since the messages at a message queue can be either received by a program or displayed at a work station, it is possible to write a program that filters messages; rejecting some, handling others, and displaying only certain messages. The independence of message queues from particular users, program or human, provides this range of facilities.

Job message queues

The message queues provided for each job are special cases of message queues. A set of logical message queues is built within each job message queue to represent the external user who requested the job and each program invocation of the job. These logical message queues are created only when message-handling services are requested by or for their associated user. The set of messages retained in these message queues at the end of the job is printed as the job log.

Since the program invocations of a job are stacked, the logical message queues associated with each invocation are also considered to be stacked. The external user is considered to be the caller of the initial program on the stack. Machine instructions are considered to be equivalent to called programs, although at a lower level of functional abstraction. A program invocation can send messages to itself, to its caller, to a named program in the stack, or to its caller. If the specified message queue is that of the external user, the message is immediately written to the work station associated with that interactive job.

Programs that send messages to their caller's program message queue give that caller an opportunity to reinterpret the message at a higher level of functional abstraction. For example, the message "cursor not over data space," which is signaled as an exception by the machine data base management instructions, can be reinterpreted as "end of file" by the CPF data base component, and this message can be reinterpreted by the CPF copy component as "copy completed." The receiving program can also handle the message as required by its specific environment or function. The messages can be written to the job log and work station or to specialized displays, they can be utilized by the program internally to correct the problem, or they can be listed.

By sending messages to program message queues, a programmer can design his programs as independent components that can be selected off the shelf for many applications. An example of this is the *command analyzer* component of the CPF which is invoked by the interpretive CL processor, by the prompter, by the source entry utility, and by the CL compiler. As it detects error conditions in commands, it sends messages to its caller's program message queue and does not have to be concerned with how those messages are processed.

Error handling

Sending an error message to a program's caller gives the caller an opportunity to handle the error condition. The message handler provides special support for a type of message that must be handled. These are called *escape messages* because the sending program is terminated. Escape messages are usually sent to a program's caller. The caller can monitor for the arrival of these messages on its message queue and specify the action to be taken, such as a branch to an internal label or a program to be called. But if the caller is not monitoring for a particular escape message, default system action is taken. This includes taking dumps and entering a debugging breakpoint, if appropriate. The final action is to send a "function check" escape message to the same program, thereby giving it a chance to clean up and terminate gracefully. Jobs are never abnormally terminated by low-level functions (with the exception of machine checks and system crashes).

A point of interest is that the exceptions signaled by machine instructions are trapped by the message handler and converted into escape messages. Thus, the full facilities of the message handler are available to handle exceptions and to put out meaningful error messages if necessary.

Summary

The System/38 message handler provides an integrated facility that supports communications between the users and/or programs of the system. Messages are formally described to the system and can be received by either human users or programs. Message queues support a wide range of applications and provide comprehensive methods for receiving and delivering messages. Job message queues support environment-independent communications with the requester of a job and allow programs to be written for off-the-shelf applications. Error handling is a built-in function of the system. And finally, the facilities of the message handler are available through CL commands. CL users thus have the same communications capabilities as system programmers. These advantages have been exploited in the development of the CPF and the other System/38 licensed programs. They remain available to application developers and to users of the system.

References

1. D.G. Harvey, "Introduction to the System/38 Control Program Facility," page 74.

System/38 common code generation

J.K. Allsen

Presents an overview of the System/38 common code generation facility and its significant design considerations.

The IBM System/38 is a completely new system in that all of its programming support components such as language translators and utilities were designed and implemented against a new machine instruction set. A new implementation of a language translator can represent a large programming development and maintenance effort. One approach employed to reduce this effort is a common code-generation facility supporting the multiple code-generation components of the IBM-provided programming support. However, this conceptually simple approach of common code generation presented many practical problems in achieving the obvious benefits.

This paper provides an overview of the System/38 common code generation facility and its significant design considerations.

Program creation

A language translator on System/38 may be either a compiler of high level source language (for example, RPG III) or a utility program which creates another program for immediate execution (for example, Interactive Data base Utilities). All language translators of either type perform some tasks in common on System/38 in order to create a program. The most obvious common task is the generation of a *program template*, which is the target for any program creation. This template is a machine-level object which contains:

- An instruction stream
- Object definitions
- Debugging information

The instruction stream in the program template is a series of bytes representing the machine instructions for that particular program; the object definitions describe the attributes of all items referred to from the instruction stream. Debugging information is also contained in the program template. This information is used during the execution of the program to associate the instructions and objects of the program with their original symbolic source forms.

A traditional approach to target code generation by multiple language translators on a given system is shown in Figure 1. Each language translator (LT) inputs a source program (SRC) encoded in a particular language. Each language translator typically reduces the source program to its equivalent target program by a series of distinct steps, or "passes." The last pass in all cases is the final mapping of intermediate text (IT) into target form. Typically, on a given system, the format of the intermediate text varies from language translator to language translator.

System/38 code generation

To achieve centralization of function on System/38, a common code-generation facility is provided. This facility is the program resolution monitor (PRM). It is

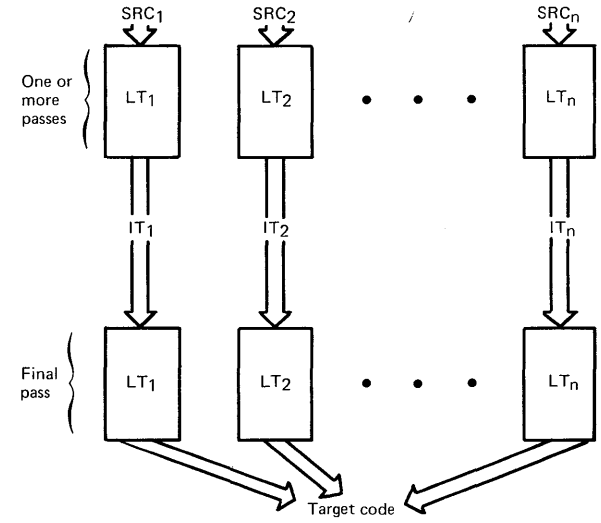


Figure 1 A traditional method for program creation by multiple language translators

part of the System/38 Control Program Facility, and replaces the final pass for all the language translators of System/38. In order to achieve this, it was first necessary to define a common intermediate text format suitable for all the language translators. This textual format is called the intermediate representation of a program (IRP).

IRP is a symbolic text format. In addition to isomorphic symbolic representations of the System/38 instructions, IRP includes symbolic declaration capability for all operands of those instructions. Constants may be either declared explicitly or defined implicitly in an instruction. Debugging information is generated by the language translator via IRP constructs. The net effect is that IRP is the actual target for every System/38 language translator, with the PRM doing the translation of the IRP character string into a System/38 program template. Figure 2 shows the flow of program creation in System/38.

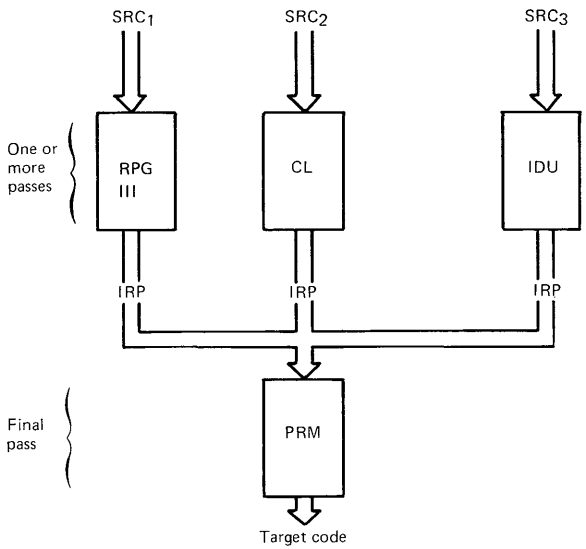


Figure 2 The flow of program creation in System/38

IRP description

The IRP generated by a language translator is passed as a single bit string to the PRM. Within this string is a series of IRP "statements," each terminated by a delimiter. An IRP statement may be either an isomorphic construct representing a machine instruction or a pseudo operation. The pseudo-operation statements do not have direct executable counterparts in the System/38 instruction interface, but are used to transmit object attribute information and debugging data to the PRM for inclusion in the program template.

Isomorphic statements follow the token sequence of their machine-instruction counterparts. A mnemonic operation code is optionally followed by a string of operands, each separated by a delimiter. All operands appear as symbolic names. For compound instructions which terminate in branches or indicator settings, the conditions being tested appear as a sublist after the mnemonic operation code; target labels or indicator names occur after the main operand list. Any isomorphic statement may be labeled with one or more symbolic names. In the generated program, these statements become branch targets.

Pseudo-operation statements all start with an operation verb, followed by appropriate operands. There are eight pseudo-operation statements, each of which falls into one of three categories: object definition, debugging information, and listing control.

There are three object-definition statements: DCL, ENTRY, and RESET. The DCL statement defines symbolic operands referred to by the isomorphic statements. The ENTRY statement defines the name of the entry point of the generated program together with names of associated parameters. RESET causes the default space name for the DIRECT attribute to be changed.

The two statements for debugging information are BRK and LABEL. The BRK statement occurs within a string of isomorphic statements and establishes a breakpoint for later debugging operations. LABEL provides information for symbolic naming of objects.

The listing control statements are SPACE, EJECT, and TITLE. These allow a language translator to control the format of a PRM-generated IRP listing which is optionally produced. To further augment this listing capability, comments are also provided for in the IRP syntax.

Rationale for the System/38 approach to code generation

The notion of a "cascade compiler" is not new, and has been implemented on other systems where compilers cascade into assembler language and then invoke an assembler to complete the program-creation cycle. At the other extreme, proposals have been published which hypothesize a "Common Target Language" which is both source-language-independent and target-system-independent. Attempts to derive such universal targets tend to culminate in grammars with almost mutually exclusive subsets of constructs unique to different languages.

The concept of the IRP target code together with the centralized final pass (the PRM) on System/38 falls between these two extremes. The symbolic nature of IRP frees the System/38 language translators from addressability and location counter considerations since all references are generated "by name." The structure of the program template generated by the PRM and its complexities are not apparent to language-translator developers.

While traditional cascade techniques took advantage of symbolic (assembler language) target code, they also paid the price in performance since the native assemblers used were designed with direct user invocation in mind. This meant that the target generated by a language translator was tuned for direct source encoding instead of automatic genera-

tion. The IRP of System/38 was derived with ease of automatic generation as its primary constraint. A secondary constraint was ease of translation of the IRP syntax by the PRM into the bit-oriented program template for a high rate of performance.

Summary

The System/38 approach to common code generation is a pragmatic one. By centralizing the final pass of each language translator into a single component (the PRM), the total development and maintenance effort has been reduced for the language translators. The System/38 definition of the common code-generation interface represents design trade-offs aimed at optimizing the benefit to compilers.

Authors

Gary F. Aberle

General Systems Division, Rochester, MN

Mr. Aberle received a BS degree in business administration from the University of Missouri in 1969. He joined IBM in 1969 as a system programmer working on the System/3 and System/32. He has been involved with the System/38 for the past six years, working on architecture for machine data base support and later as manager of the department responsible for the machine data base support implementation. He is currently an advisory programmer with interests in overall system performance and advanced data base concepts.

J. K. (Ken) Allsen

General Systems Division, Rochester, MN

Mr. Allsen received the BS degree in mathematics from Indiana Institute of Technology. Since joining IBM in 1965, he has done graduate study in computer science at the University of Minnesota. His career at IBM has included compiler design and development for various programming languages. He is currently a member of the Design Control Group for languages in the Advanced Systems Development Group at Rochester, MN.

Frank E. Benson

General Systems Division, Rochester, MN

Mr. Benson joined IBM in 1966 in Rochester, MN, where he worked in compiler design. In 1968, he started work in OS data management where he was instrumental in developing the support of OCR/MICR devices and several unit record devices. In 1973 he joined the System/38 effort working in the architectural development of System/38 data base. Since then, he has worked in the data base/data management area as well as on several other aspects of System/38. Mr. Benson received a BS in mathematics from the University of Minnesota in 1966.

Neil C. Berglund

General Systems Division, Rochester, MN

Mr. Berglund joined IBM in 1965 and has had various assignments in I/O adapter and processor design for the IBM System/3. He holds several patents for his work in these areas including an IBM Outstanding Invention Award and an IBM Outstanding Contribution Award for his work on the System/3 Model 15 processor. His most recent assignment has been technical design leadership of the System/38 CPU. Mr. Berglund is an electrical engineering graduate of the University of Minnesota.

Viktors Berstis

General Systems Division, Rochester, MN

Mr. Berstis is a member of the System Architecture Department. He received his BS in mathematics and physics in 1971, and his MS in computer information and control engineering in 1974 from the University of Michigan, Ann Arbor. Prior to joining IBM in 1977, he worked on the development of the Michigan Terminal System. Mr. Berstis is a member of the Association for Computing Machinery.

Kenneth W. Borgendale

General Systems Division, Rochester, MN

Mr. Borgendale joined IBM at the General Systems Division Development Laboratory in Rochester, MN in 1977. He is currently a senior associate programmer in the Data and Translation Functions Department of the Advanced Systems Development Group. He received a BA degree in 1976 from the University of Minnesota and has completed course work for an MS in computer science, also at the University of Minnesota.

J. Howard Botterill

General Systems Division, Rochester, MN

Mr. Botterill is an advisory programmer in the Advanced Systems Development Group where he has design control responsibility for the System/38 Control Language and the Control Program Facility user interface. He joined IBM at Rochester in 1967 and was involved in the development of the Multiple Terminal Monitor Task (MTMT) terminal system for the System/360 and the Communication Control Program (CCP) for the System/3. He received his BS in mathematics from Wheaton College, Wheaton, IL, in 1964. In 1965, he received his MS in mathematics from the University of Michigan. Prior to joining IBM, Mr. Botterill was employed at the White Sands Missile Range where he worked on the IBM 7044/7094 Direct Coupled System.

Darryl T. Brunsvold

General Systems Division, Rochester, MN

Mr. Brunsvold is a staff engineer in Advanced Systems Engineering. Since 1975, his principal responsibility has been the attachment of printer products to small business systems using microprocessors. Prior to joining the printer attachment development group, he worked with direct access storage device attachments and performed measurements and analysis of small business systems. Mr. Brunsvold joined IBM in 1970 after receiving a BS degree in electrical engineering from Montana State University.

Robert W. Collins

General Systems Division, Rochester, MN

Mr. Collins is an advisory programmer in the Advanced Systems Development Group. He joined IBM in 1966 as a systems engineer in Des Moines, Iowa, and worked primarily with OS/360 and PARS (financial) systems. From 1970 through 1973, he helped develop CP-67 and VM/370. In 1975, he became team leader responsible for storage management design and implementation on System/38. He is currently working in performance analysis for System/38. Mr. Collins received his BS in electrical engineering from Iowa State University in 1966. He is a member of the Association for Computing Machinery.

Huntington W. Curtis

IRD Biomedical Systems, Mt. Kisco, NY

Dr. Curtis joined IBM in 1959, becoming a senior engineer in 1960. After serving as Manager of Technical Requirements in Federal Systems Division headquarters, he was promoted to technical advisor to the IBM Vice President for Research and Engineering, followed by assignments on the Corporate engineering staff as director of government technical liaison and as director of scientific and technical information. This paper was prepared during his recent assignment on the General Engineering technical staff at the East Fishkill Laboratory.

He received a BS in chemistry and physics from the College of William and Mary in 1942, an MS in physics and electrical engineering from the University of New Hampshire in 1948, and a PhD in electrical engineering from the State University of Iowa in 1950. Prior to joining IBM, he was a professor of electrical engineering at Dartmouth College. Dr. Curtis is a member of Phi Beta Kappa, Tau Beta Pi, and Sigma Xi; a senior member of the Institute of Electrical and Electronics Engineers; and a Trustee of the Mount Washington Observatory.

Stephen H. Dahlby

General Systems Division, Rochester, MN

Mr. Dahlby is a development programmer in the System/38 machine development area of Advanced Systems. His experience on System/38 includes system architecture and machine design control and management positions in machine development. He is currently a manager in the Device Data Management area of the Control Program Facility. He has previously worked in the areas of performance measurement, system modeling, and compiler development. Prior to joining IBM in 1969, he received a BA degree in mathematics from the University of Northern Iowa, Cedar Falls, in 1966, and an MS degree in computer science from Iowa State University, Ames, in 1969.

Richard A. Demers

General Systems Division, Rochester, MN

Mr. Demers is a Staff Programmer in the System/38 Architecture Department. He has had design responsibility for the Message Handling and Service components of the System/38 Control Program Facilities licensed product. In 1968 he joined IBM as an Applications Programmer in White Plains, NY. He received a BA degree in Philosophy from Canisius College, Buffalo, NY, in 1969. In 1972, he moved to Endicott, NY where he worked on OS/VS1, OS/VS2, and DOS/VS support for the IBM 3895 Optical Check Reader. Mr. Demers moved to Rochester, MN in 1975. He is a member of the Association for Computing Machinery, and his interests include operating system architecture, software design methodologies, and software reliability.

Nicholas M. Donofrio

General Technology Division, Burlington, VT

Mr. Donofrio joined IBM in Poughkeepsie in 1967 working in the area of FET memory circuit development. He became a project manager in 1972 and managed the C/P and Riesling Array development activity in Burlington, Vermont. In 1977, he assumed the position of Manager of Advanced Memory Components Development. In September, 1978 Mr. Donofrio was named Administrative Assistant to the Vice President of Development and Manufacturing, GTD Burlington. Mr. Donofrio is currently the Manager of the Systems and Test Organization. He is responsible for the Design Systems of current and advanced product technologies in the Laboratory. He is also responsible for the Test Engineering, Computer Sciences, and Release/Control Center for the Burlington, Vermont Laboratory, and is the EDS Senior Manager for the General Technology Division.

Eugene F. Dumstorff

General Systems Division, Rochester, MN

Mr. Dumstorff is a development engineer in Advanced Systems I/O Development at the Rochester Development Laboratory. He joined IBM in 1969 and has been involved with various projects, primarily in the area of microprocessor development. His most recent assignment has been with development and application of a microprocessor in the I/O subsystem for the IBM System/38. Mr. Dumstorff received a BSEE from Iowa State University, an MSEE from the University of Minnesota, and a BA in business administration from Winona State University.

Wayne O. Evans

General Systems Division, Rochester, MN

Mr. Evans is an advisory programmer in the Advanced Systems Development Group at the IBM General Systems Division Development Laboratory in Rochester, MN. Since joining IBM in 1964, his experience includes computer monitoring of cardiovascular patients, communications I/O support for the System/3, and the Multiple Terminal Monitor Task (MTMT) terminal system for the System/360. He is currently working in development of the System/38 Control Program Facility. Mr. Evans received a BS in mathematics and chemistry from Adams State College, Alamosa, CO, in 1962 and an MS in mathematics from Kansas State University in 1969. Prior to joining IBM, he was employed by the NASA Lewis Research Center.

Ronald O. Fess

General Systems Division, Rochester, MN

Mr. Fess is currently a member of the Design Control Group for the System/38 Control Program Facility. He previously worked on various development projects in OS/MFT, OS/MVT, OS/VS1, and OS/MVS. Prior to joining IBM in 1969, he received a BS in mathematics from Augustana College, Rock Island, IL, and an MS in computer science from the University of Iowa. Mr. Fess is a member of the Association for Computing Machinery.

Barry Flur

General Technology Division, Burlington, VT

Dr. Flur joined IBM in Poughkeepsie in 1959 after receiving his PhD in metallurgical engineering from Carnegie Institute of Technology. He transferred to the Burlington Laboratory in 1965 working on the development of magnet film memories. During 1967, he was on assignment to the Hursley Laboratories in the United Kingdom. For the past several years, he has managed many areas of the SAMOS program, including process development, reliability engineering, test and characterization engineering, and the 82mm pilot line. In 1978 he assumed responsibility for CFET Process Engineering and Development.

Robert E. French

General Systems Division, Rochester, MN

Mr. French is a project programmer in the Advanced Systems Development Group. He joined IBM in 1969 after receiving the BA degree in mathematics from Brown University, Providence, RI. He is currently involved with machine system design. Prior to joining GSD, he worked in the IBM Poughkeepsie Laboratory on simulation of operating systems and system performance analysis.

James W. Froemke

General Systems Division, Rochester, MN

Mr. Froemke is a senior engineer with the Engineering, Programming and Technology corporate staff in Valhalla, NY. He joined the IBM Rochester Development Laboratory in 1964 where he worked in telemetry and data acquisition system design. After a leave of absence, he returned with an MS degree in electrical engineering from North Dakota State University in 1967. In 1969 and 1971, he received an IBM Outstanding Contribution Award and Invention Achievement Award for his work with the System/3 communications architecture and development. Mr. Froemke joined the System/38 development team in 1972, participating in the early architectural definition of communications and microprocessors. In 1974, he assumed management responsibility for the development of magnetic media I/O device attachments for System/38 before becoming the technical assistant to the manager of GSD Advanced Systems. He joined EP&T as program director of small central systems in 1979.

William E. Hammer

General Systems Division, Rochester, MN

Mr. Hammer is an advisory engineer in Advanced Systems Engineering where he is responsible for the architecture and implementation of the attachment of work stations to the System/38. Since joining IBM in 1960, he has been involved in several projects using microprocessors for controlling I/O devices. He received a BS degree in electrical engineering from the University of Illinois.

David G. Harvey

General Systems Division, Rochester, MN

Mr. Harvey is a development programmer and manager of the Design Control Group for the System/38 Control Program Facility. He joined IBM at the Rochester, MN Development Laboratory in 1968 after graduating from Iowa State University with a BS degree in mathematics. From 1968 to 1971, he was a designer and developer of a rule-driven interactive system for Computer-aided Mechanical Engineering Design (COMMEND). In 1971, he joined the System/3 Development Group where he later received an Outstanding Contribution Award for his work on the System/3 Model 15 supervisor. Mr. Harvey joined the Advanced Systems Development Group in 1974 and worked on system architecture before accepting his current assignment.

Nyles N. Heise

General Systems Division, Rochester, MN

Mr. Heise is a staff engineer at the IBM Rochester Development Laboratory. He joined IBM in 1968 after graduating from the University of Wisconsin in electrical engineering. He received his MS degree from the University of Minnesota in 1975 also in electrical engineering. While at IBM, his major emphasis has been on the attachment of magnetic media devices to Rochester-developed computing systems.

G. G. (Glenn) Henry

General Systems Division, Rochester, MN

Mr. Henry is the manager of IBM System/38 programming development. His area's responsibilities include architecture, design, development, performance evaluation, test, and release of licensed programs (CPF, RPG III, IDU, etc.) and some microprogram components. Mr. Henry joined the GSD Advanced Systems Development Group in 1973. His prior experience includes management of programming development for the IBM System/32 and various management and technical assignments on System/3, 1800, and other small system projects. In 1975 he received an IBM Outstanding Contribution Award for his work on System/32. Mr. Henry joined IBM in 1967 after receiving an MS degree in mathematics from California State University, Hayward. Prior to joining IBM, he worked as a consultant in data processing on a number of projects and was employed by the Shell Development Corporation. He is a member of the Association for Computing Machinery.

Roy L. Hoffman

General Systems Division, Rochester, MN

Dr. Hoffman is a senior engineer in a system organization and planning group. His current interests are in storage technology applications and system performance analysis. He received the BS and MS degrees in electrical engineering from the South Dakota School of Mines and Technology in 1959 and 1962, respectively, and the PhD in electrical engineering from the University of Denver in 1967. He joined IBM in 1962 and has held a variety of assignments including acoustical analysis, mechanical diagnostics, and pattern recognition. Dr. Hoffman is a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers.

Darrell J. Horn

General Systems Division, Rochester, MN

Mr. Horn received the BS and MS degrees in electrical engineering from the University of Minnesota. Graduate study emphasis was on digital signal processing and statistical communication theory. Since joining IBM in 1975, Mr. Horn has been a member of Advanced Systems Communications Development Engineering.

Merle E. Houdek

General Systems Division, Rochester, MN

Mr. Houdek joined IBM in 1964 after graduating from Tri-State University in electrical engineering. His previous assignments have been with Custom Systems and the 3740 Data Entry System Development group. He is currently an advisory engineer in Advanced Systems Engineering with interests in the performance analysis and modeling of CPU and I/O hardware.

Philip H. Howard

General Systems Division, Rochester, MN

Mr. Howard joined IBM at Poughkeepsie in 1954 after receiving a BS degree in Electrical Engineering from the University of Wisconsin, Madison. He received an MEE degree from Syracuse University in 1959 and attended the IBM Systems Research Institute in 1973. He is now a senior engineer engaged in system performance analysis and was responsible for the development of index functions. He has been involved with System/38 since its inception. His other work interests include computer system modeling, automated logic design for pattern recognition systems, sorting, and storage management algorithms. He is a member of Eta Kappa Nu and the Association for Computing Machinery. Mr. Howard received IBM Invention Achievement Awards in 1968 and 1974 plus an IBM Outstanding Contribution Award for work on automated logic design in 1972.

Kuang Chi (George) Hu

General Systems Division, Rochester, MN

Mr. Hu received a BSEE degree from the National Taiwan University in 1956 and an MSEE degree from the University of Minnesota in 1959. He joined the IBM Thomas J. Watson Research Laboratory in 1959 and in 1964 joined the IBM Systems Development Division in Rochester, MN. He worked on pattern recognition problems and the development of optical character recognition machines until 1968. Before joining the Advanced Systems Development Group in 1973, he worked on an IBM-Mayo Clinic joint project in developing ECG signal recognition algorithms, ECG diagnostic decision programs, and patient monitoring systems. He is currently working in the area of resource management for the System/38.

David O. Lewis

General Systems Division, Rochester, MN

Mr. Lewis is an advisory engineer in Advanced Systems Engineering where he has been involved in the design and implementation of the System/38 channel. His experience includes logic design, simulation, and microcode support associated with I/O adapter and channel designs. Prior to joining IBM in 1968, he received a BSEE degree from the University of Wisconsin.

Larry W. Loen

General Systems Division, Rochester, MN

Mr. Loen is a member of the Advanced Systems Development Group in the IBM General Systems Division Development Laboratory in Rochester, MN. His primary work on System/38 has been in the area of main storage management. He joined IBM in 1974 at the Systems Product Division Poughkeepsie Laboratory and transferred to GSD in Rochester in 1976. His experience prior to System/38 includes compiler development and testing. Mr. Loen holds a BS in computer science from Michigan State University (Lyman Briggs College).

Joel F. Miller

General Systems Division, Rochester, MN

Mr. Miller received the BA degree in mathematics from Pennsylvania State University in 1960. He joined IBM in 1960 at the Endicott Laboratory where he was involved in the programming support for the 1400 series computers and later the System/360 family. He received an IBM Outstanding Contribution Award for his work in the definition of the System/360 RPG language. After transferring to Rochester in 1967, he was involved in the development of System/3 and was manager of a Programming Advanced Technology department until 1971. Since that time, he has held various positions in the development of System/38, primarily in system architecture and system build.

Glen R. Mitchell

General Systems Division, Rochester, MN

Mr. Mitchell joined IBM in 1964 in Endicott, NY, where he was a member of the group which designed the automatic test equipment for SLT modules and cards. After joining the System/3 design group in Rochester in 1968, he was involved with the design of the communication attachments. After joining the Advanced Systems Development Group in 1972, he worked on the virtual addressing aspects of System/38, specifically the translation hardware and microcode used by the CPU and channel. Mr. Mitchell is an advisory engineer and is now working in the New Business Systems Design Group. He received his BS in electrical engineering from Iowa State University.

Henry T. Norton

General Systems Division, Rochester, MN

Mr. Norton is a senior programmer concerned with design control for the Control Program Facility licensed program product of System/38. He joined the Advanced Systems Development Division of IBM in 1960 at San Jose, where he was involved in the software design for Advanced Systems. He has since been associated with programming product test in Poughkeepsie, NY, the design of programming development tools and text processing systems in Boulder, CO, and the development of programming support for the SAFEGUARD ABM system in Morris Plains, NJ. He joined the Rochester Laboratory in 1975 where he has been engaged in the architecture and design of the programming support for the IBM System/38. Mr. Norton received his BS in mathematics from Oregon State University in 1952. He is a member of Pi Mu Epsilon and Phi Kappa Phi. Mr. Norton received an IBM Invention Achievement Award in 1971.

James J. Pertzborn

General Systems Division, Rochester, MN

Mr. Pertzborn joined IBM in 1973 at the Rochester Development Laboratory after graduating from the University of Wisconsin with a BS degree in electrical engineering. His experience includes logic design, simulation, and test-data generation associated with PLA-implemented I/O adapters. He is a staff engineer involved in the design and test of disk I/O attachments.

Ronald A. Peterson

General Systems Division, Rochester, MN

Mr. Peterson joined IBM in 1973 after graduating from the University of Minnesota with an MS degree in electrical engineering. He received his BS degree in physics from Seattle Pacific University. For the past five years, he has participated in the design of the disk I/O attachment for the System/38. Mr. Peterson is a staff engineer at the IBM Rochester Development Laboratory.

Kurt W. Pinnow

General Systems Division, Rochester, MN

Mr. Pinnow is a staff programmer at the Rochester Laboratory where he holds design control responsibility for portions of the programming support for IBM System/38. He joined IBM in 1970 at the Federal Systems Division where he was engaged in the simulation modeling of the effects of nuclear weapons. In 1974, he moved to Rochester, where he became involved in the architecture and design of IBM System/38. Mr. Pinnow holds an applied mathematics and physics degree from the University of Wisconsin (BS 1969). Prior to joining IBM, Mr. Pinnow worked for Calspan Corporation in Buffalo, NY. His interests include architecture and performance modeling of computer systems.

James G. Ranweiler

General Systems Division, Rochester, MN

Mr. Ranweiler is a senior programmer in Rochester's Advanced Systems Development Group. He joined IBM in 1967 to work on the development of System/3. In 1969, he received an IBM Outstanding Contribution Award for his design of the preassemble function of the System/3 RPG II compiler. Since 1974, he has been a designer of System/38 microcode and of the System/38 instruction interface. Currently his primary responsibility is performance of the System/38. Mr. Ranweiler received a BA degree in mathematics in 1964 from St. John's University at Collegeville, MN.

John W. Reed

General Systems Division, Rochester, MN

Mr. Reed joined IBM in 1967 after receiving a BSEE degree from Kansas State University and a MSEE degree from the University of Wisconsin, Madison. His past IBM assignments have been involved in the application of LSI technologies, microprocessors, and behavioral modeling techniques to GSD products. Since joining the Advanced Systems Engineering group in late 1973, he was responsible for the definition and implementation of the System/38 channel and was manager of the department responsible for the System/38 processor hardware and storage. He is currently an advisory engineer in the Technical Programs organization in the Rochester laboratory.

Dale N. Reynolds

General Systems Division, Rochester, MN

Mr. Reynolds is a senior programmer and manager of the System/38 system architecture, system performance and system communications area. He joined the Advanced Systems Development Group in 1973 and has been involved since in the definition, architecture, implementation, and performance of the System/38 instruction interface. He previously worked on the IBM System/32 and on prototype systems at the IBM Los Gatos laboratory. In 1975 he received an IBM Outstanding Contribution Award for his work on System/32. Mr. Reynolds received an MS in computer science from the University of Utah in 1966. He is a member of the Association for Computing Machinery.

Steven R. Ridenour

General Systems Division, Rochester, MN

Mr. Ridenour joined IBM at Rochester in 1969 after receiving a BS degree in mathematics from Iowa State University. He initially was a member of the System Control Program Development Group for the System/3 Models 6, 10, and 15, working primarily on data management for disk, tape, and unit record devices. In 1974, he transferred to Advanced Systems where he began work on the data definition interface for System/38. Mr. Ridenour was a manager in the Control Program Facility area of Advanced Systems until June, 1979. Currently, he is a manager in Programming Assurance for the Atlanta Application Development Center.

Thomas S. Robinson

General Systems Division, Rochester, MN

Prior to joining IBM in 1963, Mr. Robinson was a member of the engineering staff at RCA Laboratories in Van Nuys, CA, and Tucson, AZ. His past assignments with IBM have been primarily in the design and development of Advanced Optical Character Recognition Equipment. Since 1975, Mr. Robinson has been involved in I/O channel architecture with Advanced Systems Engineering. He received a BSEE degree in 1960 from New Mexico State University and has done graduate work at the University of California.

Francis X. Roellinger, Jr.

General Systems Division, Rochester, MN

Mr. Roellinger joined IBM in 1973 at the Rochester Development Laboratory. He received his MSEE degree from the University of Missouri in 1973, where he did research in realtime pattern recognition. From 1973 to 1978 he was with a communications engineering department, where his work included data link control analysis and microprogram design, simulation, and performance measurements. Currently he is with a communications programming department, where he is involved with I/O interfacing and computer network analysis and design.

Robert T. Schnadt

System Products Division, Boeblingen, Germany

Dr. Schnadt joined IBM in October, 1970, in the Boeblingen laboratory, worked on device design for FET and bipolar technologies and on circuit design analysis for several FET and bipolar memory array chips. He became a project manager in September, 1975, for FET memory array design and managed three major array-chip development programs. He is now responsible for the advanced FET chip development programs.

Thomas R. Schwalen

General Systems Division, Rochester, MN

Mr. Schwalen is an advisory programmer in the Advanced Systems Development Group. He joined IBM in 1970 at the Rochester, MN Development Laboratory and from 1970 through 1973 was a member of the System Control Program Development Group for the System/3, Models 6, 10, and 15. He joined Advanced Systems in late 1973 and is currently working in the development of the System/38 Control Program Facility licensed program product. Mr. Schwalen received a BBA degree in computer center administration from Wisconsin State University, Whitewater, in 1970.

Frank G. Soltis

General Systems Division, Rochester, MN

Dr. Soltis first joined IBM in 1963 at the Rochester Development Laboratory after receiving his BS and MS degrees in electrical engineering from North Dakota State University. In 1968, while on an education leave of absence, he received his PhD degree in electrical engineering from Iowa State University. Since returning to Rochester, Dr. Soltis has been involved in the definition of computer system architectures including the System/38 architecture. His current interests are in small system architecture designs and microprogramming techniques. Dr. Soltis is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery.

Perry T. Taylor

General Systems Division, St. Louis, MO

Mr. Taylor is a senior systems engineer in the GSD St. Louis branch office. Previous assignments include: Systems engineer on Army Material Command account, staff member at the IBM Systems Research Institute in New York City, and manager of System Architecture for System/38. He received a BA degree in mathematics from Southern Illinois University at Edwardsville, IL, in 1963 and is a member of the Institute of Electrical and Electronics Engineers.

John N. Tietjen

General Systems Division, Rochester, MN

Mr. Tietjen joined IBM in 1970 after graduating from Arizona State University in electrical engineering. Prior to his work on the development of the work station controller for the System/38, he was involved in the design and development of a microprocessor-based line printer controller. He is currently a staff engineer in Advanced Systems I/O Development in Rochester.

C. D. (Dave) Truxal

General Systems Division, Rochester, MN

Mr. Truxal is an advisory programmer engaged in data base, authorization-related design and language processors on System/38. He received a BA degree in mathematics in 1967 from the University of Kansas, and joined IBM in Kingston, NY, to work in virtual storage management. This work was interrupted by two years in the US Navy, serving as staff to the director of programming, BUPERS, Washington, DC. Mr. Truxal rejoined IBM in 1977, after working at Control Data Corporation, Minneapolis, MN. Other experience and interests include architecture and design of resource management and I/O facilities. He is a member of the Association for Computing Machinery.

Richard T. Turner

General Systems Division, Rochester, MN

Mr. Turner is currently a consulting Systems Engineer in the Bloomington, IL brance office. He was formerly a member of the Design Control Group for the System/38 high level machine microcode. He joined IBM in 1964 in the Federal Systems Division at Omaha, NB. Prior to joining the GSD Advanced Systems Development Group in 1973, he worked on the APOLLO project (1965-1967) in Houston and in the IBM Data Processing Division as a systems engineer in the St. Louis branch office (1967-1971). He transferred to the IBM GSD Development Laboratory in Rochester, MN in 1971. Mr. Turner received a BA degree from Southern Illinois University in 1964 and has done graduate work at Creighton. He received an IBM Outstanding Contribution Award in 1973 for work on System/3 CCP and an IBM Outstanding Invention Award for patent activity in 1979 for work on the System/38.

C. T. (Tom) Watson

General Systems Division, Rochester, MN

Mr. Watson is a manager in System/38 CPF responsible for Data Description Specifications, Device Configuration and Save Restore Functions in the Advanced Systems Development Group. He joined IBM in 1968 upon receiving his BS in engineering mechanics from the University of Michigan. He received his MS in computer science from Washington State University in 1975 while on education leave from IBM. He participated in the development of the 3740 group of products and since 1975 has worked mainly in development of the System/38 data base facility. Mr. Watson has spoken widely within IBM on flowcharting tools and techniques and on the design and implementation of the System/38 data base facility. Mr. Watson is a member of Pi Tau Sigma and the Association for Computing Machinery.

Appendix: Reader's guide

For an *overview* of the entire system, you should read:

"Introduction to IBM System/38 architecture," page 3

For the next level of detail:

"Hardware organization of the System/38," page 19

"System/38–A high-level machine," page 47

"Introduction to the System/38 Control Program Facility," page 74

"User-System/38 interface design considerations," page 70

If your particular interest is in the underlying *machine structure and technology,* read:

"Hardware organization of the System/38," page 19

"Integrated circuit design, production, and packaging for System/38," page 11

"Memory design/technology for System/38," page 16

"Translating a large virtual address," page 22

"Processor development in the LSI environment," page 7

"System/38 I/O structure," page 25

For further detail on I/O, there are three papers related to control and three to individual elements–work stations, communication subsystem, and printers:

"Application of a microprocessor for I/O control," page 28

"System/38 magnetic media controller," page 41

"Shared function controller design," page 44

"Microprocessor-based work station controller," page 36

"Microprocessor-based communications subsystem," page 32

"Microprocessor control of impact line printers for printing character-string data," page 38

The high-level machine structure is described in eight papers:

"System/38–A high-level machine," page 47

"System/38 addressing and authorization," page 51

"System/38 object-oriented architecture," page 55

"System/38 work management concepts," page 81

"System/38 data base concepts," page 78

"System/38 machine storage management," page 63

"System/38 machine indexing support," page 67

"System/38 machine data base support," page 59

To gain an understanding of the *advanced programming support,* you will want to read some or all of the following:

"Introduction to the System/38 Control Program Facility," page 74

"System/38 addressing and authorization," page 51

"System/38 object-oriented architecture," page 55

"System/38 work management concepts," page 81

"System/38 data base concepts," page 78

The next two papers present, in some detail, two especially important elements of the Control Program Facility:

"The rule-driven Control Language in System/38," page 83

"File and data definition facilities in System/38," page 87

For more detail on advanced programming support, read:

"Table-driven work management interface in System/38," page 94

"File processing in System/38," page 91

"The generalized message handler in System/38," page 97

"System/38 common code generation," page 100

International Business Machines Corporation

General Systems Division
4111 Northside Parkway, N.W.
P.O. Box 2150
Atlanta, Georgia 30055
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
U.S.A.
(International)

G580-0237-1
Printed in U.S.A.