AD 667 172

DEFENSE DOCUMENTATION CENTER FIVE YEAR PLAN STUDY. VOLUME III. STATE OF THE ART STUDY

System Development Corporation
Falls Church, Virginia

29 August 1966

TM-WD-268/003/00

DEFENSE DOCUMENTATION CENTER

FIVE YEAR PLAN STUDY

VOLUME III

STATE OF THE ART STUDY

29 AUGUST 1966

(15)

TM-WD-268/003/00

TECHNICAL MEMORANDUM

(TM Series)

This document was produced by SDC in performance of DSA-600-12301

DEFENSE DOCUMENTATION CENTER

FIVE YEAR PLAN STUDY

VOLUME III

State of the Art Study

29 August 1966

SYSTEM DEVELOPMENT CORPORATION
5720 COLUMBIA PIKE
FALLS CHURCH
VIRGINIA
22041

WASHINGTON OPERATIONS CENTER
CORPORATE OFFICES: Santa Monica, California

TABLE OF CONTENTS

VOLUME III

TABLE OF CONTENTS (Continued)

VOLUME III

ABSTRACT

This document is the result of a three-month study by the System Development Corporation, conducted for the Defense Documentation Center (DDC) in fulfillment of Contract DSA-600-12301. The report consists of three volumes:

Volume I	Five Year Plan
Volume II	Needs and Requirements
Volume III	State of the Art Study

Three alternative plans are presented for the operation and development of DDC for the Fiscal Years 1967 through 1971. Estimates of workload, equipment, personnel, and costs are given to Fiscal Year 1971 for each of the alternatives.

The first two years will be used to clean up the existing systems and to develop an Internal Integrated System that would serve as a basis for future growth in both documentary and management information operations. This integrated system will provide the basis for management and control of the total DDC operation including technical processing, ADP operations, and reproduction and distribution. It will include generalized ADP programs for use in documentary and management information processing as well as procedures necessary for efficient operation of non-ADP activities within the integrated system. The system will provide flexibility and efficiency by reducing the number of operational procedures and computer programs developed each time a new requirement is placed upon DDC.

The last three years will be spent in a continuing upgrading of the basic Internal Integrated System by incorporating new advances in the state-of-the-art in document and management processing into the system as well as any new missions that DDC is asked to perform.

A development program in document processing and user services utilizing both in-house and contractor personnel shall be activated. In addition to advancing the state-of-the-art in document and user services, studies shall be made to define future integrated system elements that will be utilized by DDC in later years.

A. INTRODUCTION

Volume III contains materials which are intended to provide aid to Defense Documentation Center (DDC) in its evaluation, selection and utilization of computer hardware and software.

Section B, State of the Art in Hardware, depicts the most likely development of computer hardware in its organization, capabilities, and application during the next five years. Such topics of interest to DDC as bulk storage developments, information retrieval equipment trends, advanced ("third generation") computer capabilities, and optical reader potential are treated in detail in this Section.

Section C, State of the Art in Software, consists of several sections, each of which analyzes an aspect of computer software of concern to DDC. Section 1 contains a discussion of general characteristics of programming languages. The second section provides detailed specifications of Fortran, Cobol, PL/I, and Decision Tables, the four major computer language classes considered to be most compatible across manufacturers lines, and expected to be emphasized in future software development. Within DDC's area of application, executive systems and file processing systems are considered of prime importance and are found respectively in Sections 3 and 4.

Sections 5 and 6 are entitled Choosing a Programming Language and Checklist of Questions for Programming Language Evaluation. They include criteria of general application in choosing an assembly language or a compiler language, and of specific application to DDC requirements.

Section 7, Present and Future Trends in Programming Languages, covers today's trends, developments expected over five and more years, and their relation to expected DDC programming language needs.

This volume is designed for use by DDC and DDC contractor personnel as a tool for:

Evaluating the current DDC computer hardware and software state-of-the-art.

Assessing today's general state-of-the-art in these areas.

Providing material for use in recommended evaluation and selection activities.

Showing selected examples of advanced applications of the state-of-the-art of hardware and software, for use in related DDC design and analysis activity.

B. STATE OF THE ART IN HARDWARE

The past few years have seen major advances in information processing. Hardware and software components have been improved in many ways. Both fields have taken strides forward but perhaps the biggest source of facility has come from their combination as parts of a single system. New applications have been met not by either the hardware or the software, but by improved combinations of the two.

The existence of time-sharing and multiprocessing today depends far less on significant changes in concept of either the hardware or the software than on the recent changes in thinking about the organization of the total system. The hardware time-sharing systems includes: large memory capacities, large peripheral storage, and accomodation for a multiplicity of communication lines. These in themselves would not provide time-sharing without the executive programs for handling the many "simultaneous" programs and peripheral and terminal equipments of various characteristics. Multiprocessor systems make use of hardware elements that are generally indistinguishable from single processor installations (except for the additions providing rapid data communications between the several processors in the system). The software for control and interrelations of the several hardware elements is a necessary concomitant.

Unfortunately, improvements in each field have often not anticipated changes in the other. This has led to duplicate capabilities despite the fact that a system can make effective use of only one or the other. For example,

hardware elements have been specifically designed to handle time-sharing. Concurrently, software has been developed to provide time-sharing on hardware not designed with this function. This may lead in a few years to the user option to have time-sharing either by means of hardware or software. However, the apparent value of this type of redundancy is outweighed by a disadvantage: the user will be unable to benefit from the advances in both fields simultaneously. By choosing an aspect of one subsystem, he may be ruling out the full advantages of the other subsystem.

1. MAJOR COMPUTER ELEMENT CLASSES

Today's information processing hardware differs from that of some years back both in details and in organization. On the other hand there have been few breakthroughs in component design. Internal memories are still primarily magnetic core although continued development has made them smaller and faster. Vacuum-tube circuitry has been replaced by transistors which in turn are being replaced by integrated circuitry. This has led to reduction in size and power consumption, together with some increase in speed. Few changes have been made in peripheral equipment except for the greater use of cathode-ray-tube displays. Bulk memory (mass storage) devices still primarily consist of magnetic tape, whose packing density and transfer rates lately have been increased somewhat. The use of disk, microform, and magnetic card memories has increased storage capabilities while decreasing access time to data stored in bulk memories. On the other hand, processes in development five years

ago, such as photographic memory or other optical memories, have not been adapted to general use. Optical scanners are only now entering the picture in a significant way as input devices and will undoubtedly become far more important in the future.

The most significant improvements in today's hardware over that of five or ten years ago has been in the area of organization of the processors. The successful application of the multiprocessing concept has led to a degree of parallel modularity which was unthinkable ten years ago. In addition, compatibility between different sized members of a given computer family has resulted in serial compatibility permitting the use for a given purpose of a machine which has only enough capability to answer that need.

Priority interrupts have become common to permit greater concurrency of multiple operations within a single system. Such features, and many others, have been obtained with no essential changes in the various system components (such as memory, peripheral equipment, and input/output equipment).

It is significant to note that the computers of the early 50s were quite rigidly divided between scientific machines and business-oriented machines. In the late 50s and early 60s this differentiation began to vanish as system sophistication increased. As the cost of hardware comes down, as it inevitably will, this trend will probably continue

in that it will be economical to build additional specialized capability into the equipment.

The new "third generation" computer series of computer equipment being implemented by major domestic computer manufacturers all highlight the rapidly declining differentiation between "scientific" and "business" computer equipment.

Each of the series mentioned below have the following common features:

A group of computers with ascending computer instruction repertoire compatibility.

A "total" instruction repertoire, with complete mathematical (fixed and floating point, single and double precision), business (comparison, data movement, editing), and general-purpose instructions.

A tenfold increase in maximum computation speed over operational computer of the 1960-1964 era.

An instruction repertoire generally different than that of preceding ("non-third generation") equipment.

General provision of assembly language, Cobol, Fortran, and PL/I software languages.

Provision of software packages to operate in batch, real-time, and conversational (time-sharing) modes.

Software and hardware capability for multiprogramming (current DDC environment) and for multiprocessing (multiple central computer processing units).

Provision of software for multiple level execution of mixed tasks, including extensive functional program (subroutine) libraries.

Another major factor inherent in these computer "families" is the modular ability to expand central memory, mass storage, and input/output equipment

to very large size (4,000,000 characters of memory, and many billions of characters of random access mass storage are available in the largest of those machines). They also can be efficiently used in scientific, business, and information retrieval applications due to their high internal speeds, flexible and extensive input/output capabilities, and large internal and external storage capabilities.

These "third generation families" include the Honeywell 200 Series, RCA Spectra 70 Series, IBM 360 Series, the newly announced Sperry Rand Univac 9000 Series, and the Burroughs "2000/3000/6500 Series" computers. The General Electric 600 Series and the "super" Control Data Corporation 6000 Series computers do not possess a comparable business-scientific instruction repertoire and have fixed-word memory organization. For these reasons of incompatibility (although they are otherwise similar), these two series are considered to have somewhat less universal characteristics than the others.

It should be noted at this point that these individual capabilities are essentially within the state-of-the-art today. Memories of almost any given size are limited only by the economy of application. The speed of computation has increased by a factor of one hundred in a ten-year period and is continuing to increase. There is now a capability to handle large numbers of input/output devices. The use of cathode-ray-tube displays, where economy permits their use, is not uncommon. The distinction between the various classes of machines is thus in the manner in which these elements are combined and the machine is organized.

What will be the characteristics of data processing hardware in 1971? One way to hypothesize the development is to extrapolate from the past, comparing the hardware of the 1960s with today's. In 1960, operational computers still utilized vacuum tubes, although development laboratories were building transistorized systems. Bulk storage of data was almost exclusively on tape with a packing density between 25% and 65% of that generally available now. Core memories ranging up to 32,000 words (190,000 characters) were extremely unusual. A typical time to perform addition ran to scores or hundreds of microseconds (millionths of a second), and a computer storage access cycle time of ten microseconds was unusually fast. These general characteristics pertain to computers in general use in 1960, although that was the period which saw the first delivery of such "modern" hardware as the RCA 501 and IBM 7090. Compare these capabilities with those of the later CDC 3600, Philco 2000 (model 212), and RCA 3301 computers. These more recent units have all been in use for at least three years, and each represents capabilities about an order of magnitude better than the machines of six years ago. The equipment available in 1966, such as the IBM 360, CDC 6600, Univac 1108, and RCA Spectra 70 are almost an order of magnitude better again in many ways. Along with this, we have noticed a great decrease in the cost of the hardware. It was not very long ago when the "rule of thumb" cost for core memory was $1 per bit. Today it is down to ten cents per bit. Most of the improvements have come about in the central processing unit; unfortunately, similar quantum improvements have not

been visible in the field of input/output equipment. It is true that disks are competing with tape for bulk storage, and that we have somewhat more sophisticated man-machine transducers than we had five years ago, but there have been no dramatic improvements in the capabilities of input/output equipment comparable to the changes in the central processing units.

a. Internal Memory

The past few years have seen great improvements in the cost, speed and size of internal memories. Most operational memories today are still based on the threaded magnetic core. Storage cycle time has decreased by a factor of about ten from units of microseconds to tenths of a microsecond. Size and cost have also come down by an order of magnitude. In addition, we now see the introduction of even higher speed scratch-pad thin film memories, as in the Burroughs B-8500, which reads a 52-bit word in an average of 250 nanoseconds (billionths of a second).

The United States Navy's Project Lightning has had its greatest successes to date in the realm of development of high-speed internal memories. Three different approaches have been sponsored by the Office of Naval Research, each aimed at an ultimate memory operating at one-nanosecond clock rate. IBM has been working with cryogenic ("ultra-cold environment") memories, on the approach that reduction of resistance at cryogenic temperatures permits increases in pulse speeds. Sperry Rand has been developing thin film memories. The

partial success of this approach is attested by the fact that computers on the market today are utilizing thin film memories. RCA has worked on the continued development of conventional magnetic cores and their descendants. Memories have been built utilizing printed circuitry with 500 cores to the square inch on a substrate less than 10-mils thick with a capability of cycle times better than six nanoseconds. Certainly the two latter approaches (printed core assemblies and thin films) are directly amenable to automatic manufacturing with the attendant savings in cost. Perhaps the greatest present bar to their use is the fact that development has been lagging in the necessary, related fields of high-speed logical circuitry and in system design which can take advantage of these memory improvements.

We can see before us the probability of automatically manufactured memory devices with a cost-per-bit approaching a tenth of a cent and operating at speeds approaching ten nanoseconds. When the associated circuitry, power supply, interconnection, cabinets, etc., are added in, the total cost of the internal memory may be expected to be below one-cent-per-bit. The hardware cost for a memory unit ready to operate will thus be in the neighborhood of ten cents per character of memory. It is obvious that the restriction of cost will no longer apply insofar as the use of larger and larger memories go. Such memory hardware may make today's 4,000,000-character memory as obsolete and inadequate in 1971 as ENIAC's ten-word memory is today.

Although such large-scale computer memories would not completely eliminate the need for external bulk storage, they would permit the transfer of data between external and internal storage in much larger blocks than at present, particularly when combined with economical large-scale, high-speed external memories as part of the input/output bulk storage device. This capability would come about by the fact that it would be economical to assign a significant portion of computer memory to the single task of communication with input/output bulk storage devices. A large block of data from a tape or disk would first be buffered in a high-speed memory in the input/output device, and then allocated at very high speed into the main memory input/output section. This data could then be transferred to other locations in memory, or a method of dynamic allocation of space will permit "floating" the main input/output section throughout the memory. In either case, the transfer of larger blocks of data at higher speed becomes feasible because of the economy of providing additional storage space for a particular use.

The use of very large but inexpensive memory units, together with cheap repetitive circuitry, permits the direct design of an associative or content-addressed memory. It has long been recognized that it is desirable to extract data from memories on the basis of the data itself rather than on the basis of its storage address.

To date, this generally has been accomplished only for small memories. For example, one of the special-purpose scratch-pad memories in the Burroughs B-8500 is an associative memory, in which data content addressing is feasible because of the small size and high speed of the scratch pad. Retrieval is accomplished by simultaneous query of each discrete data item in this small memory, with comparison for the desired information being made in a multiplicity of decision elements. As memory speeds increase, and memory and circuitry costs are reduced, this capability will undoubtedly be extended into larger memory elements. Circuits of hundreds of thousands, or even millions of components are not far off. It will thus be possible, in one computer operation cycle, to search every element of the memory in order to find the desired information. This will relieve the need for complex data arrangement and the complex programming now used for rearrangement or search.

The field of associative memories is so important to information retrieval applications, and specifically to the various functions of DDC, that more detailed descriptions are pertinent. Two distinct component approaches have been proposed and examined for the two physical forms of an associative memory -- cryogenic and magnetic. Development of cryogenic memory cells has included an ability to incorporate and embed logic capabilities in each storage cell. The

intimate relationship of storage and logic provides an inherent storage cell selectivity which is basic to associative memories. The cryogenic computer, however, suffers from the severe disadvantage of requiring a controlled "cold" environment. Power failure to the environmental control system causes loss of memory contents. Significant problems in the utilization and application of cryogenics have been reported due to the unavoidable heat dissipation of logic and switching elements. For these reasons, it is unlikely that cryogenic computers will be generally popular in the near future. The second approach to an associative memory is based on the use of magnetic components. In addition to the Burrough's scratch pad memory mentioned above, there have been reports of development of associative memories as large as 2,000 words with a five-microsecond comparison time. Over the next five years it is probable that the size of associative memories will increase although there may be no significant increase in speed.

The two major approaches to associative memories can be compared as follows:

Magnetic films and cores require high driving power, have low signal-to-noise ratios, and require large amounts of associative component circuitry.

Cryogenic systems require large amounts of power for the environmental control system. They will also require extensive circuitry at the interface between the controlled environment and the uncontrolled environment.

Even if the true large-scale associative memory does not become available by 1971, the combination of very large, rapid access, addressable memories with smaller but significant associative memories provides for a degree of sophistication in content addressing not now available. Great improvements in this area, built into the hardware, can have a significant effect on such software developments now going on such as research into sophisticated query languages. A more brute force approach to content addressing may well become economical as a result of larger core memories, much larger associative memories, and higher speeds of operation.

b. Peripheral Storage

This category of equipment includes such units as tape storage devices, magnetic disk and card units, and drums. Despite the increase in size in internal memories, these units will still be popular five years from today. As the sophistication of use of data processing systems increases, there will be a corresponding increase in sophistication of applications. We can expect bulk storage requirements to increase at a greater than proportional rate as applications become more varied. The fact that internal storage can be expected to grow in size means that it will become more feasible effectively to utilize larger external bulk storage. When such larger storage capabilities are combined with higher transfer speeds and larger internal memories, it becomes eminently possible to use external storage orders of magnitude larger than

today's with no appreciable effect on the speed of operation of the system.

It is true that system speed does not generally or necessarily depend on the speed of the peripheral equipment. Thus, for example, the rate for reading data from a tape does not have to be consistent with the speed of handling the data within the CPU. However, it is not hard to envision operations in which the necessity for frequent referral to input/output equipment slows down the system throughput. There are many ways of programming so as to minimize this penalty, but the point being made here is that such a penalty can also be minimized by the use of more sophisticated hardware, thus removing some of the burden placed on the software. The gain to be achieved from such additional hardware sophistication can either be taken as a minimizing of software requirements or in the ability to handle much more input/output data with no sacrifice in throughput time.

(1) Magnetic Tape

The fundamental basis of operation of magnetic recordings on tape, disk, or drum is sound, and techniques are sufficiently well in hand so that any improvements in these units will be matters of degree rather than of kind. For example, magnetic tape units are available today which record at 800 bits-per-second-per-track with as many as 20 tracks-per-inch width of the tape. We might expect packing densities to increase to

1500 bits-per-inch (or characters) of information, as magnetic heads are improved and new magnetic surface materials appear. This is a simple extension from professional audio tape recording capabilities, which today extend to well over 3000 cycles-per-inch of tape. With multitrack tapes, and closer track spacing, common tape data transfer speeds in the range of millions of characters-per-second will become available, but this would probably provide an increase of less than one order of magnitude. This is hardly significant when compared with the improvements in operation which are to be expected in the central processing units of computers.

There has been a notable trend, particularly in connection with the so-called "third generation" of computers, to standardize on tape format between various computer manufacturers. This standardization is still far from universal. It can be expected that future hardware capabilities will provide a still greater percentage of standardization.

In the past such standardization has been accomplished entirely on the basis of one manufacturer's accepting the specific tape format of another and designing his total data processing system to provide that format. Future developments may permit some of this compatibility to be obtained by programming the computer to recognize a number of different tape formats.

Compatibility of such physical characteristics as reel hub size, tape width, and others may never come about. It may still be necessary in the future to have available tape servo units of different physical characteristics to permit this capability. Far more important, and much more probable in the immediate future, will be both the hardware and software capabilities for control of different types of servos, and for translation between different recording formats. In the past, provisions for the development of such compatibility have not been encouraged by system manufacturers for business reasons. There is now a greater recognition that each will profit by the greater application of more universal data interchange capabilities.

(2) Disks

Magnetic disk recording is now an accepted capability for input/output data storage. Although the storage capability of disks compares unfavorably with that of tapes on a volume basis, this factor is insignificant. Far more important are: access time to a block of data on disk, measured in fractions of a second as opposed to seconds or even minutes for a tape; greater capacity of storage in a reasonable size disk pack as compared to a tape, thus reducing the requirements on the operator to change units, and also making more of the magnetically-stored data available at all times; higher cost of storage on the magnetic disk as compared to the tape.

Considering the last item first, namely, that of relative cost, we can expect no significant difference in the next five years. In the case of both disks and tapes, the unit is essentially electronic-mechanical, and major changes and improvements are not anticipated. The capabilities of a tape unit to provide inherent large-block storage or formatting are equally pertinent to a disk unit. In the next few years, just as at present, the advantages of the disks for many applications will continue to outweigh the disadvantage of high cost.

The larger capacity of the disk, as compared to the tape reel, will continue. Improvements in disk packing densities can also be expected, but to no more significant an extent in data transfer speeds than is the case for tape. We can, however, expect the total capacity of disk units to increase as it becomes more feasible to use more and larger disks per unit. The most significant improvement in disks that can be expected is again reflected by the increased availability of the equipment's electronic and memory capacity. We may well expect to see an effective capability of reading (or writing) on more than one disk simultaneously. This will come about by provisions within the unit to handle more than one block of the data by providing duplication and redundancy of electronic units. Further improvements in the design of the recording

heads and their positioning mechanisms, for which no major breakthrough is required, will result in smaller units permitting approximately the ultimate of one recording head, individually positioned, per recording track. Such a unit is already on the market but is now capable of handling only one surface at a time. Simultaneity will be achieved both by redundancy of data handling facilities within the unit, and by duplication of head control mechanisms.

(3) Magnetic Cards

One bulk storage unit which has recently made its appearance might find a number of applications which are now being handled by more conventional units. This is the Random Access Card Equipment (RACE) used by RCA in its model 3301 and Spectra 70 computer systems.* RACE records data on a broad surface of a magnetic tape-based mylar** card approximately 4 by 13 inches. Sixty-four cards are contained in a single holder and each card is individually addressable and retrievable. The addressed card is extracted, and passes the card reading head at a rate of approximately 400 inches per second. Up to eight (no inherent limitation or number) card magazines can be used

---

* The analogous CRAM (Card Random Access Memory) unit, produced by NCR, exhibits similar generic characteristics.

** A registered trademark of the I. E. DuPont de Nemours Corporation.

simultaneously on one machine. The storage capability per card is approximately 16,000 characters. Each magazine, therefore, contains approximately one million characters of data for a total of approximately 8 million characters per system. The speed of reading provides a data rate of 60,000 characters per second, but is slowed significantly by the (electro-mechanical) card accession time intervals. Total bulk reading time depends on the degree to which one card follows another immediately past the reading heads. Access time ranges from 100 to 500 milliseconds per card. The advantages of RACE over magnetic disks are the simple mechanical facilities for changing a magazine, and the fact that the card is equally adaptable to storage of digital data, magnetic video recordings, or microfilm inserts.

(4) Other Bulk Storage Devices

Several types of bulk storage devices which may some day replace even tape units are now on the horizon. The monolithic ferrite memory is now in the laboratory. In this memory, data is stored throughout the volume of a three-dimensional monolithic structure. First applications of this device will probably be internal to the central processing units for small scratch pad memories, but eventually it gives promise of rapid access to memory of billions of characters of data on an almost random basis. If development is successful, this component could replace present external bulk storage devices, but probably not in the next five years.

Other types of bulk storage devices expected to be operational within five years or less are microfilm-based storage, and bulk-devices utilizing non-metallic mineral elements. These devices will greatly expand storage and retrieval beyond current and announced ("third-generation computer") bulk storage capability.

c. Internal Circuitry

Computer construction has undergone a revolution from large vacuum tubes mounted on hand-wired chassis, to miniature vacuum tubes on printed boards, to transistors on plug-in boards, and we may expect this revolution to continue. We are already seeing applications of the next step in this progression. Modern electronic circuitry, called variously integrated, microelectronic, and monolithic circuits, is now being utilized, which requires a minimum number of operations in its construction. One semi-conductor chip which can be manufactured automatically will replace as many as four active components (transistors and diodes) and perhaps 12 passive components. Today we can buy single integrated circuits in which one element is a complete amplifier or a complete flip-flop. The day is not far off when such a chip will provide the capabilities of a multi-stage register. To date, one of the major difficulties in production of these integrated circuits has been factory yield. Because of low-acceptability yield against a set of prepared specifications, the cost per acceptable unit has been high. But there was a time, not

too long ago, when transistors were $10 each and diodes were $1 each. It is not far-fetched to forecast that the manufacturing cost of a multi-stage register ready to plug into a cabinet will be in the order of one dollar.

Additional economies are also incurred. The minute size of micro-electronic elements permits designs with a larger number of stages per plug-in unit. This reduces the requirement on cabling and permits much smaller cabinetry to be used. Again the cost is brought down to the point where the $1,000 "shoe box" central processing unit can be foreseen.

On the other side of the fence we must note the great strides being made in extending the characteristics and capabilities of the processing units of computers. The Office of Naval Research has, since 1960 or earlier, been engaged in Project Lightning, a program to develop a 1,000-megacycle computer. Although, to date, greater development success has been obtained in the field of memories, much work has been successful in designing circuits to operate at this one-nanosecond rate. The David Sarnoff Research Laboratories of RCA in Princeton, New Jersey, has demonstrated logical elements, such as gates with fan-in and fan-out up to three, capable of handling data at this one-nanosecond clock rate by the use of tunnel diodes. To date there has been no large-scale combination of these units to accomplish anything more than trivial single functions such as small

registers. A not insignificant problem arises from the fact that the velocity of propagation of a data pulse along a wire is very close to one nanosecond per foot. The physical design of combinations of these elements, capable of performing useful functions, is perhaps a greater mechanical problem than the electrical or logical design of the units. This physical design problem is now being attacked: computers are being used for laying out the configuration which will permit operation at these high speeds.

We probably cannot expect a one-nanosecond central processing unit in the next five years, but it is not unreasonable to expect 20-nanosecond systems. This will represent an increase in speed of at least five over some of the most advanced systems announced to date, such as the CDC 6800.

Cheaper, smaller and faster circuitry in central processing units permits the introduction of greater parallelism in data processing at economies which are unapproachable today. These advantages could also lead to the design of computers with hardware characteristics utilizing circuit redundancy which is unheard of today because of cost. Certainly, more capability for the same dollar and cubic inch could permit inclusion in input/output equipment of logical functions which are today relegated to the CPU. Multiprocessors become more attractive and could lead to equipment capable of economical, simultaneous performance of many functions which must, today, be performed serially by such means as time-sharing.

The new circuitry components not only have advantages in cost, size and speed but represent a new level of reliability which has not been approached heretofore even with solid state elements. It is true that transistors and diodes have life times or mean times to failure running into many thousands of hours. In fact, one readily available transistor meets a production inspection requirement of not more than three failures per billion hours of operation. The unreliability of today's circuit occurs in those portions of the circuitry which are not part of the discrete elements. These include solder joints, poor planting on conductors, and connectors. New circuit elements will reduce the need for these and other less reliable portions of the data processing hardware.

2. PERIPHERAL INPUT/OUTPUT EQUIPMENT

Peripheral equipment is considered here as that equipment which transduces or communicates with the operator rather than that which is used entirely within the system without relation to the external world. All such peripheral equipment has the requirement of the conversion of electro-magnetic signals or states into a form recognizable by human senses. Equipment used internally may convert from one electro-magnetic state to another but need not present the information to a man.

a. Punched Card Equipment

To date, the most commonly used transducers, both input and output, involve the use of discrete alphanumeric characters. The fundamental operation of the input process almost invariably involves a keyboard.

(Note here the implied exclusion of such real-time on-line data processing systems as those involved in the handling of radar data, or manufacturing process control.)

The fundamental keyboard operations have usually involved the preparation of punched cards for data and programming input. Punched cards have the advantages of being simple and flexible. They permit permanent storage of data and are amenable to checking for errors, and in the case of an error, correction can be made simply by substituting a correct card for an incorrect one. On the other hand, use of cards is expensive, although the dollar cost of an individual card, or even the cost per bit of the materials used, is low. The expense comes about due to extreme slowness in processing. An experienced operator cannot be expected to provide any more than three 80-column cards per minute. Error checking cuts this rate in half. The first large increment of expense, therefore, is the cost of labor per input unit. At an overly conservative cost of labor of $60 per week plus a 67% overhead, this is approximately 1.8 cents per character punched, and an equal amount for verification. If this cost per character seems low to the reader, it should be noted that this represents only 7,200 cards punched in a 40-hour week.

The handling of the cards as input data is also slow. Modern card reading equipment can handle up to 1,000 cards per minute, but even

this is only about 1,000 characters a second, a speed three orders of magnitude slower than the capabilities of the data processing systems handling the data. If we assume that one of the great potentials of data processing systems is their ability to handle more data in less time, it is obvious that punch cards are not the final answer to data input methods. Nevertheless, we can anticipate that punch cards will be popular, at least for the five-year period under consideration.

The U. S. Department of Labor estimates that the increased use of data processing systems, which might be expected to result in a greatly increased demand for keypunch operators, will be balanced over the next five years by the introduction and acceptance of other devices for inputting data such as optical character readers. They further estimate that the need for keypunchers will decrease beyond that period. The flexibility and comparative low skill required for operators of card punching equipment will result in their continued use.

Can we expect improvements in the preparation and handling of cards? It is true that keyboards may be improved through human engineering so as to increase the speed of operation slightly. It is also true that we might expect some slight increase in the speed of handling the cards in the machine. However, as long as the latter remains essentially a mechanical function, we cannot envisage an improvement

of speed by a factor of 100 to meet even today's requirements. In essence, the use of cards as an input medium will continue but will diminish in importance.

There have been numerous attempts to overcome the deficiency of electro-mechanical punching of cards. One approach has been the use of heat or spark to burn a controlled hole in a card. The major obstacle to this method seems to be the much higher cost of the specially treated card. Another approach places an electrostatic charge at a specified point on the paper. The use of this method obviously requires special sensing equipment for reading, and the marks on the card are not visible without another processing step. Developments of new keyboards which do not call for the mechanical manipulation of key depression have also been demonstrated. The theory here is that relief of operator fatigue will result in higher speeds. One such keyboard uses an electronic signal generated by the body capacitance of a finger touching the fixed key -- similar to some modern elevator call buttons.

Some of the speed deficiencies of card reading are overcome by reading the cards directly onto magnetic tape in an off-line mode. The CPU is, thus, not involved in the very slow process of reading the cards. With present systems, the tape record so generated can be formatted only by a second run through the CPU. This run is, of course, made at a much higher speed than that used when reading cards. This mode

of operation can be expected to increase in popularity as the sophistication of the tape unit increases as a result of cheaper electronic components.

In the past there have been attempts to overcome the speed deficiency of card handling by connecting the keyboard directly to a magnetic tape. Although this method bypasses the cards, it is not essentially different from the preparation of cards and the subsequent off-line recording from card to tape. This is a common practice today. It permits the transfer of data to the central processing unit at much higher speeds since the data input now comes from tape instead of cards. We can expect greater self-sufficiency of tape units to increase the use of this mode of operation. However, note that the transducing device is still essentially a keyboard.

b. Other Input Devices

There are many other input transducers in use today and envisioned by engineers. Such examples as the light gun or light pencil, the RAND Tablet, or special measuring equipment used in such applications as photo interpretation, definitely have roles in providing data to a data processing system. At present this type of equipment is generally many times more expensive than alternative methods of providing the data, at least in terms of initial cost. There is little likelihood that this will change. Furthermore, such devices are usually quite slow for large masses of data. Their places in the

future are assured for special purposes, but it is hard to envision large-scale application of devices whose initial cost and upkeep are many times those of alternative, more cumbersome methods.

Still further off are more exotic devices which have attracted the popular fancy and even a considerable amount of developmental effort. Foremost among these is the receptor and encoder of spoken data, together with its converse, the vocal output device. These devices are considerably more than five years into the future. Furthermore they are essentially slow. Human speech is not very fast. The fastest rate at which one might read a column of digits is about six per second. For data more complex than simple numbers, not only is speech slow, but the information content is low because of high redundancy.

c. Optical Scanners

The only truly significant change in methods of providing inputs of large blocks of data to information processing systems which has come along in the last few years is the optical scanner, variously called page reader or character reader. The machines (which are now reaching a level of capability and reliability suitable for application) recognize (machine) printed characters on a sheet of paper at operating rates exceeding 3,000 characters per second. Although equipment is available today to perform this function, most such devices are limited in flexibility of one type or another.

The simplest, and earliest, print readers, operated with single fonts, and were capable of recognizing the 26 letters, 10 digits, and a restricted number of other characters such as hyphens. Furthermore, they were designed to handle forms of a specified size and had no selectivity as to the characters which were read. All characters on the page conforming to the font requirements were recognized. These machines have great application in many fields where there is rigid control over the preparation of the material to be read. Examples include credit card charge billing and other turn around uses. Although they are the most economical to purchase and use, they have little application in the general field of information processing.

Improvements on this early class of print readers included flexibility in the paper handling portion of the system and an increase in the number of recognizable characters. A single machine today is capable of handling documents which range in size from 3x4 to 14x14 inches.

Other machines have been designed to read as many as 14 different font cases. (In this regard it should be noted that the upper case set of characters and the lower case set of characters constitute individual fonts.) Further improvements have permitted a pre-editing function to specify by underlining, enclosing in parenthesis, or the use of a colored pencil, those portions of a page which are not to be read. Thus the standard character readers have been improved

so as to be able to handle without confusion pages which contain mathematical or chemical formulae, diagrams, etc. Machines of this type are in use today.

At least one equipment is being designed to operate under the programmable control of a small on-line computer. The great advantage which occurs in this approach is the capability of specifying to the machine those portions of a page which should not be read, or calling into play different recognition logic for various fonts or characters. Although such machines are not yet in general service, they offer a significant advantage for large-scale input of printed information when documents are obtained from many sources in varying fonts and alphabets. Furthermore, the logic of the machine permits great variations in the size of the characters and the skew of the printed lines with respect to the paper edge. This latter point is one which has invariably required close control in the past. The new machine now under development, instead of being controlled by an on-line computer, includes provisions for programming of the recognition logic and field of view as an integral part of the machine. Furthermore, it will have the capability to recognize identification marks on the page and thus call up the program needed for reading of that particular paper.

The most serious deficiency to date, but one which is to be overcome as print readers come into more general use, is the lack of

consideration of the effect of the print reader on the total information processing system. Character readers have invariably been considered by both the developer and the user as additional peripheral equipment which was hung on an existing system. There has been little systematic determination of the effect of the increased input capabilities on the information processing system and the environment in which the system operates. The ability to get large blocks of input data rapidly and economically seems to be basic in any operation which performs automatic abstracting or extracting. However, little work has been done for a given application of such aspects as acceptable error or reject rates, the effect of an error or reject on the abstracting or extracting function, or the point of cross-over of economic use of a print reader versus other means of input.

d. Output Devices

Historically, output devices have provided information in character readable form. The computer controlled electronic typewriter was perhaps the first example of this equipment and is still useful where information output requirements are small and slow. More important in the handling of large amounts of data is the high-speed line printer. A typical unit provides 120 characters per line at a maximum speed of 1,000 lines per minute or a maximum of 2,000 characters per second. Note that this is still two orders of magnitude slower than the handling of data within the computer. So long as the high-speed printer is constrained to electromechanical

operation, we cannot expect gross improvements in its operation. Such changes as going from 120 to 160 characters per line is hardly sufficient. The problem of speed in the use of the high-speed printer really involves limiting the computer speed to that of the output device. Today we get around this limitation by having the computer record the data to be printed on a tape at a higher speed than could be handled by the printer, and the printing is done off-line.

An alternate method in large-scale systems involves the use of an intermediate buffer computer of smaller size than the main CPU. This buffer computer is, itself, generally a full data processing system. To the extent that it has general-purpose capabilities, its use is less than optimum. If it is considered that only a special purpose is required of the buffer computer, cheaper components will permit economy in building in special capabilities as an integral part of the printer.

Faster high-speed printers have become available in the last few years which depart from the electromechanical means of operation. The Stromberg Carlson (ITT) Model 6600 printer is an example of an all-electronic device. This equipment utilizes a cathode-ray tube which generates the characters. The printout is then transferred to light-sensitive paper to provide the final printing. This equipment now operates at 3,000 lines per minute, or one-half order of

magnitude faster than the best electromechanical systems. To a great extent the speed is limited by the sensitivity of available, easily handled paper. It is not beyond credence to believe that improvements in display devices such as cathode-ray tubes, fiber optics and better paper, can increase this speed to 10,000 lines a minute. One of the present disadvantages of this device is the fact that only a single copy is available. It is obvious that the one master copy could be automatically duplicated on the fly by electrostatic duplication methods such as Xerox equipment. To illustrate the speed of this device, we should note that the present capability of 3,000 lines a minute, 120 characters per line, is the equivalent of 20 pages of a typeset book per minute. The output needed to keep such a device in use and thus economical for extended periods of time are truly formidable. Nevertheless, we can anticipate greater application of such printout devices.

e. Document Reproducers

Not all output requirements are satisfied by bulk data outputs. Many other requirements appear. Many applications, particularly those involving legal or governmental matters require copies of a document as opposed to the data on the document. On the other side of the fence, we have the requirement for output of summarized, plotted or pictorial information. As equipment becomes more available to provide these facilities, we can anticipate that their use will increase.

It is obvious that to provide copies of a document, there must be images of the document stored within the system. If the user knows exactly which document he wants to copy and where it is stored in the system, this is hardly a sophisticated data processing requirement differing from that of any other addressable retrieval system. If it is necessary to retrieve a document on the basis of its contents from some unknown position, it is necessary to store not only the image of the document but such information as is required to retrieve it out of its location in the document memory. It is not unwarranted to consider this as two separate retrieval problems: the first to find, on the basis of the query, where the desired document is stored; the second to retrieve and reproduce the document itself. The first of these problems is similar to the problem of retrieving information from memory and printing it out in any form whatsoever on the basis of the memory content.

Document retrieval (as opposed to information retrieval) has long made use of the storage of photographic images of the desired document. The Walnut system built by IBM for the intelligence community provides the capability of retrieving documents from a photographic storage of many millions in a matter of seconds. The two-state photographic process used in the reproduction cycle cause it to be comparatively expensive and it is unlikely that this system will find widespread use. On the other hand, it is a random

access system. Various other microfilm retrieval systems provide some degree of random access. If requirements can be batched, the equipment requirements are reduced and the cost for retrieval comes down.

A recently announced document retrieval system is the Ampex Video File, the first of which has recently been sold to the Southern Pacific Railroad. This system makes use of magnetic recording of images on tape very similar to the standard television tape recorders used in broadcasting. The printout is by means of an electrostatic printer. The Video File offers the great advantage over photographic storage of the possibility of purging, rearranging, and updating the file. Descriptors and addresses are coded into linear tracks on the tape to permit retrieval either by address of the desired document or by sequential search of the transcription.

Each legal size document occupies approximately one and one-half inches of length on a two inch width of video tape. A digital track on the side is used for encoding and timing digital data. Search speed of about 1,000 inches per second of tape or 500 documents per second is realized. There is also a three second delay to stabilize and synchronize the system after the desired document image is reached. The first system is costing the Southern Pacific Railroad approximately $600,000 plus $40,000 for each video tape unit. The Video File lends itself best to sequential batch retrievals. In

those applications which can use documents stored consecutively on the tape, output prints can be obtained at the rate of two per second.

Many of the requirements for document retrieval will be more sophisticated. It is conceivable that the inquirer will not know what document is desired but will want rather to search some kind of abstract file to find out whether or not a suitable document is available. This is the typical library search problem. If it is assumed that fast retrieval, perhaps in the form of abstracts, will become available, then the document retrieval equipment becomes another unique type of peripheral equipment used in data processing systems with particular applications.

f. Cathode-Ray-Tube Devices

Many applications have dynamic requirements for display. In these applications, the presentation of one picture leads to the introduction of commands for a change in the picture. For example, computers are now being used to provide engineering drawings, including 3-views, isometric projections, and dynamic rotation to provide 3-D effects. With such a display, the engineer may see material presented which he desires to change. When the new data are provided, he wants a new picture. In this case the output display is just one element in a feedback system involving transducing from machine to man and back into the machine again. To date cathode-ray tubes have been

used exclusively for this purpose. No appreciable change is anticipated in this application. Improvements in cathode-ray tubes in such areas as better resolution, greater light output, or smaller equipment requirements are changes in degree rather than kind. We can, of course, expect the capabilities of the output display system to increase as it becomes more economical to incorporate greater amounts of storage and data handling facilities within this peripheral equipment rather than in the CPU. Although this may have the important effect of relieving the CPU of some load, the major effect will be one of economic implementation of the entire system, including the very high-speed central processor plus the output transducer of greater capability.

The storage requirements for cathode-ray-tube display need not be external to the cathode-ray tube. Many excellent high-resolution, direct-view storage tubes have been produced. Some models even provide selective erasing of areas of the tube specified by the operator. These storage tubes have even greater incompatibility than other cathode-ray tubes with the prevailing direction of design of modern computer units. Their reliability is low, their cost is high and the voltage requirements for operation tends to get higher and higher, while voltage required for solid-state components is getting lower.

It is not inconceivable that solid-state, direct-view display devices with selective erasure characteristics will be available in a few years.

Such devices, utilizing electro-luminescent techniques with or without internal storage are now being produced in the laboratory. Their rate of development has been slow and, while they may be available in 1971, it is not anticipated that they will be in very widespread use at that time.

The speed abilities for input and output equipment have been described here as being universally slow compared to the speed of operation of the remainder of data processing systems. Whether or not there is a necessity for an input or output device to handle data as fast as the CPU, depends on the application of the system. In some business applications the output data is repetitious in part. Unless an off-line mode is used, the speed limitation is that of the output equipment. In many scientific applications the amount of internal processing of data is very great compared to the requirements of input or output. In this application, slow peripheral devices would not be as damaging.

3. FUTURE HARDWARE-SOFTWARE RELATIONSHIPS AND POTENTIAL

All of the wonders of improvements in components and elements are somewhat meaningless unless new concepts of system organization are developed to make use of these concepts. One major concept of system organization is the ability to allocate functions between hardware and software in an optimum and economical fashion. It has long been a "rule of thumb" that the software for an operating system costs as much as the hardware. This

rule is not now as generally applicable as in the past. New, more exotic applications of data processing systems require far more software than more conventional uses. The cost of a skilled laborer of the data analyst and programmer type is increasing. For such new systems, costs can be expected to go up. On the other hand, more conventional business and scientific applications can make greater use of software systems supplied by the hardware manufacturer as part of the library package. It is not clear what effect this decrease in requirements for special software will have on the total cost of the system as a percentage of hardware costs when it is noted that hardware costs will also decrease. Even if hardware costs were zero, we can expect the total cost of a system designed with today's concepts to be greater than 50% of today's costs. It is necessary, therefore, to consider the effect of cheaper hardware capabilities on the requirements for software. It is an oversimplification to say that these cost trends in opposite directions indicate that we should relieve the software of many of its responsibilities by trying to build them into the hardware. There will certainly be many occasions and applications when it is most desirable to simplify the software requirements as much as possible even at the cost of more complex hardware. This depends on the application and the best way of meeting the user's needs. However, there will always be special requirements which can be met only by software of a sophisticated nature. Software will always be with us. It is probably true that greater hardware capability per dollar will have more effect on the level of

sophistication of the software needed for the more mundane applications than on the need for software per se. As the range of applications increases, a higher level of software will always be required for the newer and more exotic applications. More and cheaper hardware capability will tend to have a major affect in opening up these new fields. It is perhaps well to consider the hardware subsystem capabilities which could be used to advantage to relieve this need for sophistication. Some of the more obvious of these hardware characteristics, such as larger internal memories, more parallelism of internal operations by the use of cheaper and simpler electronic components, and higher speeds have already been mentioned. It is important to consider the possibilities of combinations of these improvements.

It would be impossible to catalogue all improvements which might be expected, but some of the following might be of interest. Certainly, if the capabilities which have been discussed for the various parts of the data processing system are considered to be feasible, then the systems described below represent only some of the capabilities which could be obtained by suitable system design. Selection of these has been made to illustrate specific points rather than to be exhaustive in describing all possibilities.

a. Peripheral Memories

Input/output devices can be combined with inexpensive data handling and memory facilities in one unit. No longer will it be necessary

to transmit keyboard data one block at a time. Data from these facilities could be formatted and stored in the input/output equipment until there is an accumulation of enough data to transmit to the central memory in large amounts at very high speed. Data transfer could thus be accomplished with less interference to ongoing computer operations. This concept is being approached today in many systems by the use of a smaller general-purpose computer for input/output buffering. A program is, of course, required to permit this smaller computer to be applied in this comparatively special-purpose function. Since a given input/output device is essentially a special-purpose piece of machinery (in the sense that the smaller buffer computer is a general-purpose device), economy can lead to a wired program for the input/output equipment. At first a tape unit might provide facilities only for buffering either input or output in very large blocks, but it is not difficult to extend this wired programming capability to specific formatting changes or even minor searches simultaneously with ongoing operations in the CPU.

b. Multiprocessor Systems

Much has been heard recently about the possibility of multiprocessor systems becoming popular. There is no accepted definition of the term, but generally it is considered that a multiprocessor system is one which consists of a number of individual computers, each capable of performing on an independent program, but linked together in some way that permits cooperative operation. Implied in the

meaning is the fact that the multiprocessor system is capable of handling problems too large to be handled efficiently by any one of the modules. If a given application requires occasional facilities greater than any one module, then the multiprocessor concept becomes attractive. If on the other hand, the major use of the facility is for these large problems, then a larger, more powerful system should be considered. The most promising role for multiprocessors is in the application in which the usual mode of operation requires individual independent use of the modules. This work, of course, implies that there are a number of users, although the users need not be physically distinct. The system arrangement is such that should any problem exceed the capabilities of one module, other modules are called in for aid as required. This, in effect, provides the capabilities of a single, more powerful system when and only when the program places this requirement on the system. When the second module comes to the aid of the first, it is necessary that it do so without losing its own place in the program or its data. When this function is performed by systems such as the Burroughs B-8500, it uses a fairly slow transfer of present status by one computer to an auxiliary storage under program control.

A second operation of a multiprocessor system involves the use of a common data bank including high-speed memory, shared by a number of separate CPUs each operating on its own program. Here, too, there is a complex programming function to assure that each CPU is

aware of data location within the memory and to prevent simultaneous challenges by more than one CPU. The effects now being obtained by programming can also be obtained by means of hardware system logic with the establishment of a hardware priority interrupt system.

This facility has existed in a limited degree in a number of systems (e.g., the IBM 7090), but has always been under program control because of the limited number of priority levels available. The only hardware requirements (in terms of elemental components) are for separate registers for each program to keep track of the point in the program being worked on, and a hardware system to determine the highest priority requiring service at any moment. A multiprocessor system serving a large number of users thus requires at least one extra address register for each user. Modern computers which have as many as three address registers are considered complex. Systems which incorporate a program look-ahead have additional registers, but these are not used for the ongoing instructions. In any case, a dozen such registers is considered to be a large number. In the future, there will be little reason to place any limitation on the number of registers available to a machine when each register is essentially a single monolithic chip.

c. Larger Address Size

As memories get larger, the ability to address a given location must keep pace with the size of the memory. Present approaches

such as paging again involve program control. As an alternate, the address logic unit size in the machine can get larger to provide additional bits for more memory addresses.

Without program control, the "brute force" approach to addressing requires a sufficient number of bits in the unit or combination of units used for addressing so that each location in memory is uniquely defined.

The imposition on address unit or instruction operand length due to memory size is, of course, completely independent of the memory length required to handle the data in a given problem. Data resolution requirements are independent of the hardware characteristics. However, if the hardware provides capability to handle longer addresses, it would be foolish not to make use of this capability when desirable, even if not required. Larger machine address capability will thus probably lead to more common use of wired double precision half word and floating point arithmetic operations, simply because of the availability of capacity to handle these functions.

Not all of these capacities, nor any other which might be imagined by the reader are advantageous in all applications. Just as we see today a number of classes of data processing systems, ranging from the "personal" computer, such as the G-15 to the monstrous combination of multisystems, we can expect this profusion or spectrum to be

propagated into the future. It is probable, however, that there will be a more definite two-dimensional arrangement of characteristics. Not only will systems continue to be classified according to speed and memory size, but also a new set of classifications may well come into acceptance describing most economic fields of application.

d. Future Classes of Systems

It has long been recognized that the gross difference between the scientific machine and the business machine is that the former performs more arithmetic operations with less data while the latter handles more data in simpler fashion. The scientific user will require less storage and higher computational sophistication. Furthermore, since his problems are generally less routine and repetitive, the scientific user may well have the greatest need for software as a continuing requirement. As was earlier mentioned, software can be expected to increase as a percentage of total costs of operation of the data processing system. The scientific user will, therefore, place the greatest demands on the ability of the machine to handle sophisticated language and thus reduce programming effort. This sophisticated language requirement is oriented more toward providing the scientists and user with the ability to prepare his own program in easily understandable language, making use of a maximum library of macro programs which might well be supplied by the hardware manufacturer.

The business user who today is considered to have the requirements of a large memory will have no less need for memory. However, since his operations, such as payroll preparation, are repeated over and over, in the programming sense they are more routine. The ongoing programming task will, thus, be less in his case than in the case of the scientific user. He will want to take advantage of many economies in a machine model which provides less arithmetic capabilities. He, more than the previous user, will consider his programs in the light of a capital investment rather than a continuing operating cost, and thus can afford an additional initial programming effort if this additional cost is compensated by economies in subsequent use.

The information retrieval system might possibly be thought of as a business application. Undoubtedly its memory requirements will be the largest of any application. Thus one of the most important considerations in the use of a machine for information retrieval is its ability to handle large memories including multiple peripheral systems. The direction being taken to apply data processing systems to the tasks of information retrieval by the development of software has, as yet, failed to produce the general-purpose concepts required. On the other hand, expected developments in hardware will probably simplify the requirements of software for information retrieval functions. It is perhaps in this field where there should now be

the greatest consideration of future hardware by software developers and vice versa.

The opposite argument could be made that there has been a greater tendency in recent years for the general-purpose machine to be non-differentiable between different applications. Its orientation to business or scientific implications has been determined more by the user's selection of input/output equipment and programming than by inherent characteristics of the CPU and memory. Although this leads to some inefficiency in the use of equipment, since it is conceivable that inherent hardware capabilities would never be used in a given application, the anticipated low cost of hardware would not preclude economical inclusion of unused capability. Whether machines themselves become inherently different for different classes of applications, or are made so by selection of modules which are assembled to provide different system characteristics is irrelevant. When the total system is considered, it is probably valid to consider the fact that a business system will be different from an information retrieval system and from a scientific system.

C. STATE OF THE ART IN SOFTWARE

1. PROGRAMMING LANGUAGES

There are literally thousands of different languages in current use for communicating with digital computers. This fact by itself is perhaps the main indication of the immaturity and primitive nature of the computing field, which is only about twenty years old.

To make any sense at all of this babel, to do any comprehensive sort of survey, or to detect any kind of trend, it is necessary to define and classify various computer-language types. An example of existing definitions are the following taken from the authoritative IFIP/ICC Glossary:

> COMPUTER INSTRUCTION or MACHINE INSTRUCTION -- an instruction that specifies a computer operation.
>
> PROGRAM -- a general term for a specification of a process to be performed on data, e.g., an ordered collection of instructions, and other data associated with the instructions. (Note: In general, the data to be processed must satisfy technical conditions, but the program and the data to be processed are otherwise independent).
>
> PROGRAMMING LANGUAGE -- an (unambiguous) language intended for expressing programs.
>
> COMPUTER LANGUAGE or MACHINE LANGUAGE -- a programming language the instructions of which are computer instructions only. See Note to problem-oriented language.
>
> COMPUTER-ORIENTED LANGUAGE or COMPUTER-DEPENDENT LANGUAGE -- a relative term for a programming language requiring translation of a low degree of complexity. A program in such a language will normally run efficiently on the relevant computer,

but may require extensive translation for another computer. See Note to problem-oriented language.

COMPUTER-INDEPENDENT LANGUAGE -- a relative term for a programming language which is not a computer language, but which is, however, intended to be translated, in normal practice, to a variety of computer languages. See Note to problem-oriented language.

PROCEDURE-ORIENTED LANGUAGE -- a relative term for a computer-independent language especially convenient for expressing a process in terms of procedural steps. Examples are Cobol, Fortran, and Algol. See Note to problem-oriented language.

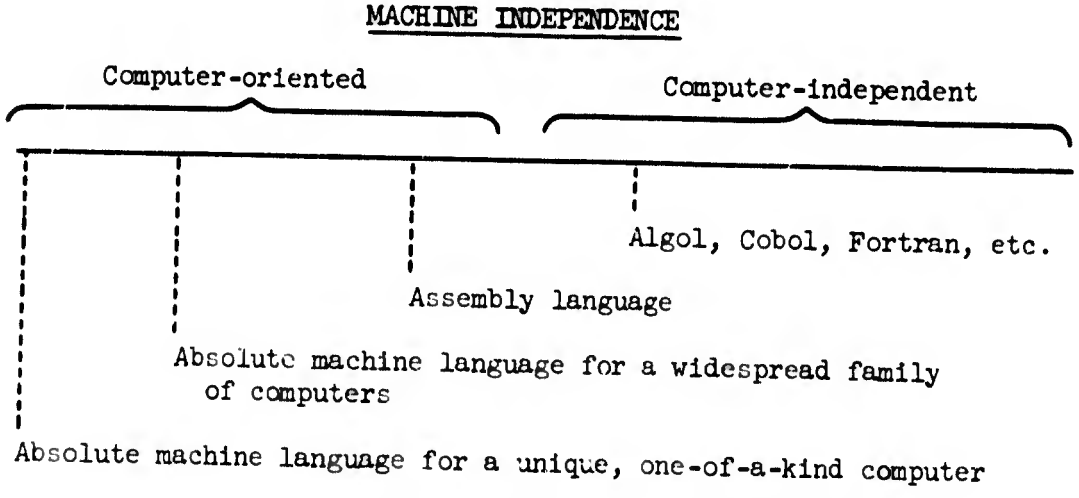
PROBLEM-ORIENTED LANGUAGE -- a language designed for convenience of specifying a class of problems, for example, algebra for specifying mathematical problems. Note: The concepts of computer language, computer-oriented language, computer-independent language, procedure-oriented language, and problem-oriented language are not all mutually exclusive.

From these definitions, we see that a programming language is a language intended for expressing or preparing the specification of problem-solving process to be performed on data. And we can recognize several ascending programming language levels: computer languages; computer-oriented languages; procedure-oriented languages, and problem-oriented languages.

With regard to the categorization of programming languages, the commonplace notion of ascending levels -- from computer-oriented to procedure-oriented to problem-oriented -- is too simplistic to be of

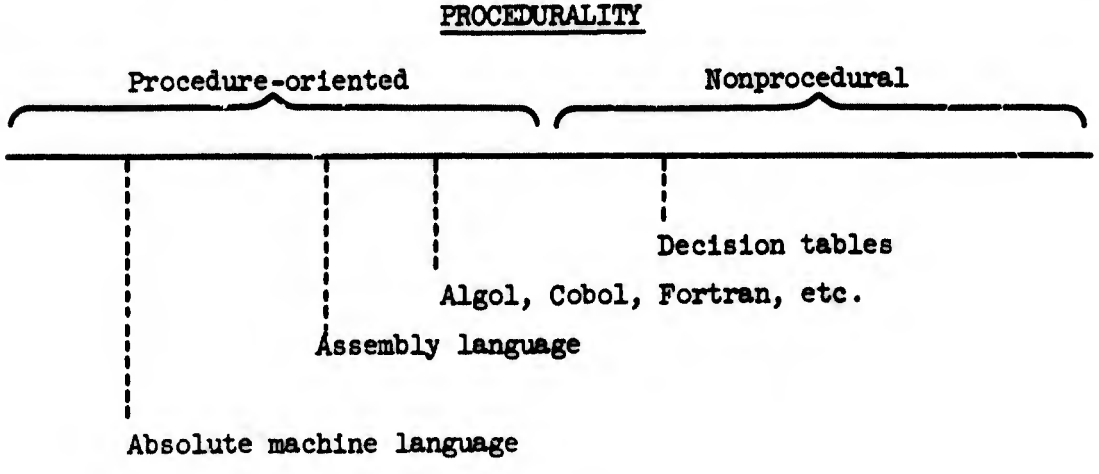
practical use. The comment that these language types "are not all mutually exclusive" does little to clarify the relationships among them. There is a general lack of recognition that we are not dealing with three levels of a single scale, but with three separate, albeit related scales: machine-independence; procedurality; and generality.

The first scale is labeled "machine-independence," and, in effect, measures the ease with which a program written in any given language can be transferred among different computers. At one end of the scale are the computer-oriented languages, and at the other end are the computer-independent languages. Nobody has ever succeeded in objectively measuring the machine-independence of various programming languages. Yet there is wide agreement among computer professionals about the relative machine-independence of major programming language groups. The following chart illustrates this agreement.



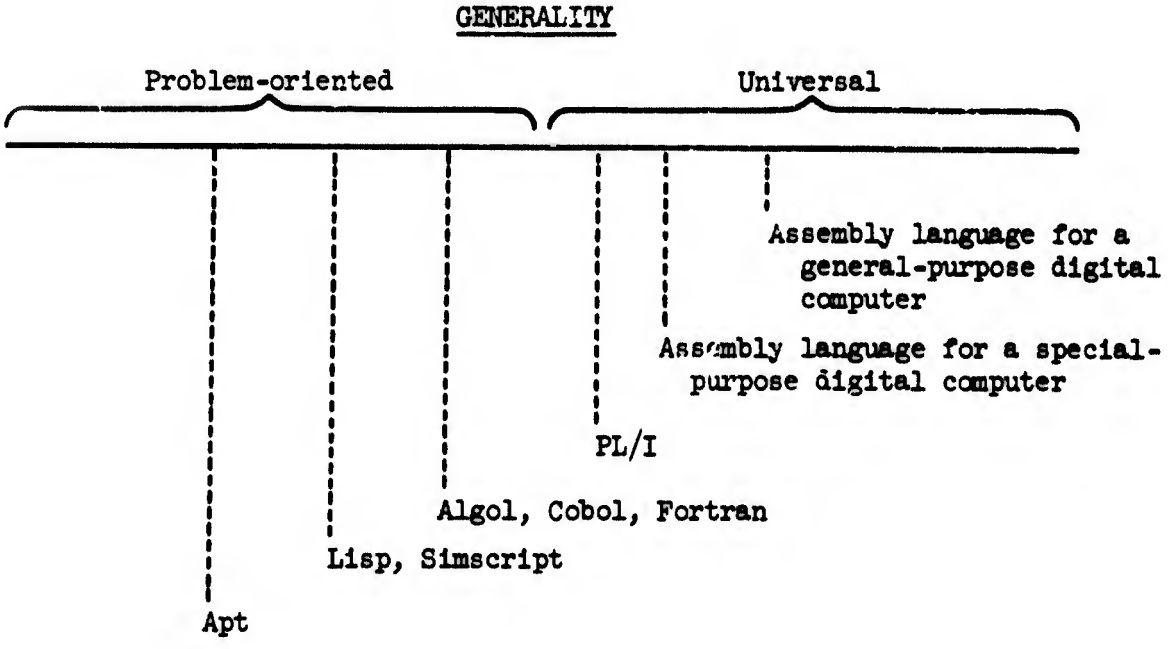
The second scale is labeled "procedurality," and measures the number of distinct, elementary steps needed to express any given computation. Procedurality is a difficult thing to measure; in fact, for some languages, it is hard to determine just what constitutes an elementary step. However, procedurality is an important if immeasurable attribute of a programming language. At one end of the procedurality scale are the procedure-oriented languages, and at the other end are nonprocedural languages.

Although procedurality is difficult to measure, there is wide agreement among computer professionals about the relative procedurality of various programming languages, as illustrated in the following chart.



Computer programming languages, whatever their position on the following scale, can be applied to most problems, provided appropriate programming techniques are used. Therefore, the criteria of effectiveness and convenience, though largely subjective, are indeed

necessary. This makes programming language generality a subjective, but effective and convenient measure. The consensus of opinion among computer professionals with regard to programming language generality is illustrated in the following chart.



Division of the machine-independence scale into computer-oriented and computer-independent halves, the procedurality scale into procedure-oriented and nonprocedural halves, and the generality scale into problem-oriented and universal halves provides for eight programming language categories. Not all of these are relevant, however, to the current state of the art in programming. In addition, other programming language criteria, (such as type of user, type of application,

mode of use) yield further categories. Nevertheless, the terms describing machine-independence, procedurality, and generality, will be used subsequently to characterize various programming languages, along with additional terms to be defined.

A programming language is essentially a language, with various attributes, that permits the specification of a variety of different computations. But why are special languages for writing programs needed?

It is true that digital computers can only do a limited number of rather elementary operations, and it does these operations on operands, or instruction-designated parts of the data stored in its memory. Each of these operations is specified by a unique code, or expression, which must also be stored in the computer's memory. Such expressions are called machine instructions, and the set of machine instructions, along with the set of acceptable operands, makes up the computer's machine language.

A program in machine language is represented inside the computer as a list of numerically coded expressions. People just aren't very comfortable with numbers. They would much rather use names or other meaningful symbols. This indeed, is the basic motivating force behind the development of all these hundreds and hundreds of programming languages: the substitution of names or symbols for numbers.

Since this is the basic motivating force, then, we ought to look at it more closely. What are the advantages of using names instead of numbers?

- names conceptually or functionally related to computer operations, processing operations, and data designations can be easier to read than (often lengthy) numerical-form instructions.
- such names are easier to write (in programming), due to elimination of mistakes occurring in writing of numerical coding. Also, with other than machine-language, a short expression may correspond to pages of machine-language code.
- changing of partially- or completely-written programs is made much easier through use of names in place of numeric machine code.

When you realize that a programmer spends most of his time, not reading his program, nor even writing it, but making changes to it, it becomes clear what a tremendous advantage it is to use names instead of numbers. You can even look at the continual evolution of programming languages as a process of learning how to use fewer and fewer names for more and more numbers, in writing computer programs.

Now, of course, there are also disadvantages to the use of symbolic names instead of machine-language numbers. For one thing, the programmer has to keep track of the names he is using, and be aware of their machine-language definitions, to avoid ambiguous or circular definitions. The use of fewer and more powerful names makes it easier

for people to use the computer without fully understanding what they are doing -- often a mixed blessing. But the main, practical disadvantage results from the fact that computers cannot run on names -- they need numbers. So, if you write your program using symbolic names, these names have to be transformed into numbers before the computer can use them. Fortunately, transforming names into numbers according to prespecified rules is something a computer can do very well indeed. So at least the process can be automated. But, it does take extra time on the computer. And, where the transformation is complicated enough, the question of quality of translation arises. A poorly translated program can use considerably more storage space and execution time than might otherwise be required.

It must be emphasized that this transformation -- from the non-numeric expressions of a programming language to the computer instructions of the machine language -- is logically a separate operation from running the transformed program on the computer. Such transformations are done on the computer under the direction of a class of computer programs known collectively as programming language processors.

Considering for the moment only programming language to machine language transformations, there are basically just two major types of programming language processors, corresponding to two different modes of operation. A program that transforms to machine-language and

executes each source language expression before transforming and executing the next one is called an interpreter. A program that converts the entire program, storing it in a machine-language form for later execution, is called a translator. Interpreters are usually easier to build than translators, and they are also better if you are more interested in the inner workings of your program, rather than just in its operation. Translators, on the other hand, usually use the computer more efficiently, since, with a translator, source language expressions are transformed once and for all, with an interpreter, they have to be transformed each time the computer executes them.

Translators are commonly divided into three further categories: Assemblers, compilers, and generators. Commonly, this division is thought to be according to the type of programming language processed: Assemblers translate computer-oriented languages; generators translate problem-oriented languages. This is ordinarily the case; yet strictly speaking, the distinctions here are misleading, since most assembly languages are more procedural than most compiler languages, and some compiler languages are more specialized than many generator languages. Consequently, a better criterion for distinguishing categories of translators would be the type of translation they perform.

In discussing program translation and its categories, we are concerned with the correspondences between the symbols, expressions, sentences, and messages in the source programming language and the

symbols, expressions, sentences, and messages in the target computer language. We shall discuss here the kinds of translation commonly performed by assemblers, compilers, and generators.

An assembler operates essentially on a symbol-by-symbol basis, translating symbols in the source programming language (source symbols) to symbols in the target computer language (computer symbols). Each source symbol encountered by the assembler results in a relatively simple assembly action -- usually, translation to a computer symbol. Additional actions are possible; for example, all assemblers permit the definition (and often the redefinition) of coding and data location symbols within source programs.

Most assemblers will: evaluate certain source expressions to determine a computer symbol corresponding to the expression; retrieve, from a computer store or library, subroutines identified by unique name symbols; accept macro definitions (specifying special-purpose subroutines), and expand macro (multiple-step) instructions (by first inserting the source symbols associated with the macro name into specified positions within the corresponding macro definition and then replacing the macro instruction in the source program). In addition to translating source symbols into computer symbols according to rules built-in to the assembler or defined in the source program, some assemblers will translate according to rules stored in a separately prepared and easily changed symbol dictionary.

Assembly languages -- that is, those programming languages translated by assemblers -- usually have a tabular or columnar format. Because of the symbol-by-symbol translation performed by assemblers, there is ordinarily a very close, often one-to-one correspondence between sentences (instructions) in the assembly language and the resulting computer instructions, as shown in the following example:

ASSEMBLY LANGUAGE:	GET	ALPHA	4
COMPUTER LANGUAGE:	073	4	16372

Assembly languages are designed for the convenient expression of the machine instructions for a particular computer and are therefore almost as computer-oriented and procedure-oriented, and just as universal, as the computer languages to which they are translated.

In contrast to an assembler, a compiler operates on an expression-by-expression, sentence-by-sentence, or even message-by-message basis. Instead of simply reacting to each source symbol, a compiler must wait to recognize an expression, a sentence, or even a message type before it can make a translation into the target language; which may be a computer language or an assembly language. In the simplest possible terms, the main difference between an assembler and a compiler is that an assembler is primarily concerned with translating from the vocabulary of the assembly language to the vocabulary of the computer, while a compiler is concerned with translating both vocabulary and grammar. An example of such a translation is shown below:

Compiler (Source) Language (Cobol)

```
IF HOURS-WORKED (I) GREATER THAN
40, GO TO COMPUTE-OVERTIME.
```

Assembly (Target) Language

```
LIX   I,2    (Load i into IndeX 2)
GET   HRWK,2 (GET ith HouRs-WorKed)
UPK   HRWK   (UnPacK HouRs-WorKed)
SUB   =40    (SUBtract 40)
BRP   COMOT  (Branch or Result Positive
              to COmpute-OverTime)
```

Because of the more complex translation performed by a compiler, compiler languages are not confined to expressing programs in terms of computer operations and data structures. Instead, a compiler language can be designed for the convenient expression of broader operations upon more elaborate data structures, useful in a variety of applications.

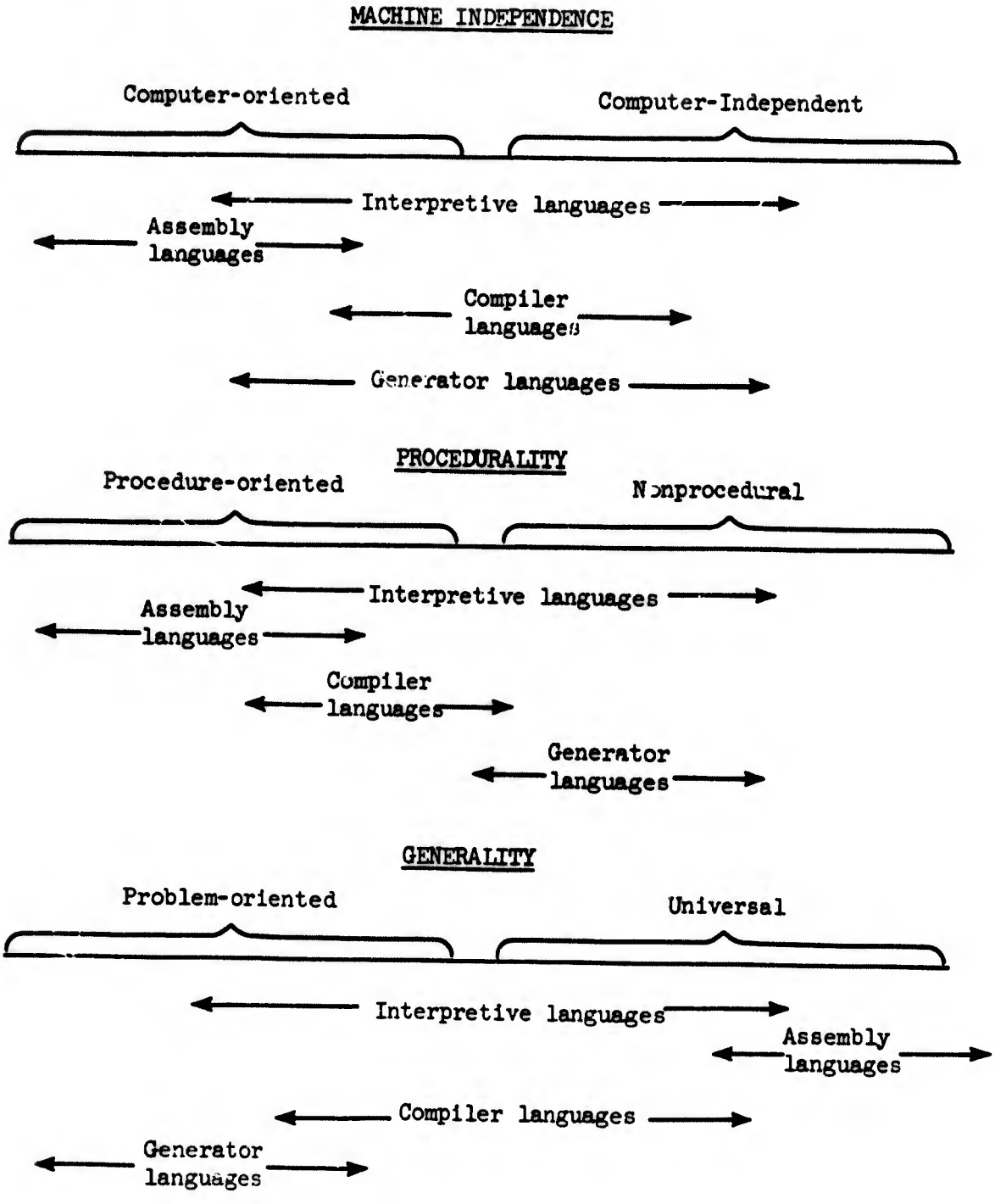
There are syntax-oriented compilers, for example, that exhibit the syntax or grammar of the source language, and perhaps of the target language as well. For each term used in describing the syntax of the source (and/or target language), there is a corresponding portion of such a compiler that processes (or a table-entry that directs the processing of) the associated class of symbols, expressions, and sentences. There also are table-driven compilers, whose processing is largely controlled by tables of parameters. Finally, there are compilers that are both syntax-oriented and table-driven (often called syntax-directed compilers), where compilation is directed by a

syntactic description of the source language in terms of the target language, or of both, in terms of some intermediary language.

In a one-pass compiler, the program being processed is scanned only once, with target language output being produced directly from this scan. In a multipass compiler, the source program being processed is scanned several times, each time undergoing some further transformation toward the target language.

In contrast to both assemblers and compilers, which, in a manner of speaking, compute target language programs from source language inputs, generators operate by a process akin to editing. A generator constructs a target language program from a generic or prototypical master program by retrieving (or deleting) selected portions, by rearranging these portions, and by systematic replacement of parameters. This is done in accordance with, or under the guidance of, source language inputs.

Generator languages are, typically, syntactically not very complicated. Programs in a generator language often consist merely of lists of parameters, which the programmer prepares simply by filling out an explicit, English language questionnaire. Generator languages tend to be less computer-oriented than assembly languages, less procedure-oriented than compiler languages and more problem-oriented than either.


MACHINE INDEPENDENCE
Computer-oriented
Computer-Independent
Interpretive languages
Assembly languages
Compiler languages
Generator languages
PROCEDURALITY
Procedure-oriented
Nonprocedural
Interpretive languages
Assembly languages
Compiler languages
Generator languages
GENERALITY
Problem-oriented
Universal
Interpretive languages
Assembly languages
Compiler languages
Generator languages

Notice that the interpretive languages span the variance of the other three language types. This is because any programming language can be either interpreted, or translated. We have already discussed briefly the relative merits of interpreters and translators. But, what are the advantages and disadvantages of the other three programming-language types? In general, assembly languages are more computer-oriented, more procedural, and more universal than either compiler or generator languages; and generator languages are less procedure-oriented and more problem-oriented than either assembly or compiler languages. The big advantage, as we go from assemblers to compilers to generators is increased ease of programming, due primarily to the decrease in procedurality. Increased machine independence is often advantageous also. But there are also disadvantages. Increased machine independence brings along with it, decreased efficiency of machine use. Another disadvantage is a narrowed scope of application. With an assembly language, you can write any conceivable program; with a compiler language, you are practically limited to writing programs that use just the operations and data structures that can be expressed in the language; and with a generator language, you can only write programs for specific kinds of problems.

2. SOME CURRENTLY AVAILABLE PROGRAMMING LANGUAGES

The last census of programming languages was published in 1963. Even though only a partial census, it listed about 75 different computer-oriented programming languages, and more than twice as many computer-independent programming languages, not counting the various dialects of such widespread languages as Cobol or Algol. Since then, many of these languages have undoubtably fallen into disuse. And even more surely, many new languages have been developed. In this section, consequently, only a sampling of widely used, highly compatible, and currently available major programming languages are described. However, an attempt has been made to cover a variety of programming language types.

a. Fortran IV

Fortran IV is a procedure-oriented, largely computer-independent, compiler language. Used by programmers at all levels of competence, it is primarily intended for the numerical calculations of science and engineering.

Fortran I, introduced by IBM in 1957 for their 704 computer, was the first compiler to make any pretense of producing efficient machine code.

Fortran II, introduced along with the 709, became the first procedure-oriented programming language to achieve wide user acceptance and has been implemented for almost all scientific computers.

Fortran IV, introduced by IBM in 1962 for their 7090/7094 computers, is a considerable improvement over Fortran II. It has been very widely implemented, for computers made by almost all U. S. manufacturers, and has replaced Fortran II in the majority of scientific computing installations in the United States.

The American Standards Association has recently published the specifications of American Standard Fortran, which corresponds to Fortran IV, and it can be expected that this will slowly replace the de facto industry standard (Fortran IV for the IBM 7094), which is treated here.

Fortran IV can describe and express the processing of several types of numeric data, including: single-precision, complex, and double-precision floating-point values; fixed-point integer values, and logical (Boolean) values. Symbolic values beyond the 48-character Fortran alphabet may be described, but their internal processing must be expressed in numeric terms. Qualitative (status) values can also be handled in numeric terms. Fixed-point scaling can be done by explicit multiplication with a power of two.

Fortran IV can name and describe single values in terms of the data types mentioned above. Machine-language symbols representing single values may be contained in either one or two computer

words -- the latter size being for complex and double-precision floating-point values. The manipulation of partial-word data values can be done by programmer-supplied machine-language subroutines, which also are the means by which integral constants and values can be treated as being composed of multiple single values.

Fortran IV can name and describe rectangular arrays of data values with one, two, or three dimensions. It can also name and describe groups (called blocks) of data values and/or arrays. Fortran IV has no special facilities for variable-length arrays, but it provides the following controls for optimizing data storage allocation: overlaying of data values, arrays, and blocks; relative origins for data values, arrays, and blocks.

Fortran IV's operands are single data values. In certain cases, values that are replaced, compared, or combined need not be similar in type -- except that logical (Boolean) values may not be mixed with numerical values. The manipulation of dynamically specified, part-word segments can be done by programmer-supplied, machine-language subroutines.

Fortran IV includes a complete set of arithmetic and relational operators, and while there are no general restrictions on the complexity of arithmetic formulas, those used for indexing arrays

can be of only limited complexity. Comparisons (relational formulas) may involve only two operands. Fortran IV includes the logical operators AND, OR, and NOT; there are no further restrictions on the complexity of logical formulas. It also includes the data-selection operator index, and the data-movement operator replace, which enables automatic conversion among integer, single-precision floating-point, and double-precision floating-point values. Fortran IV also contains a built-in collection of over 50 subroutines for the most commonly used mathematical functions.

Fortran IV can express both fixed and dynamically variable transfers of (program execution) control to numbered, imperative sentences. It has two kinds of dynamically variable transfer. In one, a transfer is selected by index from a list of transfers. In the other, one transfer is pre-assigned from a list of transfers to be the effective one.

Fortran IV can express conditional imperative sentences which are conditional on the truth value of a logical formula. It can also express selection of one of three alternative fixed transfers of control, according to whether the value of a specified arithmetic formula is negative, zero, or positive.

The loop generation mechanism in Fortran IV provides for one implicitly varied and tested loop-controlling parameter per loop.

Loop parameters can consist of integer-valued variables which are not part of an array. They are not defined after the completion of the loop with which they are associated, but they are defined if that loop is terminated prior to completion. No limit is placed on the construction of loops within loops.

Fortran IV specifies sequences of one or more loop parameter values and (thus loop iterations) in terms of an initial value, a limit value, and an increment value. Each of these values must be specified either by an integer constant or by an integer-valued variable that is not part of an array and whose value must not be varied by the execution of the loop. Completion of the loop occurs if the value of the loop-controlling parameter exceeds the limit value.

In Fortran IV, the body of a loop follows the loop-generating expression and its final imperative sentence is specified therein by its identifying number. After completion of a loop, program execution control goes to the first imperative sentence after the loop's body. A loop may be terminated by a transfer of control to an imperative sentence outside its body, but control may not be transferred into the body of a loop from outside. Fortran IV provides for transferring control from within the body of a loop to the implicit loop-control routine.

Fortran IV provides for closed subroutines, with or without parameters. It also provides for function-type, closed subroutines, as well as a fixed set of over 30 built-in, function-type, open subroutines, both types having parameters. In general, each subroutine has a defined number of parameters, which must all be listed in the subroutine call. (However, a few of the built-in subroutines have a variable number of parameters.) Fortran IV subroutines may call other subroutines but they may not, either directly or indirectly, call themselves, nor may they contain other subroutines.

Fortran IV allows the following as subroutine parameters: single data values, as denoted by constants, variables, and formulas; constant, symbolic values,* the names of other subroutines; the names of arrays. Dimensions of arrays that are subroutine parameters may be adjusted by other parameters in the subroutine call, so as to correspond to the actual dimensions of the arrays named in that call. Names defined inside a subroutine do not conflict with names defined outside it, for program elements defined outside a subroutine may not, in general, be referenced within it (except, of course, as subroutine parameters and also for simple, function-type subroutines). However, Fortran IV allows data

---

* Used only as input parameters for machine-language subroutines.

elements defined inside a subroutine to share the same storage block with data elements defined outside it, so indirect references are possible.

A Fortran IV subroutine may have only one entrance and may transfer control to only one (implicit) return point outside the subroutine, but it may have any number of effective exit points.

Fortran IV provides relatively machine-independent language features for input and output, for handling input and output devices, and for describing external data formats.

In Fortran IV, input and output are done with imperative sentences that may contain the following information: an indication of whether to read or punch cards or print on-line, or to read or write binary or symbolic data from magnetic tape or perhaps other devices; a reference, which may be dynamically varied during the program execution, to an input-output device; a reference to an external data format, according to which conversion to or from machine-language representation is done; a list of values to serve either as the source of output data for conversion or as the destination of input data for conversion or as the destination of input data after conversion. This list contains references either to single values* or to entire arrays of values. Arrays

* Indexes that select single values from arrays, may, of course, vary during program execution.

may, of course, vary during the program execution. Groups of both kinds of references (and, indeed, of other implicitly repeated groups) may be implicitly repeated -- with specified index variation of the kind allowed in loops.

Other Fortran IV imperative sentences allow magnetic tapes to be rewound or backspaced, or to have an end-of-file mark written on them.

The description of external data formats in Fortran IV is done with a declarative sentence that may either be part of the source program, or part of the data, input during program execution. A format description consists of a list of field specifications, each indicating conversion type and field size for a single, internal data value. The conversions that may be specified are between the following: single-precision, double-precision, or complex floating-point values and rational, decimal numbers, with or without a decimal exponent scaling factor; integer values and decimal or octal numbers; symbolic values and strings of characters. A list of field specifications may be partitioned into separate "line" specifications, and within the list, groups of one or more field or line specifications may be implicitly repeated a specified number of times.

Fortran IV includes a number of miscellaneous features that provide for: stopping program execution, with a restart provision;

returning control to the system monitor; appending to the Fortran IV program assembly or machine-language subroutines; presetting initial values for any or all data elements. In addition, Fortran IV has over 60 built-in subroutines. Besides the more common mathematical functions, these provide for: setting and testing the computer's sense lights; testing the divide-check indicator (for division by zero); testing for floating-point overflow, and dumping internal storage.

Fortran IV is an algebra-like programming language -- partly free-format; partly fixed-format. Almost all of Fortran IV's built-in vocabulary of symbols is familiar, being derived from algebra or from English. Programmer-supplied identifiers are either numbers (for sentences) or alphanumeric names of six characters or less (for data). The latter are thus occasionally somewhat limited in scope. Constants are either familiar or quite suggestive of the values they denote, once the few abbreviations employed in their identification are understood.

The following program elements are defined by name in the system: programs; subroutines; single values and arrays of values, and blocks of storage space. Arbitrary numbers are used to identify imperative sentences (including dynamically variable transfers of control), and external data formats. Fortran IV does not include any words of explanatory text within operating statements, comments being inserted on separate lines.

Fortran IV allows the omission of declarative sentences describing data-value type, since this information may be implied by the initial letter of the value's name.

As far as intrinsic complexity is concerned, Fortran IV has: almost 60 build-in symbols in its vocabulary (not counting the names of build-in subroutines); 6 different kinds of constants; identifiers for about 6 different kinds of program elements, and approximately 25 different basic sentence forms. It is, therefore, a programming language of moderate complexity.

b. Cobol

Cobol, which stands for Common Business Oriented Language, is a procedure-oriented and relatively computer-independent compiler language. It is by far the most widely used compiler language for business data processing, having been implemented (as a result of government requirements) for just about every domestic computer being marketed for business applications involving large magnetic tape files of alphanumeric data. It is intended for use primarily by professional programmers, but it employs English language symbology to facilitate understanding by novice and non-programmers.

Cobol was developed and is now maintained by CODASYL (Conference on Data Systems and Languages), a joint committee of representatives from industry and government.

The initial version of Cobol was developed in 1959 and published the next year as Cobol-60. The second version, Cobol-61, was quickly followed, in 1962, by Cobol-61 Extended,* and between them, these two versions form the basis for most Cobol compilers in use today. The current version, Cobol-64, was published earlier this year and is basically a clarification of Cobol-61 Extended with the addition of Mass Storage options for handling Drum and Disk Files, and a Table Handling feature that provides indexing and searching options.

The standardization of Cobol began in 1963 and is currently being continued nationally, by Working Group X3.4.4 of the American Standards Association (ASA), and internationally, by Sub-Committee TC-97/SC-5 of the International Organization for Standards (ISO). Working Group X3.4.4 hopes to publish a proposed American Standard Cobol, based on Cobol-65, sometime this Fall.**

---

* Cobol-61 Extended is basically Cobol-61 with various clarifications and the addition of the Sort feature, the Report Writer option, multiple receiving fields, and the Corresponding Option.

** The Working Group has divided the language into a nucleus and eight other functional modules: Sequential Access; Random Access; Random Processing; Sort; Report Writer; Segmentation; Library; and Table Handling. Within each module, they have specified two or more levels, and in some modules, the minimum level is the empty set. Any Cobol compiler implementing a specified level of each of the nine modules would be considered an ASA Standard Cobol compiler. For example, minimum ASA Standard Cobol would consist of the minimum level for each module, while full ASA Standard Cobol would consist of the maximum level for each module. With two or more levels for each of nine modules, over 500 and perhaps as many as 1,000 different ASA Standard versions of Cobol are possible, including minimum and full. In addition, the Working Group recommends that the implementation of additional language elements, not included in the ASA Cobol specifications, should not render an ASA Standard Cobol compiler non-standard. This proposal promises to do little to enhance the machine-independence of Cobol (surely one of the major objectives of standardization) and its acceptance, therefore, seems questionable.

A Cobol program is composed of four divisions. The Identification Division, which has no operational effect, merely serves to tag the source program and the various compiler produced outputs with the names of the program and the programmer, and the writing and compiling dates. The Environment Division serves to describe the equipment configurations of both the compiling computer and the executing computer. The Data Division, naturally, describes the data -- the files, records, groups, and fields, the working storage area, and the constants. Finally, the Procedure Division contains the imperative sentences that specify the data processing.

Cobol can describe, and express the processing of fixed-point numeric values, with automatic scaling; character-string values; status values; groups of named data elements, fixed- or variable-length tables (one-dimensional arrays) of homogeneous, indexed data elements, and files, which are tables of records, themselves data elements, accessible to the program one-at-a-time. By "data element" is meant a single value, a group, or a table. Groups may be nested up to 49 levels deep in Cobol, while tables may be nested up to three levels deep (permitting arrays of three dimensions -- or four, if the non-indexed file dimension is considered).

Cobol also permits specification of the packing and overlaying of data elements. Cobol includes the usual arithmetic operations, a complete set of relational operations, and an adequate set of

logical operations, and an adequate set of logical operations for combining relational and conditional expressions. In Cobol, arithmetic and relational operands are single, usually designated, values.

Cobol also has several powerful data handling operators -- one to scan the characters composing any designated data element, tallying and/or replacing occurrences of a designated character; another to replace the value(s) of one designated data element with the value(s) of another, either en bloc or sub-element by sub-element, with appropriate conversion and editing where the sub-elements are single values; still another to perform a serial or binary search of the sub-elements of a table until a specified condition is satisfied; and finally, one to sort the records in a file. Cobol also provides imperatives for the input and output of records from both sequential and random-access storage devices, for the entry and display of small amounts of text, and for the generation of printed reports according to formats defined in the program's data division.

Cobol has the usual operations for specifying the program execution control, including conditional execution of imperative statements, dynamically variable transfers of control and single or iterative execution of (parameterless) procedures.

In addition, Cobol permits the specification of the asychronous or parallel processing of several procedures, along with a processing delay contingent on the completion of all parallel procedures, or on the completion of the parallel processing of designated data elements.

Cobol also permits: the segmentation of large programs to be specified, the copying of Cobol text and the inclusion of subroutines from a library, and the insertion of routines written in other (usually more computer-oriented) programming languages.

Cobol is an English-like programming language. Every effort has been made to ensure that individual Cobol sentences are also grammatical English sentences. While there are staunch defenders of this policy, opinion is also widespread among professional programmers that this effort would have been better spent in designing a less complex, less English-like, but no less expressive programming language.

c. PL/I

PL/I, standing for "Programming Language/One," is a general-purpose, procedure-oreinted, and, (potentially), a highly computer-independent compiler language. PL/I is a broad language, intended for commercial, scientific, and systems programming. It is suitable for use in its entirety only by highly skilled, professional system programmers. It has been designed, however, so

useful subsets can be selected from the full language, and these subsets should be no more difficult for the non-programmer to learn than, say, Fortran or Cobol.

PL/I was designed jointly by IBM and SHARE (the users' group for IBM large-scale scientific installations). It is currently being implemented by IBM for their 360 series of computers,* and by several other computer manufacturers, notably General Electric and Scientific Data Systems. Industry interest in PL/I is high and there are expectations that it will become widely available, eventually exceeding the currently established compiler languages, such as Fortran, Cobol and Algol in extent of use.**

PL/I can describe, and express the processing of, an unusually wide variety of data types: binary or decimal, real or complex, fixed- or floating-point numeric values that are precise to a specified number of digits (including fractional digits for fixed-point values); symbolic values consisting of strings of characters or bits; address-type values for referencing imperative sentences, data elements, and storage areas; and logical values

---

* A 360 compiler for a subset of PL/I is being field tested, and recently revised plans are to augment this compiler so it will process a larger subset of the language.

** Without, however, necessarily diminishing the usage of these languages.

(either character or bit strings, or event indicators). Qualitative (status) values are handled, but somewhat indifferently.*

PL/I can name and describe: single numeric or symbolic values, as mentioned above; multidimensional, rectangular arrays of homogeneous, indexed data elements (with dynamically variable dimensions); and groups (called structures in PL/I) of named data elements -- where an element can be a single value, an array, or a group. PL/I provides for the manipulation of data elements that are dynamically associated in storage by explicit, address-type links. In addition, the language provides facilities for packing and overlaying data elements, for dynamic storage allocation, and for transferring data between primary and auxiliary storage. In describing data, implicit "default" specifications prevail for omitted portions of the description.

PL/I provides for an impressive repertoire of arithmetic, relational, logical, and string manipulations, and data movement operations on operands that are single values, arrays, groups, subarrays designated either by index or name, or lists of these elements. Operations may involve both single- and multiple-valued operands and are performed on a value-by-value basis. The widest possible tolerance is extended toward mixed expressions, so that logical values may be used arithmetically, for example.

---

* They must be uniquely defined for the entire source program, and are not associated with individual status-type variables.

Imperative sentences in PL/I may be grouped into procedures (i.e., subroutines), into blocks (to delimit the scope of names), and into groups (for execution control purposes). Data elements may be internal to a block or a procedure (and thus undefined outside it), or they may be allocated statically, at program load time, or dynamically, upon entry to a block or procedure or upon execution of an ALLOCATE statement. Dynamic storage allocation pushes down any previously allocated storage, which is popped up when storage is freed, upon exit from the block or procedure or upon execution of a FREE statement. PL/I procedures may have multiple entry points, they may be recursive and re-entrant, and they may be compiled separately for later combination.

PL/I provides for the input and output of data considered either as a stream of character-coded single values, or as a sequence or set of individual records of internally coded data elements. Stream-oriented transmission may be: data-directed, where the input or output character stream contains both the designation of the data elements and their values; list-directed, where the data values are designated in a list in the input or output statement and transmitted in a standard format; or edit-directed, according to non-standard formats specified (or referenced) in the input or output statement. Record-oriented transmission, on the other hand, deals with the buffered or unbuffered input and output to and from either linear or random access files or direct access files.

In PL/I, a procedure may initiate another procedure as a separate task, to be executed concurrently with the first procedure, which may later either wait until, or determine whether, the task has been completed. Alternatively, a task may specify the conditions under which it may be interrupted and, for each condition, a group of statements to be executed whenever the condition occurs. Interrupt conditions may be those generated by the hardware, or they may be generated by concurrently operating tasks.

The PL/I programmer may specify algorithms to be executed during the compilation of his program, as well as during its execution. In these meta- or compile-time algorithms (the PL/I report uses the word "macro"), meta-variables and meta-procedures may be declared, meta-variables may be assigned new values, meta-control may be transferred, conditionally or unconditionally, to other meta-statements, thus either skipping or repeating the compilation of intervening groups of PL/I statements. Wherever a meta-variable is designated (or a meta-procedure is invoked) in a PL/I statement, its current (or returned) character-string value is substituted.

d. Decision Tables

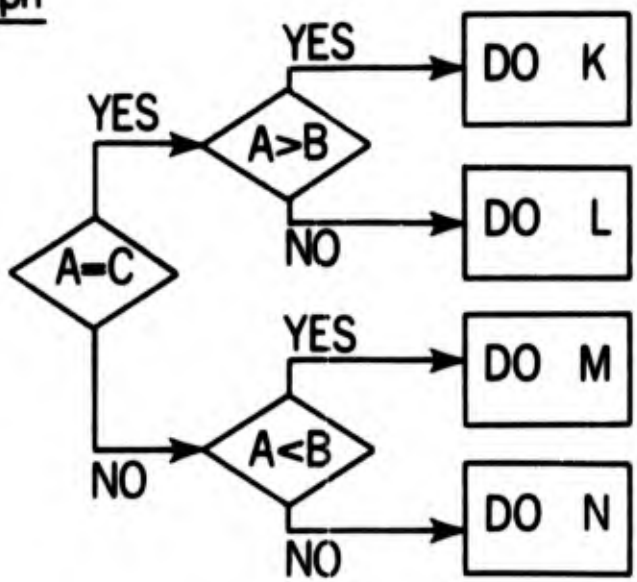
A decision table is a way of representing complex decision rules in an easily grasped, tabular form -- a form that makes it easy to see what actions are to be taken for each possible combination of conditions. The following examples show the same decision

rule represented in three ways: as a string, as a directed graph or flow chart, and as a decision table.

string

IF A=C THEN (IF A>B THEN DO K ELSE DO L)
ELSE (IF A<B THEN DO M ELSE DO N)

graph



table

	1	2	3	4
A=C	Y	Y	N	N
A>B	Y	N	–	–
A<B	–	–	Y	N
DO K	X			
DO L		X		
DO M			X	
DO N				X

In the table, the top rows are conditions and the bottom ones are actions, and the numbered columns on the right are the decision rules.

The Y means yes, this condition must hold for the rule to apply. The N means no, this condition must not hold for the rule to apply. The dash means: ignore this condition. The X means: do this action if the rule applies. And the blank means: do not do this action if the rule applies.

By convention, the rules are examined one at a time from left to right. Only the first rule found is applied, and the actions are taken one at a time in the order they are listed. Other conventions are feasible though.

Decision tables where the entries are limited to the symbols just described are called limited entry tables. Extended entry tables are possible, though, where part of the condition or action is

number of legs	=4	=4	=4	>4	-
length of nose	long	short	long	-	-
length of neck	short	long	long	-	-
then animal is	elephant	giraffe	hallucination	centipede	unknown
call	zookeeper	zookeeper	psychiatrist	exterminator	biologist

entered in the rule columns. The next example shows an extended entry decision table.

Decision tables are well known. Yet they are not as widely used as they should be in computer programming. Their chief advantage, it seems, is that they make explicit all of the possible paths through a decision routine, since each path corresponds to a column in the table. This makes it much harder for the programmer or analyst to inadvertantly overloop a path.

Several decision table processors have been written in the last few years to translate decision tables into computer programs. General Electric's Tabsol is the most prominent example. Recently, a group in Los Angeles completed a processor that translates from a decision table language called Detab, to Cobol. The processor itself is written in Cobol, and the intent is to make decision tables available to all Cobol users. Similar processors could be (and are being) written for other languages.

3. THE STATE OF THE ART IN EXECUTIVE SYSTEMS

Most current medium and large scale data processing activities utilize a system which controls the operations of the computer. An essential part of this operating system is an executive program. It must be noted that an executive program is first of all a program, and thus can be written in a number of existing programming languages. Furthermore, the characteristics of programming languages described in the previous section apply to those programs considered to be executive programs. In essence, an executive program controls the running of one or more operating programs. In addition, every program has as part of it an executive function. In the simpler cases, this executive process may be nothing more than control of the relationships between peripheral equipment demands on the computer's central processing unit and other functions of the operating program. In these simple cases, the controlling portion of the program is not separately identified as an executive, but it serves that function.

The tendency in recent years to develop data processing systems which fulfill highly sophisticated requirements has led to considerations of executive programs as a separate class. Today we normally consider that there is a need for a separately defined executive program only when the system is operating simultaneously on a number of different programs, (as in a multiprogramming or time-sharing system), or when a number of interrelated computers are operating cooperatively (as in

a multiprocessing system). Examples of advanced executive systems designed for new, powerful computer systems are described in the following sections, in order to portray today's state-of-the-art in advanced computer system control software.

An executive program must be capable of investigation and analysis of worker programs during their operation on the computer. To some extent, this limits the choice of languages available for the preparation of executive programs. Since executive programs represent a fair degree of sophistication of software, they are usually expensive to generate. Happily, however, there has been a tendency in recent years on the part of hardware manufacturers to improve the quality and variety of the software library available for use with the purchased computer. The application of such standard executives reduces the effort required by the user to generate his own probelem-oriented system. The manufacturer of the hardware can generally be counted upon to supply such executive components as peripheral equipment sub-programs, compilers, translators, program dumps, and other basic elements from which an executive program can in large part be assembled.

An executive system is more than an executive program. It consists of the program for ongoing operating control, debugging programs, other support programs of assistance to the computer operator and programmer, any bookkeeping functions imposed for management purposes,

and last but not least, the computer operator. It is essential that an executive program have certain logical capabilities, such as the decision to interrupt an ongoing process to respond to the needs of a peripheral, but not all logical conditions can be foreseen or pre-programmed. The computer operator or attendant is thus a necessary part of the decision-making aspects required of an executive system.

It is, of course, an over-simplification to say that the executive system, and particularly that portion of it which is the executive program, is charged with the responsibility of making routine decisions of computer operation during the run. Nevertheless, this over-simplification serves to emphasize the fact that the executive system operates with the elements of the running programs just as these elements operate with the data. One of the great advantages in the development of a total operating system which is provided by the use of a separate executive is the ability to prepare the executive program before the details of the operating program are fully known. Many executive systems have been designed and a full description of each would be far too lengthy for this document. The examples of a few executive systems in the following sections are chosen to illustrate the various types of applications of such systems which have been, or could be, used in the fields of interest to DDC.

a. Sipros

Sipros (Simultaneous Processing and Operating System) is a programming and operating system for the Control Data Corporation's

6400, 6600, and 6800 computers. These very large and rapid computers are multiprocessors; they consist of ten peripheral (input/output and control) processors and a central processing unit. Sipros is designed to provide for efficient use of the central processor over a variety of hardware configurations and job mixes.

The Sipros programming subsystem includes a Fortran compiler, and assemblers for both the central and peripheral processors. The Fortran compiler has a fast compilation mode that omits (the normal) code optimization; Fortran statements and assembly-language instructions can be freely intermixed; programs can incorporate peripheral processor routines; and program segments can be indicated for purposes of overlaying.

The highly parameterized Sipros operating system provides for both batch processing and multiprogramming. In Sipros, input/output areas, units and numbers of peripheral devices, and time and utilization limits are designated by easily changed parameters. Sipros loads programs into the central processor according to job priorities and equipment and memory availability, and it runs them to completion (or until they request nonbuffered input or output). Sipros tries to load as many programs as possible into the central processor, to minimize central processor idle time due to input/output delays.

Sipros' executive and monitoring functions are performed by a peripheral processor that exercises overall system control. It evaluates priorities; allocates hardware; schedules and loads jobs; handles input/output requests; and monitors job and input/output status. However, the central component of the system is the job stack which is stored on magnetic disk. While the system is operating, programs are placed in the stack, in the form of card images. This is done autonomously, by other peripheral processors, from punched cards, magnetic tape, or from on-line terminals.

Sipros recognizes 2048 program priority levels. The priority of a job may be either fixed, or periodically incremented -- up to predetermined limits, which prevent such jobs from delaying higher priority programs under heavy load conditions. The period and the limits of priority incrementation are parameters which the installation may define.

Based on control card information, the Sipros executive picks programs with the highest priorities that best fit the available peripheral equipment and memory space, and loads them from the job stack into central memory for execution. For jobs above a certain priority, the system will reserve automatically, enough peripheral equipment and memory space. That is, as such resources become available, the Sipros executive will refrain from assigning

them to lower priority jobs until the high priority job is loaded. During execution, jobs can request (and relinquish) additional peripheral equipment and memory space, and Sipros resolves any allocation conflicts that might arise.

Once a job has been loaded into central memory for execution, it stays there until its operation is completed, since there is evidently no provision in Sipros for swapping programs back and forth between central memory and the disk. While in central memory, jobs are executed by the central processor according to priority. Every 200 microseconds, the Sipros executive monitors the status of jobs in central memory, interrupts the job currently being executed, and transfers central processor control to another job (if that job has built up higher priority of if the current job has requested nonbuffered input or output). The priority of jobs being executed is not incremented, so it is possible for the system to take turns executing several jobs with equal initial priority.

b. Mcp

Mcp (Master Control Program) is a programming and operating system developed by the Burroughs Corporation for its B-5000 and B-5500 computers. It can operate in a variety of configurations, with one or two central processors and various numbers of memory modules, input/output channels, and peripheral devices. The

system is designed to load and run programs on a priority basis, while at the same time minimizing central processor and input/output channel idle time.

Mcp is a batch processing, multiprogramming operating system with compilers for both Algol and Cobol. It is perhaps best briefly described in terms of its normal cycle of operations.

Mcp resides permanently on a magnetic drum, and the computer operator loads it from the drum into core memory. By means of the computer's interrupt mechanisms, Mcp first interrogates its environment to construct a list of the available hardware resources: processors, memory modules, input/output channels, and peripheral devices. Any later change in the configuration due to hardware failure will cause an interrupt, and Mcp will revise the list.

Once the system has been loaded, and at any later time, the operator may interrupt to enter a message requesting Mcp to add new jobs to its (initially empty) schedule or, perhaps, to revise the priority of jobs already scheduled. Reading these jobs from punched cards or magnetic tape, Mcp will merge them, by priority, into its schedule. This involves updating a schedule table in core memory and creating a new program tape, which contains the source decks, object decks, and program calls that specify the system's jobs.

Mcp will load a new job from the program tape onto the drum and into core memory, to be run whenever processor time is available and none of the currently loaded jobs are ready to run. A given job will only be loaded when there are no higher priority jobs in the schedule, when there is enough storage space on the drum, when the required peripheral devices are free, and when there is enough core memory space available to begin running the job.

Jobs consist of segments of arbitrary length, including program segments, data segments, and input/output buffer segments. At any one time, segments from several jobs may be residing in core memory. Words from these segments are accessed indirectly by the computer, by means of a reference table associated with each job. Thus, Mcp may relocate segments in memory merely by moving the segment and changing an entry in the job's reference table. When a job is loaded, its program and data segments are stored on the drum, and it is not necessary for all of them to reside in core memory while the job is being run. Space is allocated in core for the job's buffer segments, but only the initial program segment is brought into core when the job is loaded. Each segment of a job has an availability bit in the job's reference table. Program and data segments are unavailable when they are not residing in core memory, and a buffer segment is unavailable when it is involved in an input/output transfer. Whenever a job

references a word in an unavailable segment, an interrupt occurs and Mcp takes control.

Interrupts may occur for a variety of reasons, and, after processing each interrupt, Mcp will resume (or initiate) that job that has the highest priority and has been loaded and is ready to run. Mcp will try to load a new job, however, if none of the jobs already loaded are ready to run. This situation could occur if the interrupt were due to a reference to an unavailable buffer segment, because, when this happens, the interrupted job must wait for completion of the input or output transfer before proceeding. The situation could also occur if the interrupt signaled the completion of a job.

If the interrupt were due to a reference to an unavailable program or data segment, Mcp would bring the segment into core from the drum, if necessary, overlaying available program segments or swapping available data segments back out to the drum. The overlayable segments of Mcp would go first, program segments of the lowest priority job would go next, then data segments of the lowest priority job, and so on.

Interrupts also occur when a job asks the system to perform input or output, when an input or output transfer is completed, when the interval time overflows, and for a number of other reasons, including

such error conditions as when a job references a word outside the boundaries of one of its segments or exceeds its expected running time. Mcp processes these interrupts in a relatively straightforward manner.

c. Gecos

The programs comprising Gecos (GE Comprehensive Operating Supervisor) -- along with such associated programs as the Remote Terminal Supervisor (Gerts), the File and Record Controller (Gefrc), the Loader (Geload), the compilers for Fortran IV and Cobol 61 Extended, and the Macro Assembly Program -- constitute the programming and operating system for the General Electric 600 series of computers.

Jobs, each consisting of a sequence of one or more programs, are read from the card reader (by the input converter, Gein) or from remote terminals (by Gerts) and are stored on the drum (or disk). Each job has a priority (from 1 to 63), usually assigned by the installation's staff. The programs of the 18 most urgent jobs are then considered in turn (by the allocator, Gealoc) for allocation of peripheral devices and core memory space. Allocation is governed by the following principles: (1) only currently available resources are allocated; (2) a program is allocated all the resources it needs or it is allocated none; (3) when there are jobs above a certain threshold level of priority, only

programs in the job with the highest priority are allocated resources; (4) each time a job is unsuccessfully considered for allocation of resources (i.e., none of its programs are allocated resources), its priority is incremented; (5) if possible, several programs are allocated resources.

After a program has been allocated resources, it may be assembled or compiled, if it contains any source language subprograms, or it may be loaded (by Geload) and run if it is an object program and if all prior programs in the same job have been completed. Programs to be loaded consist of one or more subprograms in the form of absolute or relocatable binary decks. The programmer may prefix his program with debug requests, and he may segment it into overlays by inserting control cards between subprograms. The loader provides all the necessary linkages among subprograms, among overlays, and between the program and the debug routine.

Once a program has been loaded into core memory, it is ready to run. Gecos can handle up to eight user programs in core at once, and it is the task of the Dispatcher (Gedisp) to select programs to be run according to priority and to maximize the overlap of input/output and processing. Gedisp is entered whenever a system program completes its task, whenever a user program requests an input/output delay, or (as the result of an interval-timer interrupt) whenever a user program exceeds its allotted increment

(625 milliseconds) of processing time. In selecting the next program in core memory to be run, Gedisp will look, in order of priority, first at system programs, then at user programs, and finally at the program that has just been run, and it will select the first program that is ready to run.

Under Gecos, user programs are run in the so-called slave mode. This means they use relative addresses, which facilitate their relocation, and they are prevented from accessing words outside the consecutive set of 1024-word blocks allotted to them. User programs may call on the Input/Output Supervisor (Geios), usually through the File and Record Control (Gefrc) subroutine package, for input and output operations, and they may likewise call on the Dispatcher (Gedisp) for input/output delays. Programs are retained in core memory until their processing is completed or, if Gecos detects an error, until they are interrupted and terminated. The Termination routine (Geterm) removes the program from the system, thus freeing the memory space and the peripheral equipment allotted to it. If necessary, Geterm also relocates the remaining programs, to consolidate the free memory space into a compact block. After completing these and other tasks, Geterm calls the Allocator (Gealoc), which reallocates the released resources to other programs.

Periodically, Gecos calls the Output Program (Geout) to process the system output file (which contains job accounting information, debugging output, and other low-volume output not written onto separate output files) and produces the printed or punched results.

d. Univac 1108

The Univac 1108 Executive System is a general purpose operating system. This system has been designed by Univac and is currently being implemented. The designers claim a multiprogramming capability coupled with the ability to "demand process" (time-share), control real-time processes, and execute batch process runs. The design emphasizes the batch process method of running jobs.

Users communicate with the executive by way of an executive control language. The language is open-ended and, as such, could be turned into a very powerful tool. Currently, the user programs cannot activate the routines that interpret the user commands.

The executive will load and run any program the user wishes to submit. The only restrictions imposed are those of a given equipment configuration. The user's program is confined to its own part of core by a memory protection mechanism. In this way, programs are protected from each other when several reside in core. Programs or data that are concurrently being used by more than one user are replicated for each user. The method of protecting master files and interlocking programs that have conflicting needs for data must be supplied by the individual installations.

Special programs have been designed to tailor the executive to a particular hardware configuration and an installation's special requirements. The programs that do this operate in a non-Executive mode.

The initial system will have an assembler and compilers for Fortran IV, Cobol, and Algol. These processors will output object programs in a common form of relocatable binary coding. The allocator process will assemble relocatable subroutines from the compilers and the assembler into larger relocatable routines or into an "absolute" program.

The diagnostic system is essentially a memory dump routine with snapshot capabilities. No firm, published definition has yet been made of an interactive or conversational capability for time-sharing users.

The 1108 is compatible with the earlier Univac 1107 computer. As a result, all of the programs developed for the 1107 may be run on the 1108, if the machine configuration is compatible. The system is expected to be available by early 1967.

e. IBM Operating System/360

IBM Operating System/360 (OS/360) is a supervisory system featuring a comprehensive set of language translators, data management routines, and other service programs. It comes in several versions to operate in the various configurations of the IBM Computing System/360.

The executive portion of OS/360 is an extremely sophisticated batch processor with an optional multiprogramming capability. To meet the needs of many varied installations, OS/360 has been made as modular as possible. Thus, each installation will include, in its executive, only the modules needed. Interfaces between modules are such that new nonstandard modules may be substituted for the standard modules with a minimum of effort. The executive control functions are broken into modules as are the various programmer facilities.

Users' functions may be added to OS/360 at will, making such functions more accessible and easier to use.

In OS/360, a job is the basic independent unit of work to be performed. A job consists of one or more directly or indirectly related steps. Each job step results in the execution of a processing program, such as a language compiler, or a user's object program.

Each job step requests various units of work called tasks. Each task has associated with it a task control block (TCB). In scheduling work to be performed, the executive is choosing between tasks on the basis of information in the TCBs. Any task may create new tasks, providing considerable dynamic control by programs. The system was not basically designed for the initiation of new jobs by tasks, but such a feature could easily be added.

Options are available to allow dynamic control of priorities used in selecting jobs to be run. In the most advanced system, several jobs may be run concurrently (simultaneously), and each job may include several tasks in parallel operation.

The operation of a system with this degree of sophistication is feasible only if good data management is used. One of the strong points of OS/360 is the data management function. This consists of a method of naming, classifying, storing, and retrieving data. It is used by all the components of OS/360, and is available for use by object programs.

OS/360 allows for the sharing of programs and files to a considerable extent. The protection against disclosure is relatively poor, since it relies simply on a user program giving a correct keyword to gain access to any data. Protection against uncoordinated joint use is achieved by use of a lockout system that is valid in OS/360, but that prevents any programs that use shared data from running "simultaneously."

The executive system is so structured that it may not modify itself under its own control. This is not a deficiency in a batch processor. In a time-sharing system, stopping the system to modify it is generally intolerable.

OS/360 has provision for multiaccess use. It is primarily designed, however, to allow many operators to communicate with one program. Multiaccess to multiprograms is not feasible under the present design.

OS/360 contains the most sophisticated set of tools currently being made available to the professional programmer. It is particularly rich in language processors and other support programs whose utility is considerably enhanced by their common design features. For example, there is an assembler, and Fortran, PL/I, and Cobol language translators, all of which output programs in the same format. Because of this feature, programs can be assembled that are composed of parts written in different languages.

OS/360 is being produced in modules. A system with most of the specified capabilities has been announced for the Fall of 1966.

f. RCA Spectra 70 Series Operating Systems

The RCA Spectra 70 Operating Systems (RCA OS) are systems for Models 45 and 55 of the RCA Spectra 70 computer family. The Spectra 70 System is notable because it was designed to operate programs using the same instruction codes as those used in the IBM System/360 machines.* This compatibility on the machine-language

* Certain "privileged" instructions are not "common." Thus, some of the more basic executive programs of the RCA OS are not interchangeable with their IBM OS/360 counterparts. The vast majority of programs incorporated into the operating systems will not use these instructions.

level has important implications in the software area. It means that components of IBM Operating System/360 can be incorporated with little technical effort into RCA OS. Since the RCA OS seems to be, in large measure, a subset of OS/360, this method of expanding RCA OS should be kept in mind. If such an expansion is not prevented by non-technical considerations, RCA OS can be considered to be the equivalent of OS/360.

The RCA OS is a family of batch processors of modern standard design. Certain options give limited multiprogramming (maximum of three programs) and multiaccess (to one control program) capability.

The facility for sharing programs and files exists in a restricted form. Programs are given little capability for controlling the system in a dynamic fashion.

The most sophisticated version of the RCA OS seems to be RCA 70/45-55 Disk Operating System 262. It has the following components:

Executive

Monitor

Extended Assembly System

File Control Processor (FCP)

Report Program Generator

Cobol

Fortran

Sort/Merge

Peripheral Control System

Communication Control System (Multichannel)

System Library Maintenance Routines

Random-Access File Maintenance Routines

Such a system appears to have all the necessary tools for efficient batch processing.

The RCA Operating Systems for Spectra 70 Series Models 45 and 55 are not yet operational.

g. SDC Time-Sharing System

The SDC Time-Sharing System has been developed to provide the facilities of a large computer (IBM AN/FSQ-32) to many simultaneous users. An advanced version shortly will be operational on an IBM Series 360 Model 50 computer. The system automatically distributes increments of computation so that each user seems to be controlling the central processor, with access to most of the high-speed memory and to a share of the available auxiliary storage.

The system can serve as many as 50 simultaneous users; each can access up to 46,000 48-bit words (368,000 characters) of core memory, and share four million words of disk storage and 12 tape drives. Users' terminals are typewriters or teletypewriters -- both locally and remotely located. About 35 terminals are normally

connected to the system, and six of these have cathode-ray-tube/light-pen displays. To handle direct communications with the terminals, the system incorporates a small, DEC PDP-1 computer as an on-line satellite to the AN/FSQ-32.

The system also incorporates 400,000 words of high-speed drum storage, which is not directly available to users, since it contains the system's working copies of all current users' programs (with their in-core data). The system maintains these programs in two queues: a queue of "conversational" (time-shared) programs, and a queue of "production" programs. In operation, as long as the conversational queue contains active programs (those not stopped or awaiting input or output), the system is continually swapping active conversational programs between core memory and drum storage. While in core, a conversational program is given a quantum (400 milliseconds) of computing time, and is allowed to run until it stops or is stopped, until it requests input or output, or until its quantum runs out. Then it is swapped with the next active conversational program on the drum. If no conversational programs are active (as often happens), an active production program is swapped into core and run until it stops or is stopped, until it requests input or output, or until a conversational program becomes active. The production queue is rearranged regularly to give other active production programs a chance to operate.

At the user's command, the system loads a program from tape or disk onto the drums, and, initially, into the conversational queue. Any conversational program that computes for ten successive quanta without requesting teletype input is placed at the bottom of the production queue; and a production program that requests a teletype input or that completes its operation is placed back into the conversational queue.

The system provides commands that allow the on-line user to: communicate either with the system or with his own program; announce himself to the system; bring in a program from tape or disk; interrupt the operation of his program; unload his program; query the system as to his time usage, the number of current users, or the status of his program; find out how many tape drives, disk tracks, or drum words are available for his program; send a message to other terminals; link his terminal to other terminals, or prevent it from being linked to another terminal, and have the system repeat his input messages so he can verify their correct transmission. Although the system is designed to respond to a user's command within two seconds, users' programs may or may not equal this response time.

As the foregoing list of user commands shows, the main function of this time-sharing system is running users' programs. No restrictions are placed, though, on the functions these programs may

perform, and there are relatively few restrictions placed on their use of the available resources. However, programs must communicate with the system to request input and output, which may involve tapes, the disk or the user's teletype terminal.

h. The SDS Sigma 7 Operating Systems

The SDS* Sigma 7 Operating Systems are being designed in a modular, compatible fashion to work together in an interrupting environment typical of time-sharing. The operating systems, because of their modular elements, provide four programming levels. The levels are: (1) Stand Alone, designed for minimum configurations or those used for a single function at a time in a serial fashion. Primary criteria include heavy emphasis on minimizing core space requirements; (2) Basic, handled by a Basic control monitor, designed for real-time systems with concurrent general-purpose processing either completely independent or related to the real-time system, such as data reduction. The general-purpose processing includes program creation, checkout, and program execution capabilities; (3) Batch, handled by a Batch Monitor, designed for a typical production environment in which many jobs are being processed from a job queue; (4) Universal, handled by a Universal Time-Sharing Monitor, which permits batch-type operations to proceed concurrently with time-sharing operations and includes input/output operations conversational interactive users.

---

* Scientific Data Systems of Santa Monica, California, produces the (new) Sigma 7 Computer for which these Operating Systems have been announced.

Some of the more important capabilities provided by the Sigma 7 Operating Systems are: executive request processing; primary, secondary, and tertiary storage allocation and management including swapping and overlay; job scheduling; loading of common subroutines written in a re-entrant manner so that a single copy need be kept in core to service many users; communications for conversational users; interrupt processing; job accounting; memory-write protection and control; access protection and control; file security; error processing and recovery; run-time trap processing; master/slave mode control and debug processing and control.

4. THE STATE OF THE ART IN FILE PROCESSING SYSTEMS

Over the past twenty years computers have been applied in many fields. The earliest, and still most common, applications are in connection with scientific computations and the handling of business data. As a result, it has become conventional to speak of scientific machines or systems and business machines or systems. The last few years have seen a great increase in the diversity of applications. Few of these applications have been graced by being considered separate fields. One, however, offers such great promises over non-computerized techniques that a whole skill area has grown around it. This is the handling of files of data including those which are unstructured in nature. It should be noted that this area of application of data processing systems is also of particular interest to DDC.

It may be gathered from the previous paragraph that the handling of information in file structures, or file information systems, is an extension of previously developed capabilities and techniques. To some extent this is true. Experience has shown, however, that there are two significant differences between scientific calculation and limited-scale business data processing, on the one hand, and file information systems on the other. The first of these differences is the need for a large data base for file information systems. The development of hardware with large memory capacities to efficiently handle file information problems is a comparatively recent development. Prior to the availability of such memory sizes, it was much more difficult to undertake the development of such a system.

A second significant difference is the fact that file information systems as presently conceived and operating do not serve the ultimate desired purpose of such systems. Retrieval from such systems is generally made satisfactorily today on the basis of comparison between limited, structured data elements as expressed in the query and as stored in the machine. The ultimate is to provide retrieval on the basis of more general and comprehensive combinations of data as expressed in queries and stored in the computer.

A file information system requires, as in any other case, computer programs for operation, and a computer programming language must be chosen for the preparation of the program. A file information system, however, consists of both the computer and related software. Just as hardware being developed today features larger memories, faster access, scratch pad memories, etc., so also there are developments in process in software aimed at more efficient operation of file information systems. In particular, a considerable amount of research is being conducted into file design, maintenance and retrieval techniques to approach the point where search by complex (structured or unstructured) data combinations is possible, in addition to operations on limited, structured files.

The following sections provide some sample descriptions of various file information systems now in use or in advanced stages of development.

a. Adam

The Adam (Advanced Data Management) system is essentially an interpretive, non real-time file processing system. Adam has been programmed at the Mitre Corporation for use on the IBM 7030 (Stretch) computer. Several Adam users can be utilizing the 7030 at once, the user's file processing requests being stacked and interpreted on a priority basis.

Adam is intended mainly as a tool for building experimental or prototype data management systems. The user can add his own, special purpose routines to the Adam routine file, and he can add a description of his own special purpose query language to the Adam language file. Routines and language descriptions are prepared outside the Adam context. For this, there is a compiler for a procedure-oriented language that can handle Adam data structures, there is a post-processor that converts Fortran programs to Adam format, and there is a language assembly program that converts syntax rules into an interpretive program that translates input messages into Adam pseudo-instructions. Once prepared, routines and language descriptions are added by file maintenance operations to the Adam (program) Routine File and the Adam Language Files. These two files, along with the user's data files and the "Roles" (which are files that describe all files) specify a particular data management system. All these files can be manipulated during operation of the Adam

System, so the user can modify and request reports and displays on his file processing system as well as on his data.

In Adam, a file is essentially a linear series of indexed entries, each containing all the information relevant to a single entity or object in the file. An entry is a Record, or collection, of named elements, and is actually a subfile with its own entries. Adam handles numerical values; logical status values; queries; query responses (computed by retrieving, translating, and processing a stored query), and textual material. The basic file processing operations provided by Adam allow the user to describe and create new files from card or tape inputs or from existing Adam files; to modify, purge, and search files; to do computations on file data, and to call for the execution of problem-specific routines.

The ordinary sequence of events in Adam, somewhat simplified, is as follows:

A message is entered by the user at a terminal. This causes a transfer to the Input Schedule Program which converts the message to a standard internal IBM 7030 code and passes it on to the Recognizer Program. The Recognizer scans the first few words of the message, assigns it a priority and a handling routine, places it as a task in the Task Queue, and calls the Task Control Program. If the new task has first priority, it is immediately performed. If it has second

priority, the intermediate results of the current task are saved for later completion, and the new task is initiated. If the new task has third or lower priority, the current task is continued. When a task is completed (or interrupted) and a new task is to be initiated, Task Control unstacks a message from the Task Queue according to priority, loads the appropriate Handling Routine (ordinarily "Thruput Control") from the Routine File, and branches to it. For ordinary messages, though, Thruput Control retrieves a language description from the Language Description File and passes it, along with the message, to the Translator Program. (Additional language descriptions may be added to the language description file.)

The Translator interprets the language description and, from the message, generates a list of Adam computer language pseudo-instructions to effect the file processing specified by the message. It then branches to the Processor Program. The Processor interprets the Adam pseudo-instructions and calls the appropriate file processing subroutines. It may also perform certain data conversions.

When output is requested, the Processor retrieves a format statement from the Format File and calls the Output Formatter (Program), which converts the data into standard internal format, and sends it to the Output Scheduler Program. The Output Scheduler converts this to an appropriate external character code, initiates the output operation if an appropriate output device is available, or stacks the output

if it is not. It then returns control to the program that called it (ordinarily the Output Formatter), which then ordinarily returns control to the Processor. When an output message has been transmitted, and the output device is ready for another, the current task is temporarily interrupted while the Output Scheduler initiates a new output operation. When the processor has interpreted the last Adam pseudo-instruction in the current program's operation, it returns to Thruput Control, which then returns to Task Control with a "task complete" signal.

Strictly speaking, Adam is not a user-oriented system; rather, it is a tool for building and experimenting with generalized, user-oriented, file processing systems. Adam is oriented to the batch processing philosophy of running jobs to completion, as opposed to the real-time or time-sharing philosophy of periodically interrupting the current job to work on another. The batch processing philosophy allows more efficient use to be made of the available computing resources, while the time-sharing philosophy provides faster and more reactive response for more users. The Adam Task Control Program could probably be modified to periodically rearrange task priorities, to operate in a time-sharing mode and provide additional services to users.

b. Lucid

Lucid (Language Used to Describe Information-System Design) is an SDC real-time data management system operating under a time-sharing Executive System. Lucid, coded in the Jovial language, operates on an IBM AN/FSQ 32 computer, and is accessible to users via remote teletype consoles through the SDC time-sharing system.

Lucid was designed to minimize the requirement for a user to be sophisticated in terms of computer use. All references to data are made in terms of data names provided by the user himself. Computer data location addresses are computed automatically by the program, and the user has no need to be aware of them. Lucid provides an exceedingly simple language for the description of data. The user has only to name his file and to name the data elements of which it is composed. He states the type of each element, which may be textual or numerical in nature. He then presents the system with the input data which is automatically loaded, organized and converted into computer-useable form.

Lucid is designed primarily for facilitating rapid fact retrieval. For this purpose each file processed by Lucid exists in two forms. It exists, as loaded, as a series of data records or entries stored sequentially on peripheral storage (either magnetic disk, magnetic tape or both.) It also exists in an "inverted" form, organized by data value within data field. This organization is called a

concordance. At the time of data retrieval, the data selection criteria entered in the query are processed through the concordance very rapidly. The entries containing the output values required are determined first, and only then is the sequential entry file accessed, and only those entries specified are retrieved. This scheme permits exceedingly fast retrieval of data regardless of what the retrieval keys may be.

Lucid has some capability for natural language test manipulation, which is currently being expanded.

Lucid has been coded in Jovial except for input/output routines specifically with the purpose of making it relatively easy to transfer to other equipment for which a Jovial compiler exists.

c. Colingo

Colingo (Compile On Line and Go) is an interpretive, file-processing system for the IBM 1401 computer. It was developed by the Mitre Corporation for military use. Although Colingo can only serve one user at a time, it is a user-oriented, generalized, open-ended, and (conceptually) machine-independent system. The Colingo concept would be quite powerful on a larger computer.

Colingo interprets user requests, which are stored in chains on a magnetic disk; entered one-by-one through the 1401 console typewriter, or input in batches through the card reader. The Colingo Executive

Program examines the query and retrieves the appropriate query-processing routine(s) from the disk. The routine(s) interpret the rest of the query, and ordinarily: reads an input (or intermediary) file; performs some processing on file records, and writes an intermediary or output file or report. Other possible actions include: storing on the disk of a new query-processing routine; displaying a comment or question to the user at the typewriter; pausing to allow the user to enter a parameter; copying (on magnetic tape) all system data and programs (or perhaps just data files and dictionaries) for external use; storing a file for subsequent processing against another file, or checking the validity of input data.

Files in Colingo are self-describing, that is, each file carries along its own dictionary as an initial (header) record on magnetic tape or disk storage. (Even card files, which must be converted to Colingo tape or disk format before processing, are preceded by format-conversion control cards.) The user describes his files in the style of the Cobol Data Division. Colingo allows use of both master and detail (update or change) records.

A Typical series of Colingo queries, as shown below,

```
GET AIRBASE-FILE.

IF COUNTRY EQ UNITED*STATES
AND RUNWAY/LENGTH GR 8000.

PRINT NAME LOCATION ALTITUDE
RADIO-FREQUENCY.
```

would first cause the airbase file to be retrieved from the data base and stored on an intermediary tape. Second, it would cause that tape to be read and re-written, and those records that describe U. S. airbases with runways longer than eight thousand feet would be tagged. Third, the tape would be re-read and the tagged records would be printed in the standard tabular format, as lines containing airbase name, location, altitude, and radio frequency.

d. Univac Information Management System

Imrads

The Imrads Mac (Information Management, Retrieval and Dissemination System -- Master Application Compiler) is written in Cobol and operates in conjunction with the Executive Software System of the Univac 1108 computer system. Imrads Mac is a compiler that generates Cobol Statements that will be compiled on the specified installation's computer to create its own Imrads System. Imrads Mac has the ability to design the organization and control of the data base that will be a part of Imrads, taking into account the characteristics of the storage devices available and the volumes and security classification of data to be stored. Imrads has the ability to make use of a standardized Univac procedure for organizing a data file on a specified storage device.

There are two major inputs to Imrads Mac: one names the hardware system, and the other describes the entire problem at the systems

level. The hardware includes the central computer configuration (memory, peripheral equipment, communication interfaces), and any telecommunication input/output devices. The problem description includes the total number of separate descriptions for all independent information processing operations to be controlled by a single Imrads. Each description will contain the definition of all input and output data in terms of data elements and data sets (equivalent to files, records and sub-units of records); interrelationships which exist between data elements and data sets; decision rules governing data validation, and operational procedures.

The output from Imrads Mac is a tailor-made Information Management, Retrieval and Dissemination System that includes a Process Controller, which is a resident computer software operating system, and an unfilled but defined Data Base.

Imrads itself is a transaction-oriented system, that is, the data within the transaction itself is the true dictator of the action which follows. Resident in the computer at all times is the Imrads Mac-generated Process Controller Program, which communicates with both the Executive System and the data base.

The capability therefore exists within Imrads to permit the entering of a new data file and its associated processing logic, security criteria, and retrieval schedule, etc., at any time during system operation.

The Imrads mode of operation is that of using interpretative logic for the processing of the transaction stream. In the case of a request for information, the request goes through an analysis phase in which procedural tables are generated to carry out the request.

5. CHOOSING A PROGRAMMING LANGUAGE

In the past, by choosing a computer, you automatically chose a programming language to go along with it. That simple age is past, perhaps never to return. Nowadays, a variety of programming languages are available for most of the many existing computers. Each of these languages and their processors have their own capabilities and limitations, and while these individual features can sometimes be objectively described, the evaluation of a language and its processor can only be done from the narrow viewpoint of a particular application. Unfortunately, no single, existing language is "best" for all applications.

The problem, then, is to select a language that will enhance the productivity of the computing facility that will use it. This productivity depends on the following factors: the problem(s) to be solved; the people involved; the computer(s) used; and the operational environment, including the programming language and its processor.

In choosing a language and a processor, their effects on each of the other factors must be considered and weighed.

The most important properties of languages and processors that are associated with these first four factors are: the capability of the language; the ease of learning it and of programming in it; the efficiency of the processor, the quality of its output, and the

machine-independence of the language; and the ease of translating, debugging, operating, and maintaining programs written in the language.

Measures of these properties are evaluations of the ways in which the features of the language and processor effect the other, external factors of the situation. These properties may be measured experimentally, by programming suitable test problems; they may be measured empirically, by gathering data on actual operating and programming experience; and they may be derived analytically, by evaluating the pertinent pitfalls in each approach: controlled experiments in programming are both difficult to construct and expensive to conduct; observational data can be misleading, even when available, especially if some of the pertinent factors go unrecorded; and the validity of an analysis depends greatly on who is doing the analyzing. For the best result, it is wise to try all three approaches together -- in the hope that their shortcomings will cancel out.

In taking the analytical approach to evaluating a programming language and its processor, it is first necessary to acquire or compile descriptions of them. Then, on the basis of these descriptions, it may be possible to evaluate the properties of the language and processor in relation to a particular set of external

factors.* However, unless quantitative criteria applicable to these properties are established, understood and accepted, qualitative statements concerning them will be difficult to interpret. But discovering or incorporating such criteria in the descriptions of a language and processor can be exceedingly difficult. In any such descriptions, because of the complexity of the subject, appropriate criteria for adequate evaluation may be ill-formulated, too qualitative, hidden in masses of detail, or lacking entirely.

In choosing a programming language, all the factors we have so far discussed must be considered -- machine independence, procedurality, generality, and whether the language should be an interpretive, assembly, compiler, or generator language. For general programming use, interpreters are usually too inefficient, and generators too specialized. Consequently, much of the interest here has centered around the issue of assembly languages versus compiler languages. This issue is not yet dead, despite the reams of paper and hours of talk that have so far been expended on it.

In any discussion on the utility of programming languages, it seems the main problem is that there are very few facts, and very many opinions. The following paragraphs on the relative merits of compiler languages on the one hand, and assembly languages on the

*Keeping in mind that trade-offs are often necessary between the various features of languages and their processors.

on the other -- particularly in the areas of programmer training, program production and maintenance, program communication and transfer, and program execution -- is not going to redress that balance by much.

One of the claims frequently made for compiler languages is that they are easier to learn than assembly languages. This claim is not universally valid nor even typically true. An assembly language for a simple, straightforward machine can be much easier to learn than a complex compiler language such as Algol, Fortran, or Cobol. Certainly, manuals for compiler languages are not shorter or easier to read, nor are compiler language programming courses less time consuming. The amount of intellectual effort needed to acquire professional capability for programming probably depends less, therefore, on the programming language being learned than it does on the quality of the training manual and of the instruction. Furthermore, much that a professional programmer needs to know -- the application (e.g., document retrieval), system testing strategies, other programming tools -- are largely independent of the programming language.

With regard to ease of learning, compiler languages may typically have smaller vocabularies than do assembly languages, but their syntax is almost invariably more complicated, while assembly-language syntax stays pretty much the same from one language to the

next; only the vocabulary changes. Moreover, this common syntax is extremely simple. Because of this syntactic simplicity, an experienced programmer may not even have to spend much time learning a new or unfamiliar assembly language before using it. A well-indexed reference manual may be all he needs to start coding.

On the other hand, it is undoubtedly much easier for a part-time programmer or nonprogrammer (for example, a system analyst) to learn just the basic elements of a compiler language such as Fortran than it is to learn all the details of FAP (IBM 704 Series Assembly Program).

Perhaps the major and most impressive claim made for compiler languages, as compared with assembly languages, is that they reduce significantly the amount of effort needed for program production and maintenance. The basis for this claim is twofold: (1) a source program written in a compiler language is shorter (contains fewer steps) than an equivalent source program written in an assembly language; (2) a program written in a compiler language is easier to modify than one written in an assembly language.

In terms of the total amount of computer instruction coding generated to make up a program, it is not at all clear, in general, which language type has the advantage. Procedure languages, such as Cobol, may even require more symbols or characters per program than

any assembly language. So far as effectiveness of production and maintenance is concerned, however, the main factor is not how much the programmer has to write; rather, it is the number of individual program steps he must keep track of. Consequently, for a compiler language, the appropriate unit for measuring program size is the statement; and for an assembly language, the appropriate unit is the instruction.

The instruction statement ratio varies from language to language and from compiler to compiler, but typically, a compiler will generate an average of four or five machine instructions per statement. A reduction of programming effort by a corresponding factor of four or five cannot be inferred from this, however, because the statements in a compiler language are typically two or three times more complex than the instructions in an assembly language. Furthermore, heavy use of macro instructions in an assembly language can reduce the number of steps in a program almost as much as can the use of a procedure language.

Perhaps the most important general advantage of compiler languages is that they simplify and facilitate program modification, which is important because it starts minutes after the first line of code is written and continues until the program is scrapped. An important consequence of the reduction in the number of program steps and the ease of program modification afforded by compiler languages, as

compared with assembly languages, is a reduction in the amount of effort needed for program testing, which is a vital and time-consuming part of the program production process.

The use of a slow compiler can, in certain circumstances, increase turnaround time enough to seriously retard testing; and an incompletely checked-out compiler can, by itself, make program testing spectacularly frustrating. In addition, there are many aspects of program testing -- such as test design, test data preparation, and the analysis of test results -- that are little affected by the language used to write the program. Nevertheless, program testing consists largely of error detection and correction, and by reducing the number of program steps, a compiler language reduces the opportunity for making errors.

The above arguments in favor of compiler languages as tools for program production are dependent on the applicability of the language to the problem being solved. Some languages, like Algol and Fortran, are best suited to scientific and engineering applications. Others, like Cobol, are best suited to commercial applications. Such languages can be used outside their intended areas of application, but the benefits to be derived from this are doubtful.

Compiler languages do reduce the need for program documentation, which reduction is largely caused by eliminating the need for very detailed

step-by-step program descriptions. The question of program transferability -- the ease with which an old program can be fitted into a new or different computer -- is also related to program readability. Unlike assembly languages, most compiler languages are largely (but not completely) machine-independent, and thus promise to facilitate program transfer greatly.

However machine-independent the language in which a program is written, it is seldom possible to transfer it to a different computer without making some changes to it. Assembly language programs have little transferability, except among so-called "program compatible" families of computers. So, it is almost always necessary to learn the details of an assembly-language program before it can successfully be transferred to a different machine. But with compiler language, it sometimes is possible to change a program so that it can be compiled to run on a different machine without delving into all the details of its operation -- a capability that is more likely to hold for smaller, more independent programs, and programs originally palnned for possible transfer.

Sometimes, however, even with a machine-independent compiler language, it is not possible to transfer a program without first learning the program thoroughly. This is likely to be the case with programs that are closely integrated parts of larger programs, and with programs

written to take advantage of unique machine characteristics.* In such cases, of course, the cost of program transfer is increased. Still, the greater readability of compiler languages makes learning the details of a program's operation easier, so that even here, program transfer usually requires less effort than complete reprogramming.

The efficiency of a program is customarily measured in terms of the amount of storage space it takes and the amount of computer time it uses. It is quite clear that no compiler yet built can match the efforts of a skilled assembly-language programmer in conserving these resources.

Great variations exist among compilers in the efficiency of the object code that is produced. Typical reasons for this are: more emphasis is placed on code optimization in some compilers than in others; compiler writers vary in experience and skill; a compiler that has been used and improved for several years will likely produce much better code than one that has just been released; code optimization algorithms can be much more difficult to construct for some computers than for others; some compiler languages can be more easily translated to efficient object code than can others, and some permit the use of

*Which may argue for use of machine code characteristics when using assembly language, but avoiding "tricky coding" when using a compiler language where a computer change is contemplated.

much more efficient data structures than do others. Despite these variations in object-code efficiency among compilers, some generalizations are possible. With a good compiler, for example, a poor or novice programmer will typically turn out better, more efficient code than he would using an assembler -- the compiler places many inefficient coding techniques out of his reach. An expert programmer, on the other hand, will turn out less efficient code using a compiler than he would using an assembler. But if he is using a good, well-seasoned compiler, one that has seen several years of maintenance, improvement, and use, his programs may be only 10 to 15 percent larger and slower than they would had he written them in assembly language.

In most cases, the selection of a programming language is an important decision, and should be made on as factual a basis as possible. The foregoing discussion has treated the issue of assembly languages versus compiler languages. Since there is usually only one assembly language for any given computer, assembly language selection (as distinguished from computer selection) is seldom a problem. Such is not the case for compiler languages, where, without some general guidelines, language comparison and selection can be quite difficult.

The next section presents a set of questions, which cover most of the important and relevant factors, such as the applicability,

capability, and utility of the language, and its implementation, inputs, operation, outputs, and performance, provide some of the needed guidelines for comparing and evaluating assembly languages and compilers.

6. CHECKLIST OF QUESTIONS FOR PROGRAMMING LANGUAGE EVALUATION

a. Applicability of the Language

For which applications is the language particularly suited?

For what class of machines is the language particularly suited?

For which machines has the language been implemented; or planned?

Is there an "official" version of the language?

Is there a standard alphabet for all dialects of the language or does each dialect have its own?

Disregarding differences in alphabet, is there a standard symbol set for all dialects?

Pragmatically, in terms of existing programs, subroutines, sentences, and symbols, how much of the language is acceptable to all of its compilers or assemblers?

What problems are usually involved in moving a program, written in the language, from one computer to another?

b. Capability of the Language

(1) Data Types and Structures

What data types can the language describe and process? (e.g., floating-point numeric values; fixed-point numeric values, with unrestricted and automatic scaling; complex, multiple precision, or vector values; symbolic values, in a variety of alphabets and representations, with justification options; Boolean values, categoric or status values; storage location values.)

Can single values be named and described in the language? Described in how much detail? (I.e., type; coding; machine-symbol length and format.)

Are there any limitations on the length of the machine-symbols that represent single values?

Can single values be described as composed to constituent single values?

Can arrays (i.e., structured collections of homogeneous elements, distinguished by index) of values be named and described in the language? Arrays of unrestricted structure and number of dimensions?

Can groups (i.e., collections of possibly heterogeneous elements, distinguished by name) of values be named and described in the language?

What complex data structures can the language handle? (E.g., groups of groups; groups of arrays; arrays of arrays; arrays of groups.)

Can the language handle values, arrays, or groups of fixed or variable length or number of elements? With automatic provision for dimension parameters and/or dynamic storage allocation?

Does the language automatically provide for the manipulation of arrays and groups whose elements are dynamically associated in storage by explicit links?

What controls does the language provide for optimizing data storage allocation? (e.g., packing; overlaying; relative or absolute origins.)

Can data structures apply to auxiliary storage as well as main memory. (i.e., with automatically included storing and retrieving routines.)

In describing data and data structures, does the language provide optional levels of detail?

(2) Operations on Data

Are there any restrictions on the data types and structures that can, in general, be designated as arithmetic, relational, Boolean, or data movement operands? (e.g., only single-value operands; only operands of similar type, or like coding or representation.)

Can arbitrary subsets of the sign strings (bits, characters, digits) that comprise machine-language symbols be designated as operands? Can these designations be dynamically varied?

Can arbitrary subsets of the elements that comprise arrays or groups be designated as operands? Can these designations be

dynamically varied? Can they be dynamically determined? (i.e., by automatically included search routine.)

Does the language include a complete set of arithmetic operators? (i.e., addition; negation and subtraction; multiplication; division; exponentiation; absolute value.) In any context where a numeric value is computed, are there any further restrictions on the operands or the complexity of numeric formulas?

Does the language include a complete set of relational operators? (i.e., is equal to; is not equal to; is greater than; is less than; is greater than or equal to; is less than or equal to.) Are there any further restrictions on the operands or the complexity of relational formulas? (e.g., only one operator per formula.)

Does the language include a "partial" (first 3), or "complete" set of logical operators? (i.e., and; inclusive or; exclusive or; not; implication; equivalence.) Are there any further restrictions on the complexity of Boolean formulas?

What data movement operators does the language include? (e.g., substitution, with editing or conversion; permutation -- exchanging; sorting; insertion and deletion; storage and retrieval for auxiliary storage.) What further restrictions govern the use of the data movement operators? (e.g., only one operator per sentence; or fixed-length memory word limitations.)

(3) Decisions

Can the language express both fixed and dynamically-variable transfers of control? Transfers to what types of program element?

Can these decisions be made during assembly or compilation, or dynamically during execution?

(4) Loops

Does the language's loop generation mechanism provide for an implicitly varied loop parameter? Can more than one parameter per loop be implicitly varied?

Are there any restrictions on loop parameters? (e.g., data type).

Are there any limitations on the number of iterations for loops, or on the construction of loops within loops?

Does the language restrict transfers of control into or out of loops?

Does the language provide for transfers to the implicit loop-control routine from within the loop?

(5) Subroutines

Does the language provide for both "open" subroutines (inserted directly for each call) and "closed" subroutines (inserted once and linked to each call)?

Does the language allow function-type subroutines?

Are there any limitations on the number of subroutine parameters? Must they all be listed in the subroutine call?

Can the language describe formats for subroutine calls, or must these adhere to a rigid format?

Can subroutines be nested? Can they call other subroutines?

Can subroutines call themselves (recursively) and, if so, does the compiler automatically provide for keeping track of their parameters?

What environment elements can be subroutine parameters? (e.g., values; arrays; groups; clauses or sentences; subroutines.)

Are parameters called by name (where the subroutine uses the parameter clause itself) and/or are they called by value (where the subroutine uses a copy of that which the parameter clause expresses)?

Are there any distinctions made between input and output parameters?

Are environment elements outside the subroutine available within it?

Does the language allow for alternate entrances to a subroutine?

Does the language allow for alternate entrances to a subroutine? For alternate exits, or for transfers of control to the subroutine exit from within the subroutine? For alternate returns from closed subroutines?

(6) Input and Output

How does the language provide for input and output? (i.e., general file or format description declaratives and input/output imperatives; machine-dependent language features; general input/output subroutines; machine-dependent subroutines; machine coding.)

Is the language restricted to a particular set of input/output devices or is it general enough to handle, without major extensions, most devices? (e.g., magnetic tape; paper tape; on-line typewriters; magnetic drums, disks, and other "addressable" storage devices; punched cards; printers; plotters; display equipment; data transmission equipment; console switches.)

Does the language provide for efficient utilization of random-access devices as well as serial input/output devices?

Does the language allow the utilization of other special input/output hardware features? (e.g., alternate recording modes; high speed or block transfer; search or positioning operations; scatter read/write; backward read/write; writing without affecting subsequent records; sensing external conditions; overlapping of read/write and computing; special character sets.)

Does the language provide imperatives for, or declaratives that control the automatic inclusion of, standard file control routines? (e.g., for: buffering; allocation of input/output areas; blocking and unblocking; label preparation and checking; error detection and correction; checkpoint and restart procedures; tape alternation for multiple-reel files; handling optional files; handling multiple files on single input/output units; record positioning; file rewinding and locking; sensing end-of-file or special record partition flags.)

Does the language provide for communicating with the computer operator?

Does the language allow input/output assignments to specify input/output media? To specific physical or logical units?

How independent of input/output device assignments are the input/output coding imperatives?

What data structures are permitted as operands in input/output imperatives?

Can a file have different record formats and sizes? Does the language facilitate distinguishing between different record types in the same file?

Does the language provide for the description of report formats and external data codes? With automatic code conversion or editing between internal and external formats? With other features to facilitate report generation?

(7) Miscellaneous Capabilities

Does the language allow the utilization of special machine features? (e.g., real-time clock; program interrupts; parallel processing; specific machine registers; repeat operations; search operations; logical operations; conversion operations.)

Does the language allow the inclusion of machine code? With a facility for referencing data described in the source language?

Does the language allow the automatic presetting of initial values or constants prior to program execution?

What implicit subroutines are incorporated into the structure of the language?

Does the language provide control over truncation, rounding, and overflow or underflow? Can integer truncation be expressed?

Are there declaratives for describing the equipment that is available (or necessary) to run the object program? Is the adaptation to (or check for) the available equipment done automatically, at compilation or assembly, or dynamically, at execution.

Does the language provide for the expression of optimizing information? (e.g., expected use of data; expected range of values; expected use of imperatives.)

What other features does the language incorporate?

c. Utility of the Language

(1) Readability

How much of the language's vocabulary of symbols is familiar? (i.e., from ordinary algebra or from English.) How many of these familiar symbols are reasonably self-explanatory alone, or in their immediate context?

Are all identifiers or names arbitrary alphanumeric symbols? Of unrestricted length? With provision for separating multi-word identifiers?

What program elements can be identified (named) in the language? Are the elements decidable from the names alone? Are they decidable from their immediate context? (e.g., from the sentence containing them.)

Can comments be inserted between sentences? Between symbols?

How free of local ambiguities is the language? (i.e., where the meaning of an expression depends on information outside its immediate context.)

(2) Learnability

What documents are available describing the language, and at what level of sophistication are these? (e.g., reference manual; formal description; training manuals for programmers and nonprogrammers.)

How many different symbols does the language have in its vocabulary?

How free of arbitrary restrictions is the language? (i.e., where the correctness of an expression depends on information outside its immediate context.)

(3) Usability

Brief summary of programming experience with the language.

Does the language allow the automatic inclusion, in programs, of imperative sentences and/or clauses from a "library?" Of declarative sentences and/or clauses?

How large a corpus is available of programs and subroutines that are written in the language?

Does the language incorporate features useful for debugging?

Does the language have features that facilitate automatic program segmentation? That facilitate team programming? (e.g., reserved domains for definition of identifiers.)

Does the language have features that facilitate the identification of programs, and the tapes, decks, and listings associated with them?

Is the language such that declaratives describing details and constraints on the problem solution are easily separated from those declaratives essential to the method and procedures of solution?

Can the programmer describe computations to be performed by the compiler, as well as by the object program? (e.g., in specifying the dimensions of data structures.)

Does the language allow manual, automatic, or dynamic storage allocation? Are there options?

Can declarative and/or imperative sentences be "nested?" To any level?

What percentage of the symbols in a "typical" program are "key" symbols? (i.e., symbols with a fixed spelling, not programmer created.) What is the average size of the key symbols in such a program?

Does the language have features that facilitate program transcription? (e.g., optional omission of repetitive or redundant expressions.)

Can permanent and/or temporary synonyms and abbreviations be created? Dynamically as well as manually or automatically?

How free of syntactic and semantic redundancies is the language? (i.e., where expressions must agree in syntax or in meaning, so that otherwise correct expressions could be incompatible.)

d. Characteristics of a Compiler

(How) is the compiler related to similar or previous compilers?

Who constructed the compiler? Who maintains it? When was it released for operational use and/or maintenance?

On what basis is the compiler available for use by organizations other than the constructors/maintainers?

Is the compiler's design relatively stable with respect to the source language it accepts and the target language it produces? With respect to its performance and operations?

How large is the machine language version of the compiler? (i.e., total number of instruction words and constant data words.) How much of it consists of standard, "off-the-shelf" modules?

Is the declarative information in the source program, describing its environment, used by the compiler in producing the object program, or is it used dynamically, by the object program itself? Is there an option?

Does the compiler perform any additional optimization of storage allocation (other than expressed in the source language)?

Does the compiler dynamically allocate storage among the various internal tables and lists its uses during compilation?

What is the mode of operation for program compilation and execution? (i.e., compile and execute: must be separate runs; must be in sequence; are identical, via interpretation.) Are there any options?

Does the compiler allow batch compiling? Can it then provide for job accounting?

How many passes does the compiler require in translating source language to the final target language? (i.e., where a pass consists of the processing and transformation of all or most of the program compiled.) Is there provisions for "restarting" with any of these passes?

How much console operation does the compiler require? (i.e., on the average, how much time per compilation can be charged to operator set-up and intervention?)

Is there an "error check only" compilation mode? How is it selected?

What source language input errors does the compiler recognize? (e.g., missing or incomplete expressions; duplicate identifiers; improper format; excessive target computer storage; program inconsistencies; program inefficiencies.) What action(s) does the compiler take upon encountering such errors? (e.g., skip to the next job; stop compilation; stop and wait correction; stop and offer continue option(s); continue in checking mode; ignore error and continue; flag error and continue; continue with "most probable" correction.)

What known, source language input errors does the compiler fail to recognize? What known, source language errors will cause "catastrophic" failures in compilation? (e.g., complete termination of compiling; unflagged and erroneous output; unexpected compiler halt or hang-up.)

Does the compiler allow the compilation of subprograms, and their automatic integration with previously compiled parts?

Does the compiler impose any limitations on the size of the source language input program. (e.g., maximum number or size of: imperative sentences, identifiers; constants; data dimensions; various declarative sentences.) Are there any grouping or sequencing restrictions?

What physical forms may the compiler's inputs have? Are there any options? Are there any format restrictions?

Does the compiler accept any other programming languages as input? Are they sensitive to each other's declaratives? Can sentences from these languages be arbitrarily interleaved?

What other inputs does the compiler accept or require in addition to a source language program. (e.g., compiler control information; test data; debugging requests.)

What computer(s) does the compiler run on? What is (are) the minimum required configuration(s)?

Can the compiler be easily adapted to different configurations? (e.g., to take advantage of more equipment; to change input/output assignments.) Does it check the machine configuration before operating to see if enough hardware is available?

Does the compiler operate in conjunction with a monitor or executive system? Is this optional?

Can the compiler be used on-line, in a conversational, interactive, or incremental mode of operation?

Does the compiler use a subroutine and/or data description library?

Are there supporting utility programs available that allow the use of source language throughout program development, especially during debugging?

Are there supporting utility programs available to facilitate compiler maintenance?

Does the compiler serialize source language sentences or lines for reference and to provide insertion capability? Does it sequence an already serialized source language program and then reserialize it to renew insertion capability? Is there provision for compiling alterations and patching these into an intermediate version of a previously compiled program?

Does the compiler provide for the inclusion of debugging requests in the object program? (e.g., instruction traces; data reference traces; dynamic dumps; post-mortem dumps.) Are these requests and the resulting outputs in source language terminology?

Is there particular emphasis in the compiler on economizing compilation time, object program running time, or object program storage space? Can this emphasis be selected?

Brief summary of operating experience with the compiler: questions similar to those involving programming experience with a language.

What target language does the compiler produce as its final output, and what kind of language is it? (e.g., absolute machine language for execution; relocatable machine language for loading; symbolic, machine-oriented language for assembly; intermediate language for further processing; pseudo-language for interpretation.)

What intermediate language(s) does the compiler produce as incidental output(s)? Are these mainly for reference or are they in a form that would allow further independent processing?

Does the compiler produce a listing of the source language program? Of the target language object program? How are these listings cross referenced? Is any information included to

facilitate tracing the program execution flow? (e.g., lists of entrances, exits, and other intra-program references.) Does the listing include, for each source language sentence, the amount of storage space thereby allocated and, for imperative sentences, the minimum and maximum execution time?

Does the compiler produce a listing of data references? How is this listing organized? (e.g., by set-used criteria; by referenced and/or referencing element.)

Does the compiler produce a "storage map," listing identifiers and the storage areas associated with them, and showing buffer areas and data and program overlays?

Does the compiler produce a listing of all errors encountered in the source program? Does it produce a listing of possible errors that may have produced erroneous target language coding? How does the error listing locate and explain the source language error?

Can the programmer select the outputs from a compilation?

Can the compiler-produced programs be easily relocated in memory? Can they be easily adapted to different physical or logical configurations of the target computer? Do they check for an adequate or appropriate configuration before operating?

Does the compiler produce re-entrant programs? (i.e., in a time-sharing or multiprocessing computer, a program that may be re-entered before it has completed its previous operation. A re-entrant program does not modify itself and, whenever it is entered, if it has any work in progress, it will either save the current contents of its working storage area or turn to a fresh working storage area.)

Can the programs produced on the compiler be run under available monitor or executive control? Are there any options?

What physical forms do the compiler's outputs have? Are there any options?

What is the compiler's average compilation speed (e.g., in instruction and constant words, output per minute) and how is this related to program size? What other factors greatly influence this speed?

What are the average expansion ratios between the source-language sentences and symbols input and the target-language sentences and symbols output by the compiler?

Are there methods for predicting, from a source-language program, the approximate size of the resulting target language program? How much storage space does the compiler allocate for fixed overhead, input/output buffer areas, etc.?

How much of the target computer's command structure does the compiler utilize in the programs it produces? How efficient a program does the compiler produce, in terms of execution time and storage economy, relative to other methods of program preparation? (e.g., comparable to, significantly better than, significantly worse than a "good" programmer using assembly language.) How much does this depend on the size and type of problem and the quality of source language programming?

Does the compiler take advantage of the absence, in the source program, of unusual or complex language forms in order to produce a swifter compilation or a shorter or swifter target program? Does the compiler use any of the optimizing information that may be included in the source program?

How much optimization of target-language coding does the compiler perform? (E.g., elimination of: duplicate constants; unnecessary store and access operations; operations for evaluating duplicate formulas; the execution of unnecessary sub-formula evaluations in conjunctive and disjunctive Boolean formulas.)

How reliable is the compiler? (i.e., how many compiler errors requiring correction were reported during the last month's operation, and how many words or other units of output does this represent?)

e. Characteristics of an Assembly Language

The state-of-the-art of computer assembly languages is such that certain common features are normally found in assembly language software delivered by the manufacturers of medium- and large-scale computers. These features constitute, in effect, a basic capability which a manufacturer should provide.

A one- or two-pass assembly program which:

Accepts magnetic tape or punched card inputs

Allows batched assembly of groups of source programs

Accepts, and assembles, machine code and symbolic code inputs

Produces assembled (coded) program output on magnetic tape or punched card or magnetic drum/disk media.

Produces, for printing (or storage on input/output media), listings of program assembly results, which:

List assembled instructions in symbolic and machine code form

Include programmer-produced comments

Indicate clearly any detected coding errors, designating the type(s) of error detected in each case

Contain tabulated listing(s) of each programmer-defined data constants and of programmer-defined instruction addresses, with the lists indicating each instruction referencing a constant or instruction address.

May be partially or totalled supressed form printing in accordance with installation practice.

In addition to these basic assembly language features, the following software elements should be supplied as part of the assembly language package.

A magnetic tape containing a library of assembly software, including:

A memory dump subroutine, allowing post-mortem and operational (snapshot) prints of selected portions of memory;

A generalized report generator, allowing page titles, alphabetical and/or numerical column headings, multilevel subtotals, and page breaks on subtotal and total limits.

A macro instruction generator, allowing macro instructions to be included by the installation.

A selection of subroutines, macros, and utility programs, appropriate to the needs of the installation application.

This discussion so far has emphasized computer instruction capabilities. The other aspects of major concern in assembly language evaluation are the representation and processing of data in memory (and in peripheral and storage devices). This is partly a hardware problem but a good assembly language system should provide efficient:

Internal packing, unpacking, overlaying, and conversion of data

Transmission of data between locations in memory

Sorting, merging, and collating of data

Ease of programming in allowing simplified definition of repetitions or lengthy data formats.

Evaluation also should include cognizance of the timeliness of quality, and depth of written materials explaining the assembly language system, including handbooks; programming manuals; machine logic materials; system utilization and application experience summaries, etc.

f. The Major Factors in Language Selection

These may be summarized as falling into three major classes:

Quality, cost, and timeliness of availability of the competitive language software.

Compatibility and ability to convert coded programs for execution on different hardware.

Efficiency of the programs produced (where equipment utilization and installation application so require), and

Ease of use (in re-education, training, debugging, and coding) of the language.

The Defense Documentation Center should consider all of the factors cited in the above sections in evaluating software systems for use on new computer configurations the Center may wish to acquire or use. When the choice devolves to use of an assembly language (the current situation) or a computer language, the following steps will aid in the evaluation process.

FIRST: Code up a small but representative set of test routines in the source language of the compiler, along with some test data for these routines. These routines should be designed with at least four purposes in mind:

To determine whether the compiler accepts the full source language.

To determine whether the object code produced by the compiler does the appropriate calculations.

To determine the speed of the compiler.

To determine the efficiency of the object-code produced by the compiler, in terms of its use of storage space and execution time.

For this last purpose, the test routines should include sentences that reflect the kinds of applications for which the compiler will be used as well as sentences that, based on experience with similar compilers, can be expected to produce the most troubling cases of inefficient object-code.

SECOND: After the test routines have been written in the compiler source language, recode them in the assembly language of the target computer. Assign a senior programmer to the task (one with experience in programming for the target machine), and tell him to turn out the tightest code possible, using all the facilities the machine provides. (Questions of tradeoffs between storage space and execution time that arise here should be resolved with the same criteria you would want used in your operational programs.) Since almost any code can be improved, you might want to repeat this step a few times, until you're

confident you've squeezed out all the fat in the assembly language routines. You will, of course, check these routines out with the test data to see that they too perform the appropriate calculations.

THIRD: These assembly language routines are then your standard of coding excellence, and you can specify that the compiler you're buying turns out object-code that averages no more than x percent more total instructions (and constants) and y percent more total execution time, using the same test data (x and $y < 0$).

The question now naturally arises, what are reasonable values x and y to ask for? This answer depends on a lot of factors, not the least of which are the test routines themselves. But perhaps some general answer is possible, assuming that all you want is a reasonably efficient compiler, for a reasonably straightforward source language, on a reasonably ordinary machine, and at a reasonably competitive price. Under these assumptions, a good guess, for x would be from 20 to 30 percent; for y, from 10 to 20 percent.

The evaluation methods outlined above fall within the following six progmatic objectives, which should be met by DDC in its evaluation and selection of computer software.

- Any new software systems should entail a minimum of maintenance, programmer education, and modification by the DDC data systems staff.

- The language selected should be one produced by all of the major computer manufacturers.
- The language should be of sufficient stature, and widely enough used, effectively to guarantee that the major portion of the language will be included in future languages of the type predicted here.
- The hardware manufacturer must be able to deliver the language (in debugged, post-field test condition) to DDC when hardware conversion is approved. This means availability of the software and explanatory manuals, etc., prior to equipment delivery, eliminates as much as twelve months of concurrent software field testing; software modification; programmer education, and reprogramming of the current system.
- The computer coding produced (and provided) by the software system should be efficient in terms of speed of program execution, and in utilization of memory and storage devices.
- The system must include the current DDC software capability of concurrent execution of programs by the computer (parallel processing or multiprogramming).

7. PRESENT AND FUTURE TRENDS IN PROGRAMMING LANGUAGES

The major trend in computer programming languages today, and for the last ten years or so, has been the shift in the proportion of use from the highly procedural and computer-oriented assembly languages, to the less procedure-oriented and more computer-independent compiler languages. Two underlying trends have been the primary causes of this shift: improvements in compiler technology and greater availability of compiler building capability; steadily decreasing computation costs and a correspondingly increasing concern for programming and reprogramming costs. These underlying trends should continue at an unabated and perhaps even an accelerated rate, thus guaranteeing that the shift in favor of compiler languages over assembly languages will continue. What this means in practice is that, compared with assembly languages, compiler languages will be more attractive for use in an increasing percentage of applications.

In addition to this long-term trend, there are three more recent trends in computer-independent programming languages that deserve consideration. The first two are seemingly contradictory -- the trend toward increased specialization, as exemplified by the various programming languages being developed at MIT under Project MAC, and the trend toward increased generality, as exemplified by the (IBM) Programming Language 1 (PL/I). Actually, this seeming contradiction is easily explained. Existing programming languages, like Cobol, Fortran, and Algol, are well established. To be successful, a new programming language must be clearly

superior, or it must be either more general or more specialized than any of the established languages.* And, in a sense, the impulse toward programming-language specialization is merely a reflection of a more basic impulse toward generalization -- the generalization of application programs.

As a particular area of application of digital computers becomes well known and better defined, as more and more programs are produced for that area, and as these programs become more general-purpose in nature, the input languages and the parameters that control their operation become correspondingly more expressive. And the natural next step is to increase the expressiveness of these input-parameter languages and combine them into a full-fledged, though problem-oriented programming language. Such problem-oriented languages are ordinarily implemented either by interpreters or generators.

It is quite reasonable to expect that the rapid proliferation of problem-oriented programming languages and the slow (but seemingly steady) introduction of more universal languages (such as PL/I) will have a somewhat inhibitory effect on the growth in the use of such established compiler languages as Cobol and Fortran. This is far

*Actually, of course, a great deal of experimental work is being carried out with the aim of improving programming languages, without necessarily either specializing or generalizing their applicability and this is particularly the case, for example, with Algol. But much of this work is a long way from practical utility.

from being a prediction that Cobol and Fortran are soon destined for replacement. For one thing, there is the already mentioned counter-trend toward the increasing use of compiler languages. Yet in the long run, it seems clear that the conventional programming needs of most installations can be met by a universal, compiler language (such as PL/I or its successor) coupled with a careful selection of problem-oriented, interpreter or generator languages. The ordinary application programs would be written in the appropriate problem-oriented language, while most other programming would be done in the compiler language. It is, however, in the coupling of the compiler language with the various problem-oriented languages wherein lies the greatest unsolved problems and consequently the greatest unfulfilled opportunities in programming language development. Let us look then at what is involved in synthesizing the antithetical trends toward programming language specialization, and programming language generalization. To synthesize these two trends, to couple a general-purpose compiler language with a variety of problem-oriented languages, it is the compiler language that must be extended.

To be suitable for such systems programming, a compiler language needs facilities for expressing asychronous processing, the manipulation of symbols and expressions, and of strings of characters (from a variety of alphabets) and even of bits, and the manipulation of data elements linked explicitly together in lists and list structures. Second, and

perhaps of primary importance, the compiler language should provide for its own extension, not just by the addition of new subroutines to the library, but by the addition of new data types and structures, as well as the operations over them. Such extensibility can only partly be provided by incorporating into the compiler language a conventional macro definition and expansion capability, which is essentially just sophisticated, symbolic editing. What is really needed here is a capability for specifying translation rules not just in terms of the symbology of the compiler language, but in terms of both the symbology and the terminology of both the compiler language and the target language (and perhaps also of any internal or intermediate languages used by the compiler). With such a scheme, a translation rule could specify symbols and terms (designating entire classes of characters, symbols, expressions, sentences, etc.) from the compiler language and from the target language. To implement a problem-oriented source language with such a capability, it would be necessary to provide a set of translation rules to the compiler, probably as a library file. The compiler would interpret these rules first in translating source programs into the compiler language, into any intermediate languages, and finally, into the target language.

It is quite likely that a syntax-oriented compiler would be needed for the realization of such an ambitious translation scheme, but the real challenge is to provide this capability in a convenient, easily

understood package. The specification of complex programming language transformations is evidently going to remain the job of the professional systems programmer for some time to come.

This brings us to the third recent trend -- the trend toward user-oriented, conversational programming languages. This trend has been enabled by developments of the past few years in computer time-sharing, which makes on-line access to a large, powerful computer, both economic and attractive for a wide variety of users. Some definitions are required, however, before we can discuss this trend.

By a user-oriented programming language, we mean one that is well designed for use by nonprogrammers -- that is, by people who are not professional programmers.* By a conversational programming language, we mean one that is well designed for on-line use -- a language that permits the processor to carry on an instructive dialogue with the user (usually on a sentence-by-sentence basis) while he is composing his program.

Aside from careful and thoughtful language design (which is of primary importance here) there are two approaches to achieving even greater user acceptance of such an advanced language. The first approach is increased specialization. The more problem-oriented a programming language is,

*Since anybody who uses a programming language to communicate with a computer is, by definition, programming, and in a sense therefore, a programmer.

the greater its acceptance is likely to be among users concerned with that particular problem area. A programming language for cryptographic problems, for example, might be complex enough to daunt a professional programmer and still be enthusiastically adopted by professional cryptanalysts.

The second approach to user acceptance is decreased procedurality. It is not the concept of programming that repels users, rather, it is the excruciating level of detail required by most programming languages. Nonprogrammer users are more apt to accept a programming language if they can describe their data simply and naturally, and if they do not have to specify the processing of each individual value, but can specify operations over aggregates of values. Of particular value here are the file processing languages, which relieve the user of concern with the details of transferring blocks of data among tape, disk (or drum), and core storage. This greatly eases the burden of programming and, from the viewpoint of the nonprogrammer, who would balk at learning a conventional programming language, it makes a file processing language worth learning. (File processing systems and languages are discussed in detail in another portion of this software analysis.)

With regard to the acceptance of a programming language by nonprogrammer users, we have mentioned the importance of excellence in design, and we have discussed the effects of increased specialization and decreased procedurality. Easily equal to design excellence in the importance of

its influence on user acceptance, however, is the factor of interaction in the language between the user and the language processing system. A language that permits a conversational interaction whereby the user receives immediate feedback and continual guidance from the system can dramatically increase user acceptance. The trememdous potential of on-line, computer-aided instruction in the use of programming languages has hardly been exploited. Thus, although a user-oriented, conversational programming language can be just as complex as a more conventional programming language, on-line instruction and guidance can make it easier to learn and use, while the fact that it may be more problem-oriented, dealing well with a particular problem area, and less procedure-oriented, dealing with such data aggregates as files and eliminating such bothersome details as data transfer considerations -- all this can make it much more worth learning and using.

We have discussed several important trends in computer programming languages: the trend from assembly languages to compiler languages, the antithetical trends toward more specialized and more generalized programming languages, and the trend toward user-oriented, conversational programming languages. Let us consider now, in broad outline, the kind of programming system that could result from these trends -- hopefull, within the next few years -- that would be applicable to the needs of the Defense Documentation Center.

First of all, the system would provide a capability for on-line, multi-access, and conversational interaction with a time-shared computer. An on-line capability is necessary for those users who need quick and unpredictable access to their files. A multi-access capability is needed to make the on-line operation economically feasible and to facilitate the solution of problems requiring inputs from multiple sources. Of course, rapid, on-line service isn't always possible, particularly when the user has called for a lot of processing. For these situations, there must be an off-line, batch-processing capability as well, so the user can give the system a job, ask for an estimate of how long it might take, indicate when he wants the results, and how to format and produce final products. He would be able to come back at any later time, ask for a display of the current status of his job, and change the deadline if he wants to.

Next, the system would be a generalized, user-oriented system. It would have an ability to handle a wide variety of complex tasks, yet it would be easy enough to use that a nonprogrammer could specify his own application with very little specialized training. Since "user-oriented," to most users, is almost a synonym for "inexpensive," the user's terminal is likely to be a simple typewriter-like device for the entry and display of textual data.

The system would provide long-term continuity of application. Continuity of application means that if the system has to shut down, for preventive

maintenance or any other reason, it can employ restart and checkpoint procedures to resume operation at the point of interruption of processing.

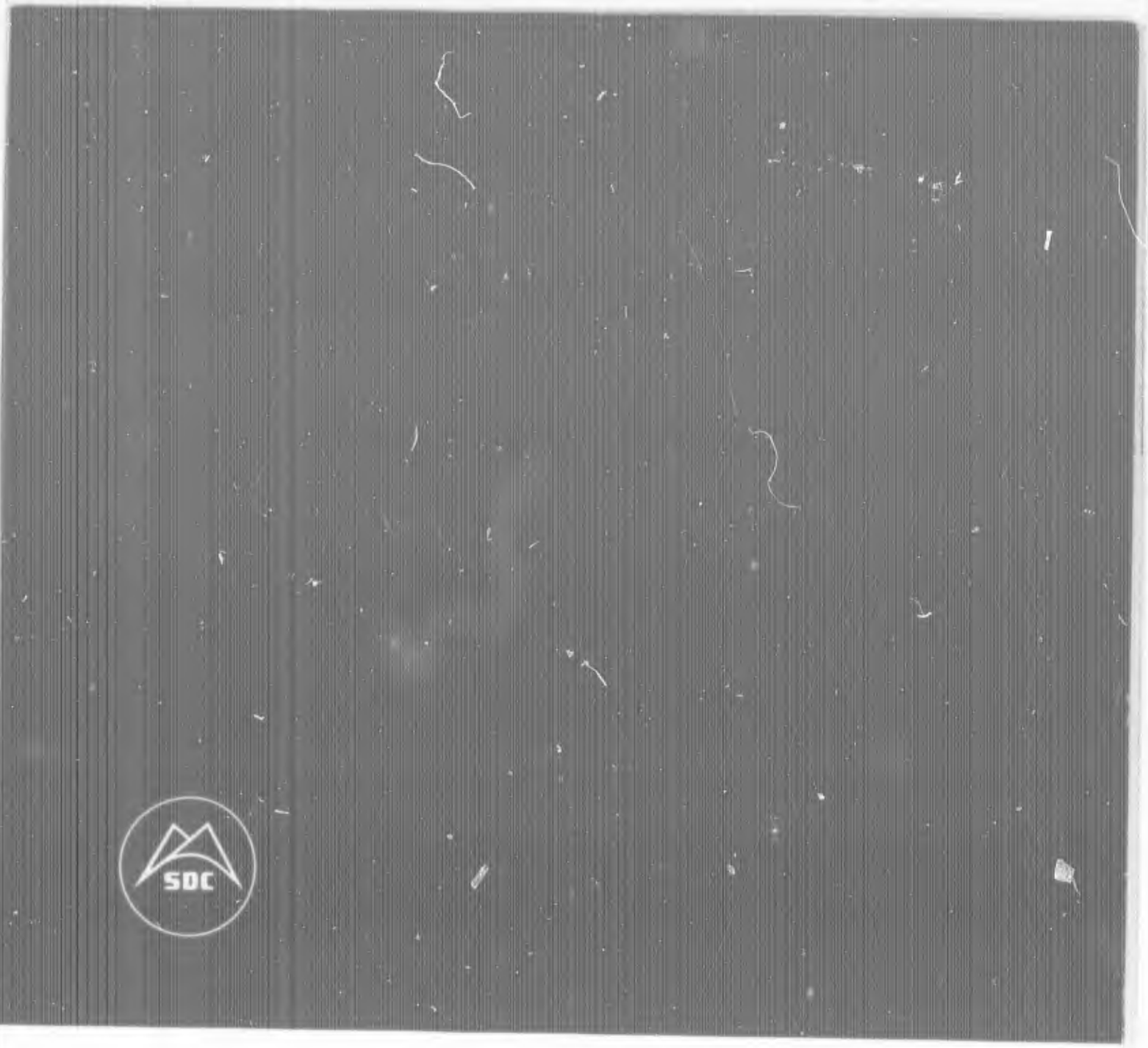
The system would permit many users and applications to share programs and files. This means dynamic sharing, so that several users can concurrently access the same program or file. There would be obvious restrictions, so that a user could not retrieve data that are being altered, or alter data that are being retrieved. Also, protection mechanisms would be necessary to prevent unauthorized access.

The system would provide a file management capability for both data and programs. The management of data files is the primary capability required by DDC. But programmer users will also have to maintain files of programs, and even non-programmer users will be interested in keeping files of prespecified operations, to have them on call when they are needed. The ability to do on-line or off-line filekeeping operations on data and programs and on other system files is quite important. It will allow the user to request reports and displays on his system and on his application, as well as on his data, thus making both system and application, in a sense, self-documenting. It will also allow him to easily change and evolve both his system and his application. System evolution means that the system is open ended; the user can add any arbitrary program to his files and, through the system, supply that program with data and call on it to be performed. Application evolution, on the other hand, means that the user -- the nonprogrammer user --

can change the operations in his application, or add or delete them. It also means that he can change the formats of his files, or of his inputs or outputs, by adding, changing, or deleting elements of data.

The system would be largely machine-independent in design and implementation. The ordinary user would not have to be concerned about what kind of machine the system is running on. The external characteristics of the system would almost completely submerge the hardware characteristics, and this would probably hold true for much of the internal design, too. The system programs would be written in a relatively machine-independent compiler language, for example, Cobol or PL/I. All this should make transferring the system to a new computer a lot easier than recoding it.

This system of the future, which may be available within five years, represents an operational synthesis of software elements currently extant, but not yet combined into a total computer software system. This synthesis will require a concentrated effort of computer manufacturer(s) software personnel, and will not be available within the next three years.



System Development Corporation/5720 Columbia Pike/Falls Church, Virginia 22041