ESD-TR-65-171

AD614704

ADVANCED PROGRAMMING DEVELOPMENTS : A SURVEY

FEBRUARY 1965

Best Available Copy

DIRECTORATE OF COMPUTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

and

COMPUTER ASSOCIATES INCORPORATED
WAKEFIELD, MASSACHUSETTS

COPY 2 OF 3
HARD COPY $ 4.00
MICROFICHE $ .75

DDC
RECEIVED
MAY 11 1965
DDC-IRA E

Best Available Copy

[redacted]

20040826032

HEADQUARTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE

LAURENCE G. HANSCOM FIELD BEDFORD, MASSACHUSETTS 01731

REPLY TO ATTN OF: ESRC

25 February 1965

SUBJECT: Advanced Programming Developments

Best Available Copy

During the past year the Directorate of Computers has conducted a survey of certain advanced programming developments which have been or are being implemented in the United States. The results of this survey have been gathered together into the volume accompanying this letter. Because of your demonstrated interest in advanced programming and data management systems, we are sending you a copy of the survey in hopes that you will find it of some interest. Any comments you might have will certainly be welcomed.

FOR THE COMMANDER

Paul G. Galentine Jr

PAUL G. GALENTINE, JR.
Colonel, USAF
Director of Computers
Deputy for Engineering & Technology

1 Atch
Survey

ESD-TR-65-171

ADVANCED PROGRAMMING DEVELOPMENTS : A SURVEY

FEBRUARY 1965

DIRECTORATE OF COMPUTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

and

COMPUTER ASSOCIATES INCORPORATED
WAKEFIELD, MASSACHUSETTS

ABSTRACT

This document constitutes a representative survey of twenty computer software systems which have been developed within the last decade. The surveyed systems have been grouped into six major categories: (1) general purpose programming and executive systems; (2) functional systems; (3) man-machine interface systems; (4) special purpose programming systems; (5) time-sharing systems; (6) generalized data management systems.

Each system discussed within these categories exhibits a particular feature or set of features which constitutes a distinct contribution to the effort to produce more general, more flexible and less job dependent systems which can be conveniently operated by a user.

The survey notes that to date efforts to develop these features into integrated computer systems and sub-systems exist only on a small scale and in only a small number of laboratories scattered across the country. In view of the many potential applications of such integrated, generalized computer systems, especially in military environments, the survey concludes that the time is ripe for a technical program designed to demonstrate the feasibility of generalized computer systems and sub-systems in the solving of traditional data processing problems.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

Paul G. Galentine Jr.

PAUL G. GALENTINE, Jr.
Col., USAF
Director, Directorate of Computers

CONTENTS

		Page
Introduction		1
Note		6
Part I.	General Purpose Programming and Executive Systems	7
Part II.	Functional Systems	27
Part III.	Man-Machine Interface Systems	52
Part IV.	Special Purpose Programming Systems	69
Part V.	Time-Sharing Systems	74
Part VI.	Generalized Data Management Systems	83
Conclusions		91
Bibliography		94

INTRODUCTION

Computer programs are detailed instructions written to adapt a completely general purpose machine to a specific problem or application. If an engineer has a mathematical problem to be solved, he must write a sequence of instructions to control the arithmetic and logical circuitry of the machine, thus formulating a specific problem solving procedure. This sequence of instructions constitutes a program which must be executed or run on the machine to obtain an answer. Similarly, if a manager wishes a report on inventory containing extractions from several files of data in memory, he, too, must write down a sequence of instructions to control the same arithmetic and logical circuitry of the machine to search tapes, retrieve data, format it and activate printout devices to obtain the desired result. Next, this program must be executed.

There is another type of computer program, however. Designed and written by computer specialists, this type of program, often called "software," is intended to provide the engineer or the manager with more than just an empty machine. Software programs are tools in the form of programs which the engineer or the manager can use for building, controlling and modifying the complex sequences of problem solving procedures required in his sophisticated computer applications.

Clearly, then, the motivation for the development of software and, indeed, its contributions have been:

(1) Improving the ease with which application programs can be written or modified and users' problems solved.

(2) Reducing the cost of constructing, modifying and executing application programs and solving users' problems.

Following is a review of a representative though by no means exhaustive cross-section of the major software developments that have been achieved recently. The systems discussed are sufficiently contemporary to be regarded as exemplifying current technology and sufficiently sophisticated to

be regarded as advanced state-of-the art. For some readers there will be no mention of some systems and too little mention of others. This is due largely to time, space and available material limitations.

The twenty-plus systems under review have been grouped into five separate categories, and a sixth special category, the significance of which is to be seen in the light of the first five categories. These categories are: (1) general purpose programming and executive systems, (2) functional systems, (3) man-machine interface systems, (4) special purpose systems, (5) time-sharing systems, and (6) generalized data management systems.

A person familiar with these systems will notice a discrepancy in the groupings. Thus COLINGO, listed under (2), functional systems, could just as easily have been included under (3), man-machine interface systems. Similarly, BASEBALL, listed under (3), man-machine interface systems, could have been included under (2) functional systems. But these discrepancies are more apparent than real.

To begin with, the categories identified in this document do not refer to characteristics which the systems under one category possess and which all other systems do not possess. Rather, the categories represent various perspectives, or points of view, which system designers chose to emphasize in the design of their systems.

Thus, the categories which have been established are intended to indicate a designer's preference with respect to that which is of primary importance in the construction of automated data processing systems. In this light, then, CL-II, AOSP, DODDAC, NECPA, OASIS, and INTIPS, all examples of general purpose systems, have been the products of designers who have determined that extended and sophisticated programming, executive, and utility systems are the key to more flexible, more powerful computer-based systems. In particular, the design of general purpose executive systems is accomplished without reference to a specific system or application. In fact, taking into account the specific requirements of a particular application, such as Hq USAF or Hq USSTRICOM, would defeat the purpose of general purpose systems which are intended to provide an integrated programming tool that can be easily used by and, therefore, of assistance to programmers in the process of designing, preparing, and executing specific sets of programs required by specific applications.

In contrast to the general purpose point of view which emphasizes general programming tools quite apart from the requirements of a specific application,

there is the category of functional systems which includes the COLINGO, NAVCOSSACT, TUFF-TUG, FICEUR, 473L, and ACSI-MATIC systems. These systems have been designed from a point of view that stresses a particular function, e.g., information retrieval. In this case a system must be primarily constructed around the function which it will be called upon to exercise more often than any other function. This functional approach, which by no means ignores the advantages of general purpose program design, chooses to measure all its design decisions in the final analysis, not by how much general-purpose capability each alternative might achieve for the system, but by how much the design will contribute to the major function for which the system is being implemented.

In contrast, then, to the general purpose and the functional point of view, there is the perspective which stresses the development of the man/machine interface as being of primary, but by no means exclusive, importance in system design. This category includes SKETCHPAD, BASEBALL, DEACON, JOSS, and Culler-Fried. Each of these systems has introduced a new dimension along which man and machine may interact. With SKETCHPAD the dimension is graphical; with JOSS, the dimension is easy mathematical computation for scientific and engineering users; with Culler-Fried, the dimension is push-button representation of operator/operand functions characteristic of mathematical problem formulation. Not all of these systems have been constructed with a specific job function in mind. But each has emphasized the necessity for making the man/machine interface as flexible and unconstrained as possible in the performance of some job, real or potential.

In contrast to the three aforementioned approaches, there exists yet another one to computer-based system design. From this point of view emphasis is placed on the development of a particular problem-oriented language which is tailor-made to some specific problem area; that is, the development of a language which is easy for the computer to manipulate and which is simultaneously much like the specialized language normally used by the potential user. The point of view represented by the problem-oriented language designer is quite similar to the man-machine interface designer but with one important exception. The latter wants to provide a tool which can be used over a large class of problem areas. Thus, a graphical technique can be used not only in solving engineering problems, but also in solving artistic, mechanical drawing, mathematical and geometric problems as well. On the other hand, the special purpose problem-oriented language is one which can be used in only one specific problem area. It cannot be generalized and for that reason,

it can pay a great deal of attention to providing a specific problem-oriented language which maximizes convenience for some practitioner in a specific area. STRESS, a language for expressing structural engineering problems and COGO, a language for civil engineering problems, are two examples of this approach to system design.

A fifth area approaches the problem of system design from a point of view which is quite different from all of the above. This point of view, regardless of whether the computer has been endowed with general purpose systems, functional systems, or special purpose systems, provides the computer with the ability to be accessed simultaneously by a large number of separate and distinct users. This perspective on computer-based system design has been labelled the "time-sharing" perspective. Its proponents view the time-shared computer as a kind of public utility from which any user can draw whatever computational power he might need at any time without delay or difficulty. In one sense this perspective differs quite radically from the other four because it is not concerned primarily with the various ways in which computational power can be increased or tailored to a specific user.

It was previously mentioned that the sixth category, generalized data management systems, could best be understood in the light of the preceding five because this category views the design of computer-based systems as a problem in the speedy implementation in a given system application of as many of the features of the preceding categories as are relevant and, indeed, possible. Thus, in this category, the designer is concerned with emphasizing general purpose programming as well as job specific programming, and user-on-line techniques as well as special purpose languages, and, if possible, making the resultant design time-sharing. Two systems represent an attempt to exploit all of these categories, viz., Advanced Data Management (ADAM) and Language Used to Communicate Information System Design (LUCID).

Nothing has yet been said about the motivation behind each of the perspectives cited above. There are, of course, several motivations, ranging from personal biases, to the special circumstances of a specific environment, to the exigencies of time, money, and personnel, to the special talents of a specific designer or user. With respect to the category of generalized data management systems, for example, one quite clear motivation has been economics, since the use of these systems as test vehicles for complex computer application obviates the need for constructing expensive prototype models. This and many other reasons no doubt inspired the various points of

view exemplified in the following discussions. Regardless of motivation, however, these points of view stand as alternatives to the problem of the economic and effective use of digital computers. Moreover, the specific systems discussed attest to the success of each of these points of view and strongly suggest that the current state-of-the-art in software is sufficiently advanced to countenance further development in the exploitation of many, if not all, of these techniques into effective operational systems and sub-systems.

NOTE

The compilers of this survey have taken the liberty of quoting extensively from the published material referenced in the Bibliography. Where accurate descriptions of the systems are given, credit is due the designers themselves and their writing staffs. Where errors occur, the compilers of this volume take full responsibility.

In the following six sections, the order in which the systems are presented was established on heuristic grounds, and no evaluational judgment is intended.

PART I. GENERAL PURPOSE PROGRAMMING AND EXECUTIVE SYSTEMS

In this section those systems which have been chosen to emphasize generalized programming techniques are reviewed. The fundamental strategy behind this approach to computer application takes cognizance of the fact that users of many computer systems are not only people, but also other programs. It, therefore, addresses itself to those inter-facing problems which occur between programs, both system and user. Hence, generalized programming techniques attempt to provide linkages between system and user programs, without reference to specific systems or users. Similarly, generalized programming techniques attempt to anticipate the numerous housekeeping functions required by complex program configurations, regardless of the content of particular programs. Thus, the core of generalized programming techniques is the realization that computers can be endowed with greater power, flexibility, and generality by providing them with programs capable of manipulating other programs in the same way that more conventional programs merely manipulate data. The six systems discussed in this section exemplify this strategy.

A. CL-II PROGRAMMING SYSTEM

The CL-II Programming System is an integrated collection of computer programs which provides assistance to the user of a large-scale digital computer in preparing and executing programs. The system has been developed by Computer Associates, Inc. for the IBM-7090 (94) computer in response to a need for:

(1) A comprehensive programming environment within which a large group of programmers can efficiently attack and solve problems involving large programming efforts.

(2) Efficient utilization of computer hardware.

(3) Minimization of the cost of modification and extensions of extant programs.

(4) A system which evolves in accordance with the needs of particular users.

Previous attempts at automatically linking and operating programs for the convenience of the programmer had been limited to compile and load time linkages and to programming some of the computer operator's functions into the monitor system. This necessitated the programmer's considering his program subject to the framework of the basic computer hardware. Under the circumstances, each programmer must repeatedly be concerned with various difficulties of assembling and operating programs when these difficulties have little relation to the problems his programs are devised to solve. Further, the operating or monitor system can do little more toward improving overall machine efficiency than reducing job set-up time. The CL-II Programming System was designed to provide a more convenient and natural framework than the basic computer for the construction and execution of programs. The design of the CL-II System was guided by the following considerations:

(1) The system will provide mechanisms for linking programs to data, hardware components, or other programs. These mechanisms will be such that they can be employed at any time in the process of defining, loading, or running a program.

(2) The system will have means for controlling the simultaneous execution of a number of programs.

(3) The system will have no built-in assumptions about job scheduling. Thus, there will be no fixed "job monitor" or "executive"; rather, any program will be able to perform monitoring functions.

(4) The system will have means for creating, maintaining, and gaining access to "files" of programs, data, and descriptions of data.

(5) The system will have means for generating the code for accessing elements of data sets or programs from the information contained in a description of that data set or program.

In the light of these considerations, the CL-II System was endowed with the following characteristics:

(1) Once defined to the system, a program is essentially an extension of the system itself in the sense that it can generally be called by any other program in the system at any time. This is true not only of "user" programs, but of "system" programs as well.

(2) A change in the structure of a data set or program requires at most a recompilation of programs referencing that data set or program in order that all accesses be correct; i.e., no changes are required in source programs to reflect changes in the structure of data or programs they use.

(3) The addition or deletion of types of input/output devices or the addition or deletion of a number of units of a given type need have no effect on many user programs.

(4) The "kinds" of problems amenable to the system include: sequences of small jobs (the kind of problem acceptable to conventional operating systems); large production programs involving considerable segmentation, program and data overlay, and the like (such as large scale simulation models); "command and control" type programs; and "time-shared" programs.

The CL-II System went through three stages of construction: Primary CL-II, Basic CL-II, and the CL-II System or, simply, CL-II. Roughly, Primary CL-II consists of the basic programmatic extensions to the computer hardware which form an "extended machine" with which the programmer deals. Basic CL-II is Primary CL-II to which has been appended a sufficient number of "system programs" (compilers, assemblers, dumps, monitors, and the like) to make the system easy to use for implementing user programs. The CL-II System is then Basic CL-II plus the collection of user programs available at any time.

Primary CL-II performs certain basic functions. At any time during the operation of CL-II, there are one or more programs in high speed memory which may be operating in parallel. With each program is associated a priority number. Each program has one or more <u>paths</u> which may be operating (i.e., the computer is currently sequencing through the instructions comprising that path), is <u>ready</u> to operate, or is <u>waiting</u>. There are available to the system one or more <u>files</u> from which programs and data can be obtained by name.

Whenever there is an available control element of the computer which can be assigned to a program, it is put to work on the first ready path of the highest priority program which contains any ready paths. Thus, the scheduling which is built into CL-II is extremely simple. It should be noted that the scheduling of programs does not derive from the system itself but rather from priorities determined through the interaction of the user programs with each other and with the system.

The collection of "instructions" which can comprise the programs operating within CL-II includes all the instructions, plus a collection of "system macros" provided by Primary CL-II These system macros include the following:

(1) Mount a tape.

(2) Demount a tape.

(3) Assign a peripheral device.

(4) Read, write, or position a peripheral device.

(5) Allocate a block of high speed memory.

(6) Input or output data.

(7) Start a path.

(8) End a path.

(9) Establish a program or data set.

(10) Execute a program in line or in parallel.

(11) Await an input, output, path completion, establish operation, tape mounting, etc.

(12) Release a program or data set.

(13) Start a program.

While certain of these macros are related to functions provided by conventional utility systems, the majority is not. The "establish" macro is the

heart of CL-II. Basically, the establish macro procures a program from the file, allocates sufficient space for it, brings it into high speed memory, and performs all manipulations (e.g., fixing absolute addresses, etc.) necessary to prepare the program for execution in the memory locations to which it is allocated.

When a program has been established, it can be called by the program which established it. This call can cause the program to be executed as an in-line subroutine, or as a completely independent program (in parallel with the program calling).

Any program which has requested that the system perform some activity such as input, output, establish a program, execute a program, start a path, mount a tape, etc., can at some later time "await" the completion of the activity. If the activity indicated is not complete, then the path containing the "await" is placed in waiting status and it is arranged so the completion of the activity indicated will automatically place that path in ready status.

In order for the user to effectively use CL-II, there is a collection of "system" programs which are always available to Primary CL-II on a "system file." Primary CL-II, to which is attached this system file, is called Basic CL-II. The system programs in Basic CL-II include the following:

(1) Sequential Job Monitor

(2) Data Description Translator

(3) Compiler for (extensions of) Language L and JOVIAL

(4) Compiler Generator System

(5) Data Input/Output Processor Formatted Data

(6) Miscellaneous Service Programs

A very brief description of these system programs follows:

1. <u>Sequential Job Monitor</u>

The job monitor initially available in CL-II is a reasonably standard sequential job monitor. One can, in a straight-forward way, call for the input and

filing of data descriptions, input and filing of instances of data sets, compilation and filing of programs, output of instances of data sets, and construction and execution of collections of programs.

2. Data Description Translator

A data description in CL-II is provided by an instance of the data set ODES (Object DEScriptor). The object descriptor for a specific data set contains a complete description of all of the elements of the data set, including names, types, and mapping functions for all the various representations of the data set which may occur (internal memory, punched cards, magnetic tape, and so on). In addition, it includes the specification of the composition of (collections of) data elements into large elements, resulting finally in the complete data set.

3. Compiler

The compiler in the CL-II System is general purpose, table-driven, and has the ability to generate efficient code. By supplying appropriate tables, control is obtained over (1) the language to be compiled, (2) the amount of context searching preliminary to code generation, and (3) the specific machine code to be generated corresponding to any input or sub-statement. Thus, modification of the input language or accommodation of a completely new language is handled by modification of tables rather than by rebuilding all or part of the compiler. Similarly, "tuning" the generated code or allowing for "quick-and-dirty" code to be generated under certain circumstances is controlled through modifications of the tables.

Initially, the languages which are accommodated in CL-II are L_0, the algebraic language available in the CL-I System, and JOVIAL. Actually, the versions of L_0 and JOVIAL are somewhat extended versions in the sense that they are able to reference elements of any type of data structures for which ODES descriptions are available in the files.

4. Compiler Generator System

The compiler generator system is a collection of programs (several of which are also a part of the compiler itself) and data sets which can produce the tables required to "drive" the compiler from three sets of specifications: (1) the syntactic structure of a programming language to be translated by the compiler; (2) the "generation strategy" and context inspection of syntactically

analyzed statements in the programming language; and (3) the rules for the selection of specific fragments of machine code corresponding to statements or substatements in the programming language being translated.

5. Data Input/Output

Given the description of a data set (the ODES for that data set), the data input processor in CL-II will input, edit, and convert an instance of the data set resulting in an internal copy which is in correct format for filing the data set and/or "handling" it to a user program for immediate processing. The formats in which data can be represented on external media such as cards are very general. For integers floating-point numbers, and scaled fixed-point numbers these formats include the designation of fixed or free field, inclusion of signs, decimal points, exponents, and so on. The data input processor also has means for checking the adequacy of the data being input as to format, size constraints, duplication, missing values, etc.

Similarly, given an ODES and an instance of the data set described by the ODES the data output processor can output the data set in appropriate formats. Special headings can also be supplied.

6. Miscellaneous Service Programs

There is a collection of service programs available to help the programmer in debugging his programs in CL-II. These include:

(1) Dump programs, which can be called to produce octal and/or symbolic dumps of specific programs or data sets in memory, optionally including a "road map" of all programs and data sets which are currently occupying memory.

(2) Octal correction programs, which load corrections into programs prior to their being run.

(3) Symbolic correction programs, which load corrections represented symbolically into programs prior to their being executed and/or into their file copies.

As previously mentioned, the CL-II System is the union of Basic CL-II and the collection of user programs developed within the system. Thus, CL-II

is fundamentally intended to evolve with the construction of user programs and will undoubtedly take on different characteristics with different collections of users.

The CL-II System contributes to generalized data management systems in that it provides a general-purpose programming environment in which the programmer is freed from concern with many aspects of programming not connected with his problem.

It, of course, is not fundamentally problem-oriented in the user sense, but facilitates the development of application solutions within its framework.

B. AUTOMATIC OPERATING AND SCHEDULING PROGRAM (AOSP)

The Automatic Operating and Scheduling Program (AOSP) for the Burroughs D-825 Modular Data Processing System is comparable to that portion of the CL-II System known as Primary CL-II. Hence the AOSP does not purport to be a complete programming system which includes compilers, monitors, etc. Instead, it should be viewed as a general operating system, designed for the D-825 with the aim of facilitating the use of hardware in a multi-computer system. In fact, like Primary CL-II, the AOSP should be viewed as an extension of the hardware itself. Specifically, the AOSP provides the control for multi-computer, multi-processing in a multi-computer system with shared high speed memory without requiring a master/slave relationship among the computers. The AOSP itself may be executed by more than one computer simultaneously.

Although the D-825 has numerous interesting system design features, many of which do part of the job of an executive system, the most interesting properties from a programming point of view are:

(1) Several (up to 4) independent computer modules may be in operation simultaneously, with the entire memory accessible to each computer;

(2) When instructions in memory are being executed by a computer module, all addressing is relative to base address registers. These registers are in the computer module (that is, not in memory).

The hardware permits any program to be executed from any location in memory, referencing any other locations in memory, with no change of any part of the instructions of the program--only some changes to the data words

of the program need be made, and by AOSP convention, all of these data words are grouped into a contiguous block at a standard (relative) location within the program.

Hence, there is no need to have more than one copy of the instruction part of a program in memory, even if two or more entirely independent jobs call upon that program--only separate copies of the data part need be made, and the same instruction may be executed simultaneously by two or more separate computers.

Insofar as the AOSP imposes any restrictions on the external or internal forms of programs or data sets, the restrictions are not onerous, and are easily met by a human programmer writing in assembly language, or by a compiler. Even those operations which must be performed by the AOSP (rather than directly by the program) can be specified in as much detail as the programmer chooses. In return for the mild restrictions, the programmer is permitted greater freedom.

No absolute addresses (or unit numbers, etc.) are required. The hardware is designed so that there are no preferred addresses in the machine, and this feature is carried up to the Extended Machine. All references to independent programs and external data sets are by arbitrary name. All memory allocation is handled by the system.

Facility-sharing, multi-processing, and parallel-processing occur as a matter of course. As many separate jobs as will fit in the memory of the machine may be in process at once, and as many will be being actively executed as there are computer modules functioning in the system.

Indeed, a single job may split itself into several parallel branches for simultaneous execution.

Response to external signals for initiating new jobs is handled by the system without disturbing those already running; thus (except for manual manipulation of tape reels and card decks) operator intervention in the system is restricted to the exchange of messages on the console typewriter, which occurs in parallel with the running programs. The machine never stops, short of catastrophic error conditions.

Many of the necessary functions of the system are encoded in routines whose form differs in no way from a regular user's program, and are therefore

subject to modification without disturbing the "Basic Machine" portion of the system.

The AOSP reserves to itself the execution of all control mode machine instructions (halt, I/O transmission, response to interrupts, etc.). The users' programs, therefore, have available all of the machine's normal mode instructions, plus a set of AOSP Macros. These are short instruction sequences which have the effect of requesting the AOSP to perform some function on behalf of the calling program. These functions are essentially the same as those listed for Primary CL-II. In addition, although they are not considered parts of the AOSP itself, there do exist file-maintenance routines, assemblers, compilers, and dumps which run under AOSP control.

As in Primary CL-II, the AOSP uses only a simple job sequencing rule. Priorities can be assigned by an AOSP Macro, and any time one of the computers in the system is looking for work to do, it commences (or resumes) execution of the highest-priority task which is in ready-to-run status. When it is needed, a more sophisticated monitoring algorithm can be written.

The only other important difference between the AOSP and Primary CL-II lies in the area of simultaneous execution of parts of a program. Both systems allow, as a matter of course, the simultaneous execution of independent programs, and both allow a single program to initiate a new parallel path of control. However, within CL-II, this parallel path is a separate named entity, possessing properties which can be tested by functions of the system. The AOSP, on the other hand, was designed to be experimental in this regard, and no mechanism for automatic coordination of parallel parts of the same job was provided in the initial version. Thus, the programmer must explicitly program the synchronization, and communication and protection between the parallel paths of his program structure. Basic facilities for doing this are, of course, available in the AOSP.

The functions of the AOSP may be considered in three categories:

(1) Responding to Interrupt Signals - The AOSP has provisions for recognizing and dealing with all of the numerous interrupt conditions provided by the hardware; when any such condition occurs, the computer module designated to handle it temporarily suspends operation on its current program.

(2) Responding to Explicit Requests from the Programmer - A few of these will be specifically mentioned below. An adequate set of allowable

requests was included in the first AOSP, and others have been added. A very important operation in this category, for illustration, is the initiation of an input-output operation. The completion of the I/O transmission is recognized by a computer interrupt, and handled automatically (Category 1).

(3) Monitoring of Memory Organization and Program Control - At any instant, the system memory is organized into numerous areas, which are of different kinds, and have different relationships among them.

An area of memory is either available space, or is (with certain systematic exceptions) uniquely identified by its name and file name. The AOSP has at its disposal a directory of the files which currently exist in the system; a file is a collection of objects (generally records) in an external storage device or in core memory. The objects are programs or blocks of data; an object can be readied in memory by simple request to the AOSP, giving the object's name and the name of the file on which it resides. In addition, any object may contain the names and file names of other objects which are necessary to it. These will be readied automatically when the object itself is readied. When the object is a program, this list of necessary objects is transformed by the AOSP into a block of allocation addresses, which the program may use to reference the objects after they have been readied.

Monitoring of the flow of control is handled through a job table. Each entry of the job table contains all the information which must be loaded into a computer module's private registers in order for that computer to begin (or resume) execution of a program; the system is quite similar to the state word technique described in "A Multiprocessing System Design," M. E. Conway, Vol. 24, AFIPS Conference Proceedings. Each entry in the job table represents an independent path of control, and there may be an arbitrary number of them. Whenever a computer is released from another task, it proceeds to execute the first available job in the table. There is a direct request operation which allows a program to initiate a new job-table entry, i.e., a parallel path of computation within the same program.

All run-time operator communication with the AOSP is via the console Flexowriter, which can generate a peremptory interrupt. The AOSP constantly maintains in a ready condition an instance of a Responder program which interprets messages coming in from the Flexowriter. The most frequent such messages merely give the name and file name of a program which is to be executed. The Responder causes the entire structure of objects rooted at the named program to be readied in memory. A job-table entry for this path of

control is established and program execution commences. There are a few special tasks the Responder can be asked to perform directly, (such as terminating a particular job), and there is an implicit system file containing the basic file-handling programs, which can be called without file names.

The AOSP is successful in its primary purpose of making available to the programmer the facilities inherent in the hardware. The cost of this effectiveness would be extremely hard to define, let alone to measure. The AOSP itself occupies relatively few words of memory, even with the job table, I/O processing, etc. If necessary this could be reduced further at some cost in speed, by keeping parts of the AOSP on secondary storage. There have been no attempts to measure the percentage of time spent in the AOSP during the execution of programs; but since it would be effectively impossible to perform most of the AOSP's functions without some sort of equivalent program, the cost is hard to evaluate.

The AOSP does require that programs conform to certain conventions of organization. But these restrictions do not appear unnatural. It is not difficult to cause a compiler to produce code in conformity with the required structure and subjectively, new programmers working in symbolic machine language find it harder to learn the machine operations than to learn the program organization. Also, a clever programmer finds it possible, if he wishes, to violate the spirit of the restrictions without appearing to violate the letter, if he really needs to do something for which the AOSP does not provide.

The facilities for parallel processing were deliberately kept at a basic level. It is possible to initiate a new parallel path of control. There are facilities for awaiting the occurrence of a bit condition, by means of which one path can report to another, and there are facilities for temporarily denying access by all other paths to a particular block of data. The intention was to let programmers play with these basic mechanisms in order to discover what problems will arise, before building in a more complicated structure of automatic protection and reporting between paths. The system is admittedly not designed for efficient parallelism in the small, e.g., for simultaneously evaluating two terms of a sum, since it takes a couple of milliseconds to initiate a path; rather, it is designed for parallel operation of relatively large segments of a program structure.

Some comments on the machine configuration are useful. The AOSP does not require an exact machine configuration, though it was designed to operate

on a general D825 Data Processor. Equipment availability, processor, memory, I/O modules and peripheral devices are designated by tables. In general, equipment may be freely moved in or out of the system. If a new type or special purpose device is added to the system, proper program controls would have to be incorporated into the operating system. This was realized in the initial implementation and has been accomplished with relative ease for various applications.

With respect to core allocation, there are a number of versions of the AOSP ranging from 5000 to 6800 words of resident core. The range depends on the various peripheral devices on a given system. However, in the D825, a three address, variable instruction machine, the 6800 words of core represents approximately 11K instructions.

The initial implementation of AOSP required ten man years. Since then, an additional ten to fifteen man years have been spent on AOSP. The length of time required to design the initial implementation of AOSP was six months, to code it eight months and for checkout four months.

C. INTEGRATED INFORMATION PROCESSING SYSTEM (INTIPS)

Project INTIPS is essentially an exploratory development program being conducted by the Rome Air Development Center, New York. The objective of the program is the development of a multi-programming user-oriented computer system. The main element of the INTIPS system is an executive routine which is designed to integrate a control computer, numerous dissimilar operational computers, a hierarchy of storage devices, a common pool of peripheral equipments, and numerous consoles or query stations.

A program called the Executive Control Program has been written to coordinate the operation of the entire computer complex. Under this program's control, it is expected that any of the computers may utilize any of the peripheral equipments, and, more importantly, any portion of the entire equipment complex can be brought to participate in the solution of the problems of users operating from their individual consoles. The Executive Control Program (ECP 1A) for Phase I implementation became operational in September 1964. It is functionally modularized to facilitate expansion and revisions to accommodate the addition of new features and characteristics, and to facilitate experimentation with alternative approaches. Among the functionally independent modules in the ECP are an input-output controller, priority and interrupt handler, dynamic storage allocator, program and

equipment scheduler, and a coordinator.

The Executive Control Program will be executed from the CDC-160A Control computer. The language currently available for use in program preparation is Executive Control Program Interpretive Language (ECPIL). This language is intentionally very similar to the CDC-160A basic instruction set and is employed by the operational programmer in preparing programs to be executed by the Executive Control Program. More sophisticated languages are being anticipated for later INTIPS implementation.

In its current state of development the Executive Control Program is expected to handle up to five independent programs at any one time on any given computer in the complex. Additional jobs will be stored in a job queue.

Part of the equipment configuration for INTIPS includes an Interim Exchange (IX-400). This will be replaced by a Central Exchange (CX-400) in a later version.

The Interim Exchange is a 3-by-15 electronic switch capable of accommodating up to three computing or "controlling" modules and up to 15 non-computing or "controlled" modules. Any of the "controlling" modules can request a connection to any of the "controlled" modules with a special instruction recognized and executed by the Executive Control Program.

The Central Exchange which will replace the Interim Exchange is a 16-by-61 electronic switch capable of accommodating up to 16 "controlling" modules and up to 61 "controlled" modules. It contains its own internal memory (1024 words) and can execute seven special instructions, five of which are given directly to the Central Exchange and may be given by any controlling module; the remaining two instructions are given to the Master Coordinate of the CX and may be given only by the Control Computer.

The Control Computer makes the assignments of controlled modules to the controlling modules via the Symbolic File.

The Symbolic File resides in the 1024 word memory of the Central Exchange and allows the Control Computer to store assignments for symbolic addressing of controlled modules and of alerts and to store a history of the last request and action taken at each controlling module.

With respect to storage allocation, one of the major characteristics of the system is that all information, data as well as programs, is stored in segments/overlays of 512 words. A program may occupy from one to seven contiguous segments of (core) memory. Programs may be of infinite length, but no more than the number of segments/overlays specified in the Job Control Record (JCR) will reside in memory at any given time. The term "Segment" refers to a 512-word block of information contained in the control computer memory, whereas the term "overlay" refers to a 512-word information block in auxiliary storage. That is to say, segments are numbered from one through seven with reference to their physical positions in memory whereas overlays in auxiliary storage are numbered according to their position in a program or data file. All input/output transfers, with the exception of "Print" and "Display" are of an entire 512-word segment/overlay. When ECP 1A loads a job program for execution, it places only the first overlay into memory. Each successive overlay must then call in others as required through appropriate instructions which have been provided automatically by an assembly program. All addresses are assigned relative to "zero" in operational and all overlays, data or program, are completely relocatable under ECP 1A control.

The major contribution of the initial INTIPS, when completed, will be in the area of executive control. The nature of that contribution will be in elucidating the kinds of techniques required of an executive routine whose major function is the efficient integration of an heterogeneous configuration of hardware.

D. DEPARTMENT OF DEFENSE DAMAGE ASSESSMENT CENTER (DODDAC)

The current Department of Defense Damage Assessment Center (DODDAC) activity includes data reduction, data base creation, and a number of complicated damage assessment models, including Rapid Damage Assessment and Hazard and Vulnerability models. System Development Corp., Santa Monica, Calif., has developed a software system for controlling these tasks known as the Executive Control System for a CDC 1604-160 computer complex which includes tapes, discs, and CRT displays. The Executive Control System provides the linkage between the various system elements--programs, hardware, and operators. It is the internal mechanism through which users and operators exert control and interact with the DODDAC program system.

The Executive Control System performs the following specific functions: system start-up; mode/model control; equipment monitoring; interrupt

processing; operator-program communication; control of input/output data flow; 1604-satellite intercommunication; initial request interpretation; priority and queuing; task control; and program and environment sequence control.

The system is comprised of four distinct programs:

(1) Executive Control Program (ECS)

(2) Executive Start-Up Program (ESU)

(3) Mode/Model Control Program (ECC)

(4) Non-operational Request Processing Program (ECN)

ECS, which refers to a specific program, not the entire Executive Control System, consists of several components comprising the permanent core portion of the Executive Control System. The individual components of ECS perform the functions of master control, interrupt processing, on-line card processing, request interpretation, priority and queuing, task control, program and environment control, data collection and input/output. The last function is performed by the Input/Output Monitor, a group of subroutines assembled as part of ECS.

ECS directly controls the operation of the other Executive programs and the operational models, Rapid Damage Assessment (RDA) and Hazard and Vulnerability (H & V). It controls the sequencing of programs and environment for RDA and H & V by interpreting tables of preset sequence parameters. The non-operational subsystems, Utility, Data Base Load, and Data Reduction, have separate control programs which guide internal program and environment sequencing; however, they initially obtain control from the Executive Control System. Because the non-operational subsystems destroy the permanent core Executive, they return control to ECS by reading in and branching to the Start-up Program (ESU).

The non-permanent core Executive programs, ESU, ECC, and ECN, exist in core only when their specific services are needed. ESU, which is read in by a paper tape bootstrap, prepares for the operation of the DODDAC system by communicating with the computer operator to obtain time and equipment information and by establishing the initial system environment. ESU completes its responsibilities by reading in and transferring control to ECS.

ECS reads in and transfers control to ECN when it is necessary to interpret or process a non-operational request. If possible, ECN implements the request itself and returns control to ECS. More frequently, ECN reads in and transfers control to a non-operational program or subsystem which performs the required functions and returns control to ECS. Because of the extensive capability of the ECS to provide linkages between very large and very complicated system and user programs, as well as between system users and system hardware, the ECS qualifies as an example of the general purpose programming approach to computer system design. This is not to say, however, that DODDAC is completely characterized by its ECS. Rather the system includes an extensive set of data management functions as well. Among these functions are: the Data Base Load Subsystem, miscellaneous support programs, and the OASIS subsystem. A description of this latter subsystem follows.

E. OASIS SUPPORT SYSTEM FOR DODDAC

The OASIS support system was built to provide an environment for the development, check-out, and operation of DODDAC operational programs. OASIS was superimposed on the conventional CDC software; this software being, at times, degraded in the process.

OASIS includes a JOVIAL compiler and an assembly program which can operate from a COMPOOL of data. The system controlled the operation of programs which had been previously stored on a Master Tape. All space allocation problems are handled by the programmer explicitly. Only absolute binary programs are stored on the Master Tape. In order to operate the system, the programmer must communicate control information to the system via the card reader for every phase of the processing. For example, four phases are required to compile and add a JOVIAL program to the OASIS Marker Tape. A sample control card deck to perform this follows:

CLEAR:	Command to a utility program to clear core.
LOAD:	Command to perform a card-to-tape operation on the JOVIAL source deck, moving it from the card reader to a specified tape.
JOVIAL:	Command to perform the compilation, the output of which is standard input to the CDC assembler, CODAP.

WAIT:	Command to the operator to switch tapes; system awaits completion.
UPP:	Command to invoke a "preprocessor" which modifies the standard CODAP input for COMPOOL data elements.
UAA:	Command to perform an assembly (with a modified CODAP routine) the output of which is an absolute binary routine.
UMT:	Command to a utility routine to add the binary routine to a Master Tape.

Separate programs are combined into larger operating units via a system routine that requires that the components be listed and the order of execution specified. The programmer supplies the system with the disc address or tape record address of each component. All programs have been previously bound to absolute memory locations and the programmer is responsible for memory allocation.

F. NECPA MASTER CONTROL SYSTEM

The NECPA (National Emergency Command Post Afloat) Master Control System (MCS) is an executive system which performs the following processes:

(1) Controls and schedules NECPA Data Processing System to work on tasks of highest user-defined priority.

The MCS has a table of tasks in waiting ordered by priority. This table has an associative list structure. The MCS has the capability of changing priority of any task in the waiting task table and can also be called upon to cancel or delete any entry in this table. The MCS has the ability to print out upon request, any or all of its tables.

(2) Controls I/O operations to allow a set of 'psuedo-offline' operations to process independently of, but concurrent with, the program being executed.

The MCS controls all the I/O for the system in an attempt to optimize I/O throughput by:

(a) Maintenance and interrogation of an I/O configuration matrix.

(b) Control of assignment of physical tape channels and units.

(c) Implementation of tasks by parallel rather than sequential I/O data transfer.

(3) Maximizes system throughput.

The MCS allocates channels and units to give the optimum overlap for parallel reading and writing of data.

(4) Reallocates vital system functions in the event of certain types of system equipment degradation.

In the event of degradation of the system I/O configuration, the MCS reallocates I/O equipment to perform the essential tasks and also attempts to optimize the I/O within the degrading configuration. The MCS can interrupt any task currently being processed and save the results for restoration at a later time.

(5) Maintains list of jobs in job queue.

(6) Maintains tape log. The MCS maintains a log of all data tapes required by functional programs. Some of the information contained in this log are (a) identification of each file, or (b) date of last update, (c) number of physical reels comprising a file, and (d) the security classification of each file.

(7) Checks security classification of job and tapes required for jobs.

(8) Informs operator of all significant system events. The MCS keeps a set of system statistics. Some of the functions carried out by this set of programs are:

(a) Informs and instructs operator about all input data errors or ambiguities.

(b) Informs operator whenever task is begun, completed, interrupted, resumed, or cancelled.

(c) Upon completion of task MCS reports:

Description of task
Number of interruptions
Completed tapes saved

(d) Upon completion of system shift MCS reports:

Completed tasks
Cancelled tasks
Deleted tasks
Uncompleted tasks with restart information

(9) Checks and positions all tapes mounted for jobs. The MCS checks that the most recent version of a file is being used unless specifically overridden by the task request or by the operator.

PART II. FUNCTIONAL SYSTEMS

In this section those systems which have been designed to perform a large and complicated, but nonetheless well delineated job function, are discussed. (The systems which follow exemplify the job function of information retrieval. There are, of course, quite different systems, such as Lincoln Laboratory s GENDARE which is designed specifically for data reduction, but they have been omitted to simplify the discussion.) The necessity for providing the user with a functional capability has overridden the desire to provide powerful programming capability. Thus the systems listed here have been designed to perform a specific function and the ultimate design criterion was not one of system generalization, programming sophistication, system evolution or man-machine flexibility. Rather the ultimate criterion for design decisions was whether or not a particular alternative would contribute to the performance of the system in that function which it would exercise more than any other. This is not to say that the systems which follow are lacking in flexibility, evolutionary growth, or generalization. In fact, many of them show striking advances in each of these properties. Rather, what is characteristic of the approach embodied in these systems is that whenever an either/or decision had to be made with respect to improving the information retrieval capability, i. e., the functional capability of a given system at the expense of evolution, generalization, flexibility, etc., the decision was always made to enhance the function at the expense of the general capability.

A. COMPILE ON-LINE AND GO (COLINGO)

COLINGO (Compile On-LINe and GO) is a general purpose storage and retrieval system developed for the USSTRICOM Interim Command and Control System by the MITRE Corporation, Bedford, Massachusetts.

COLINGO provides a set of modular computer programs for file generation, file maintenance, data retrieval, and control on an IBM 1401 computer.

Mathematical and logical operations using the system's data base are provided by both stored program and on-line programming techniques

through the use of the COLINGO Control Language. Control of the data, computer programs, and equipment is achieved through on-line interpretive execution of statements in this language entered by cards or from a single console typewriter. The design includes a Basic Program Set which provides programs for rapid and convenient data file update, data file addition, and stored program addition. Programs are constructed as closed subroutines which may be written in Autocoder, SPS or COBOL. This design allows expansion of the Basic Program Set and the addition of special purpose and convenience features. On-line file generation and on-line programming in COLINGO are an integral part of the system design. Special data maintenance and verification routines, as well as automated system dictionary and vocabulary preparations, are also included in the system. These features will permit evolutionary growth of the system and provide the capability to produce special purpose operational programs with a minimum of programming time.

The data acceptable to the COLINGO system can vary widely in type, but will, in general, be structured in files according to a uniform set of rules. Consequently, the procedures and programs used to retrieve and process the data can be made applicable to any data by designing the programs to process formats and not the data. Moreover, new files can be introduced into the system and files in the computer can be changed or deleted with no change to the retrieval programs.

Data files consist of serial records with provisions for unlimited trailer records. Any number, or order, or size of unique fixed-field data categories is permitted. Files are organized to three levels: file, property and sub-property; and are described by the COBOL data division format. The COBOL data division is preprocessed by a dictionary and index generation program to organize the data description into a fixed-format dictionary for use by the COLINGO program. All properties and subproperties may be retrieved.

In addition to the tape files COLINGO has 26 QUIC files on disk. These files are packed and can be generated in a very flexible manner. Retrieval from the QUIC files is much faster than retrieval from the tape files. Normally several of these files are used for stored queries, lists of files in the system, report masks, etc.

Dictionaries and vocabularies are maintained for the entire data base. File generation, updating, regeneration and deletion all cause automatic

updating of the dictionary indices and vocabularies. Files generated outside of the COLINGO system require preprocessing.

The basic retrieval function in COLINGO is specified by the Query Statement which consists of three parts: the File Statement (which names the file to be interrogated); the Criteria Statement (which specifies the selection criteria); and the Output Statement (which describes the desired output information, the output device(s), and any desired manipulation of the retrieval data).

Example of Query Statement:

(File Statement)	GET AIRFIELD-FILE
(Criteria Statement)	IF RUNWAY/LENGTH EQ 5000
(Output Statement)	PRINT RADAR/TYPE

The parenthesized words in the example are for illustration and do not appear in the Query Statement. The above example would print out the type of radar found at each airfield in the AIRFIELD file that had a runway 5000 feet in length.

Apart from the basic Query Statement, the Control Language has certain special features some of which are mentioned below:

- Misspellings may be corrected by the operator at the typewriter before the statement is entered.
- Column headings, titles, security classifications, date and time, page number, etc., may be specified (some come automatically).
- Query input and hard copy output can be assigned by the operator to a variety of devices.
- The operator can interrupt any query and suspend or discontinue it.
- Provision is made to store any query statement which may then be invoked by name.

- Comments to the operator may be imbedded in stored queries.

- On-line changing and/or sorting on selected properties is permitted.

- Arithmetic expressions involving constants and retrieval data fields may be specified by the user and interpretively evaluated.

- Direct provisions are made in the Control Language for dictionary and file addition, insertion, replacement and removal.

- Queries may call one or more other queries.

- The IF/NOT function permits conditional branching within a query run (ex. IF RUNWAY/LENGTH LQ 5000 EXECUTE SHORT IF/NOT EXECUTE LONG where SHORT and LONG are other queries.)

- Variables may be inserted in previously stored queries on-line.

- The COMPUTE verb will cause the evaluation of FORTRAN-like arithmetic expressions.

- Qualifying records may be counted and property or sub-property values may be summed.

- Data output may be by one of the standard output formats provided or through a report mask specified by the user.

Several error messages appear automatically. They essentially say "A parameter is missing," "What you want does not exist" or "I don't understand you."

The principal subroutines of the COLINGO system include: a disk handling routine with bootstrap; tape handler with bootstrap; four dictionaries with their own search routines; a dictionary updater; a disc allocator; various output routines and routines for decoding and executing query statements.

The essential characteristic of COLINGO is its ability to map any data base format into its own format in a very short time and then to query the resultant formatted data with facility. As a consequence, COLINGO is a highly flexible

information retrieval system. It is this characteristic, rather than the nature of its query language or the structure of its executive routine, that was of primary importance to the system designers. However, the ability to chain queries and store them for subsequent use permits complex procedures involving extracted data to be constructed on-line. Thus, COLINGO also has some desirable features of a user-on-line system. On the other hand, COLINGO has certain shortcomings which, from the point of view of efficiency, make it undesirable as a universal tool in other areas where flexibility in data retrieval may be considered unnecessary.

B. NAVCOSSACT INFORMATION PROCESSING SYSTEMS

The NAVCOSSACT Information Processing Systems have evolved through a developmental process. The first system was developed for the David Taylor Model Basin in 1961. This system, for the IBM 704, was further developed into the current system operating on the CDC 1604 at CINCPAC, the Univac USQ-20 for the National Emergency Command Post Afloat, and the 7090 at the Navy Information Center in the Pentagon. All of the current systems are essentially similar; differences stem mainly from the unique characteristics of the hardware they utilize. The following writeup describes the principal features of the NAVCOSSACT Information Processing System in terms of the CINCPAC System.

The NAVCOSSACT Information Processing System is a set of programs which generates and maintains formatted files of information and selectively retrieves data from the files. These programs are:

(1) File Maintenance Subsystem, which generates and updates formatted files on magnetic tapes.

(2) The Information Retrieval Subsystem, which retrieves information from magnetic tapes and formats printouts in a useful manner.

(3) Library Maintenance Subsystem, which generates and updates the information processing system library. The library is a magnetic tape file of several types of tables used by the file maintenance and retrieval programs.

The system is said to be generalized because it will process any members of a group of files having a specified structure. Generality is achieved

through the format table. Every master file has a format table which identifies the items or data fields in every record in the file and describes the characteristics of the items. The user of the information processing system describes the operations to be performed on a file in a language which resembles a procedure-oriented programming language. Routines in the information processing system programs use the data descriptions in the format table to translate statements in the language into tables which direct the programs in performing specific operations.

The File Maintenance System generates and maintains the data files. The user defines the format of each file through the use of a file format table and then manipulates the data through the use of the provided Macro Instructions.

The File Maintenance System allows the user to define Argument-Function Value or Code Conversion tables. These tables are used to convert coded information to uncoded form and vice-versa. They are referenced by the table LOOK UP Macro Instruction.

The Macro Instructions are divided into four basic categories: (1) environment establishing instructions to step through the file, (2) data handling instructions, (3) control instructions for branching, and (4) output instructions. They are composed of a Tag Field, an Operation Field, and a Variable Field. Examples of data handling and environment establishing instructions are:

TAG MOVE B TO A $

STEP AIRFIELD RETURN TO TAG $,

where TAG is a label, AIRFIELD is a file, and A and B are variables.

Macro instructions may move information from a transaction record to a master record. Data conversion and generation of new data may be performed. Source data may be checked for validity, and messages may be written on a print tape to identify errors which occur during processing.

A transaction file may consist of source data on punched cards, punched paper tape, etc., or it may be a master file. A source data transaction file is a formatted file, although it does not conform to the specifications for master files. It may, in fact, have records in two or more different formats,

in which case each record will have a field identifying the format of the record. Associated with each different transaction record format is a set of macro instructions which specifies the operations to be performed on records having the designated format.

When a master file is generated, records are read sequentially from the transaction file, with the designated set of macro instructions being "executed" each time a record is read. When the file maintenance program determines that all transaction records having the same keyfield have been read, a record is written on the new master file. In the special case of each transaction record having a unique keyfield, a master record will be written for each transaction record that is read.

When a master file is updated, records are read sequentially from both the transaction and master files. When a match occurs between transaction and master keyfields, a set of macro instructions is executed, causing a master record to be updated. If the purpose of a transaction record is to generate a new master record, the file maintenance program will recognize that no corresponding record exists on the master file being read and will execute the macro instructions at the proper time.

The information retrieval program has a COLINGO-like query language with approximately the same retrieval capabilities. It accepts as input a master file and a set of one or more queries requesting information from the file. Queries are written in information retrieval language, in free form, on punched cards. There is no provision for on-line typewriter input. A query has two major parts: an "output statement" which designates the items requested as output and the format of the output, and a "search statement" which specifies the conditions which items in a record must meet for information to be retrieved from the record. The output of the program is written on one or more tapes for printing.

The program is modular and operates in either three or four phases, depending upon the type of output desired. The first phase translates a set of queries from information retrieval language to tables which control the operation of succeeding phases. The second phase searches the master file for records which meet the conditions specified in the search statements of queries. When a record is found which satisfies the conditions stated, all or part of the record is written on one of two scratch tapes, depending upon the type of output desired. Some queries do not require a search statement; they will cause data to be taken from every record of the

master file and written on a scratch tape.

The third phase is a magnetic tape sort routine. Certain types of output require that the data be sorted on one or more items. The records to be sorted are on one of the two scratch tapes generated in phase two. The fourth phase reads the scratch tapes, edits the data, and writes the output tapes for printing.

There are several minor differences between the Information Retrieval System (IRS) and COLINGO query languages such as:

- IRS may use a function which requires computation as criteria to qualify a record. COLINGO would require one pass of the file to compute the function and another pass for qualification.

- A pattern compare capability (IF MORALE EQ EXCELLENT would qualify in IRS if the variable item contains the value THE MORALE OF COMPANY B IS EXCELLENT.)

- The IRS SUM verb may have several operands; COLINGO allows only one.

- IRS requires parentheses around operands, if they are values.

- IRS has a more powerful COUNT verb which will give subtotals for the count of each property value and a total count for the property. COLINGO will only give the total count of the number of records that qualify.

- The IRS SORT routine gives an ascending sort. COLINGO has both ascending and descending sorts.

- IRS does not have the capability for conditional branching while executing a string of stored queries.

- IRS has approximately the same output capability that COLINGO has and both allow the user to add his own special output routine.

The Library Maintenance System provides the means for the generation and maintenance of the Library File, which corresponds to COLINGO's QUIC files. The files contain a master dictionary, stored queries, format tables

macro-instruction lists, and code conversion tables. Updating a Library File consists of adding, deleting, or replacing an entire file or table.

NAVCOSSACT Information Processing System has a considerably greater number of error messages than COLINGO has; however, due to the restrictions on the format of the card input, it is also easier for an error to be made.

The hardware environment, in existence in the facilities for which NAVCOSSACT has been responsible and IBM has provided the programming systems, is quite dissimilar. The hardware at the various centers include the IBM 7090, the AN/FYK-1(Y) Data Processing Set produced by CDC, and the AN/USQ-20 produced by Sperry Rand Corporation.

Outlined below are hardware configurations and the IP systems that have been designed for each . (These do not represent minimum configurations on which the systems will operate.)

NAVIC Information Processing System

IBM 7090

Hardware configuration:

1 IBM 7090 computer with 32K word memory
11 IBM 729 VI tape drives
4 IBM 729 V tape drives switchable with the IBM 1401 computer
2 IBM 1301 (4 modules) disk units
Capacity is about 108,000,000 characters

CINCPAC Information Processing System

AN/FYK-1(Y) Data Processing Set

The AN/FYK-1(Y) Data Processing Set is an integrated system consisting of 1604A and 160A CDC computers with switchable peripheral equipment. The AN/FYK-1(Y) is implemented at CINCPAC.

Hardware configuration:

4 CDC 1604A computers with 32K word memory
IBM 1402 card reader/punch
(up to) 28 IBM 729 IV tape drives
1 Analex printer
1 CDC 160A for the Master Control
4 CDC 160A computers

Remington Rand USQ-20

An information processing system has been developed for the USQ-20 computer in support of NECPA.

Hardware configuration:

1 USQ-20 with 32K word memory
12 RD 1241 tape drives
1 Paper tape reader/punch
2 Analex printers
1 AN/UGC-13 teletype
2 Card readers
2 Card punches

C. TUFF-TUG

The systems known as TUFF (Tape Update of Formatted Files) and TUG (Format Table Tape Updater and Generator) were earlier generation programs for the 704/7090 which were functionally equivalent to the NAVCOSSACT File Maintenance and Library Maintenance Systems discussed above.

D. 1410 FORMATTED FILE SYSTEM (FICEUR)

The IBM 1410 Formatted File System (FFS) is a generalized system of programs designed to build, maintain, retrieve from, and produce reports on a wide variety of information files. The system is in operation at a number of Air Force Commands as part of the Intelligence Data Handling System (IDHS) centers as well as in the Fleet Intelligence Center Europe (FICEUR).

Files in the Formatted File System look much the same as COLINGO files. Both systems allow for three levels of organization, and both systems maintain a dictionary or file format table describing the file structure. The one significant difference in file structure is the Formatted File System's provision for variable length fields.

In the Formatted File System the user may, by use of conversion tables similar to the code conversion tables in the NAVCOSSACT system, prepare queries in item names familiar to him. The system, using the conversion table, will substitute the coded item names before executing the query. This feature is useful if the data in the file is received in a coded form or if the data was coded to optimize use of retrieval logic.

The files, which look much the same as COLINGO files, may consist of any type of information which is definable to the system. The information can range from small, fixed-length items to indexes of documents and graphic materials containing repeating items of information and textual information. It is a system which was designed to allow the user flexibility in dealing with his data. Constraints on the user have been kept at a minimum; he is not limited to the number of files he can define to the system, nor is he committed to the format of an existing file in the event that he wants to change that format. In many cases, the user is able to accept data previously undefined to the FFS into his system without new programming. He is permitted to create any number of output formats from a file and can merge information from a number of files. The user has a logical, English-like, retrieval language at his disposal with which to seek out the answers to his interrogations, and the ability to format those answers in a variety of ways.

The Formatted File System can be operated on any IBM 1410 (or 7010) with the following minimum configuration:

*1 Central processing unit - 80K core positions
6 Magnetic tape units (729)
1 High-speed printer (1403)
1 High-speed card reader/punch (1302)
**1 1301 Disk - Mod 1

The program system, which contains approximately 32,000 instructions, is of modular design and contains the following modules interfaced with the IBM 1410/7010 operation system:

File generation
File maintenance
Retrieval
Output

File Structure

The FFS is capable of processing data records which can contain three different types of fields:

1. Fixed Fields - which occur only once in a data record and occupy a fixed position within the record relative to the intitial character of the record.

2. Periodic Fields - which can occur more than once within a data record. They contain a category of data which is repetitive in nature. One occurrence of related but different periodic fields is termed a periodic subset. All occurrences of a subset within a data record is called a periodic set.

3. Variable Field - which is included to accept qualitative information for which no fixed or periodic field is provided. The contents of the variable field are unrestricted. The length is limited only by the data record size. Each data record is limited to one variable field.

---

*Original system was programmed at 40K but advanced versions require 80K. The USAFE system is a 100K version.

**Optional for all except the DIA/IDHS Mod 1 version.

The fixed fields appear first in a data record, followed by the periodic fields, and terminating with the variable field.

The general record structure is subject to the following limitations:

Maximum number of characters in a data record	5400
Maximum number of field definitions per record	299
Maximum number of periodic set definitions per record	8
Maximum number of characters in the record control field	30
Maximum number of characters in a field or group of contiguous fields to be referenced collectively	52
Maximum number of subsets within any periodic set	599

File Maintenance

The FFS has the capability to perform the following file maintenance functions:

1. Create a new data record and insert it into the specified file.
2. Delete an existing data record from the file.
3. Change an existing data field (fixed or periodic) within an existing data record.
4. Add a periodic subset to an existing data record.
5. Delete a periodic subset from an existing data record.
6. Add a variable field to an existing record.
7. Add additional information to the existing variable field of a data record.
8. Delete the entire variable field from a data record.

In all the file maintenance functions, the data record to be updated is identified by the record control field (sort key).

The FFS will accept input records on either punched cards or magnetic tape. The input transactions to the update program can either contain single field values along with the name of the field they represent, or they may be packed fields described to the system by a separate descriptor table.

The input fields can be accepted in the input format, or they can be edited and converted by use of tables or subroutines specified in the File Format Table. Updates to several files can be processed during a given run.

The system user can elect either of two options to record the results of a file maintanance run:

1. All transactions are listed with the "from" and "to" values of the field(s) affected by the transaction. Erroneous transactions which have not been effected are also listed with a diagnostic code indicating the nature of the error (e.g., attempt to change a non-existent record).

2. Only the erroneous transactions are listed with their diagnostic codes.

Retrieval

The FFS is capable of retrieving records from the data files based upon a logical combination of conditional statements about the values contained in the fixed and/or periodic fields in the data record. The general form of a conditional statement is:

(Logical Connector), (Field Name), (Operator), (Data Values)

A sample query with three statements is:

IF, DAY OR YEAR, IS GREATER THAN, 192,
AND, COUNTRY NAME, IS EQUAL TO, CANADA,
OR, COUNTRY NAME, IS EQUAL TO, UNITED STATES, MEXICO.

Synonyms may be substituted for field names and operators. Subroutines and tables may be specified for editing and converting data values to the data file format. The logical connectors permitted are IF, AND, OR. The standard operators are:

EQUAL
NOT EQUAL
LESS THAN
LESS THAN OR EQUAL TO
GREATER THAN
GREATER THAN OR EQUAL TO

In addition to the above, the FFS allows the Overlap (OVP) operator, which tests to determine whether a convex geometric shape (circle, polygon, line, or point) defined in the query statement overlaps a convex shape (not necessarily the same type) in the data records. The FFS Query Language allows one level of parenthesis in the combination of logical statements. The Condensed Format feature allows the specification of a query in skeleton form followed by a list of data values; the values are then substituted into the query skeleton and executed by the retrieval program.

Queries can be batched to search one or several files in a given retrieval run.

The user may specify that the results of his retrieval run be sorted on the values contained within specified fields (fixed or periodic) of the records. When queries are levied against several files, the retrieval records from the different files may be sorted on values common to the different files to produce a multi-file report.

The FFS includes the capability to define up to two file indexes per file. The indexed file feature improves query response time through two reductions in the number of record searches per run:

1. The Query is applied only against data records identified by the file index(es).

2. The retrieval program passes only those tape reels (in the case of multi-reel files) which contain data records identified by the file index(es).

The FICEUR query language capability is similar to COLINGO's with the following differences:

(1) No on-line typewriter input capability is provided.

(2) All elements of a query (operators, item names, etc.) must be set off by commas.

(3) One level of parentheses is allowed. (COLINGO does not allow parentheses within a query.)

(4) A period has a special function of specifying the mode of logic, i.e., whether it is intra- or inter-record logic.

(5) Capability to merge files is provided.

(6) Capability to prepare outputs using data from several files is provided.

(7) Circle and polygon search routines are provided. It is possible to do the equivalent of these searches in COLINGO by using the Great Circle Distance subroutine and stored queries, but not via the more convenient and efficient subroutines.

(8) Capability to order data records retrieved from a single file by employing a criteria not explicitly in the data records is provided.

Output

The FFS contains a generalized output capability. It allows the production of reports on any combination of three output media:

(1) Printer listing.
(2) Punched cards.
(3) Fixed-length magnetic tape records.

The user can control the output format by specifying:

(1) Formatting specifications showing the position of the data.
(2) Control information specifying conditional action.
(3) Data manipulation information specifying counts, totals, subtotals, and integer computations within a record or across records.

(4) Printed report pages specifications including absolute and conditional spacing, skipping, and ejection to a new page.

Printed reports consist of page headers and trailers and one or more types of body lines. Body lines may be data fields, constants, computed values, or combinations of the three. Fields (either data or computed values) may be edited to remove leading zeros, insert decimal points or other characters, and other standard editing functions; fields may also be converted by use of special purpose subroutines which have been entered into the system.

Punched card and magnetic tape reports may contain all the above options except those unique to a printed report, such as page headers and trailers and spacing, skipping, and ejecting. Magnetic tape reports may include several types of records with the restriction that all records be the same length. The record length may be specified to the program within a maximum value, which will be specified at each installation based on machine core storage size.

The Output Program also contains an extract option which allows a user to output data from a file without defining a formal output format. The user identifies the desired fields by name; the report contains a tabulation of the specified field values with the approved field labels.

E. THE 473L SYSTEM

System 473L is the Headquarters, United States Air Force, Command and Control System. It is an on-line real-time information processing system designed to facilitate management of USAF resources, particularly during emergency situations. The system is being developed in three major increments, each increment having successively greater capability relative to operational scope, equipment, and programs.

The initial increment (OTC) utilized an IBM 1401 Data Processor subsequently replaced by an IBM 1410 Data Processor. The peripheral equipment included an IBM 1402 Card Reader Punch, IBM 1301 Disk, IBM 7330 Tape Drive, IBM 729 Tape Drives, IBM 1415 Console Inquiry Station, IBM 1403 Printer, and a TRW Computer Communications Console. The second increment utilizes a Librascope AN/FYQ-11 Data Processor, ITT AN/FYA-2 Integrated Consoles, Plotter Tape Transports, IBM 1402 Card Reader Punch, and an Analex Printer. The final system will include a second set of the aforementioned equipments which will provide a

dual system, capable of sharing the work load and of providing system back up.

Input data is received on-line from the AUTODIN network. On-line staff control is provided for, both in the preliminary screening of AUTODIN messages and in the control of outputs at the individual consoles.

The 473L System provides support to the Air Staff in the following major functions: situation monitoring, resource monitoring, plan evaluation, plan generation and modification, and operations monitoring. Support is provided to each of these functions in the form of specific operational capabilities. An operational capability is a set of interrelated computer programs designed to provide a console operator with information supporting solutions to significant and predefined problems encountered in the exercise of USAF responsibilities for evaluating, modifying, or monitoring plans or operations.

Thus a central 473L function is information retrieval, and it is in this area that its major contribution to system flexibility exists. Two major methods of communication between the system operator and the data processing subsystem have been provided in the 473L System. The first major method of communication is the operational capability overlay -- a plastic mask which fits over the Integrated Console Logic Keyboard. This mask delineates the set of programs related to a specific operational capability. Operation via the overlay provides, upon activation of the logic push buttons, an indication of which push buttons may legally be activated, presents a cue message on the display, allows alpha numeric data to be entered, and responds to the completed cue message by displaying a follow-on cue message or a data output. In this manner, the operator can exercise a choice at each decision point and is led in a step-by-step manner through the procedures necessary to obtaining the desired information. The console employed in the 473L system is the TRW Space Technology Laboratories Computer Communication Console. (A description of the capability of a similar console is that of the Culler-Fried console following).

The second major method of communication is via the Query Language. The Query Language is a quasi-English language, very similar to the COLINGO query language, which enables an operator to retrieve specific data in one of several formats from any system file. It is also used

internally by all other programs as a means of retrieving data from the files.

In using the Query Language, the operator enters his query directly by means of the typewriter portion of the Integrated Consoles. The program interprets the statement, makes the desired retrieval, and formats the output. In addition to direct retrieval, the Query Language includes the following functions:

- Computation, e.g., summing numerical values.
- Logical operations, e.g., finding largest or smallest values.
- Sorting retrieved data for output.
- Manipulating files, e.g., combining parts of two files into one.
- Storing and recalling frequently used query statements.

The Query Language can also be used as a tool for system programmers in that it permits a sequence of operationally related statements to be processed as a group, has very powerful file reorganization capabilities, and has variable output format capabilities.

Certain noteworthy characteristics of the language follow.

The language allows a file to be qualified by any number of values of attributes. In addition, the attributes can be embedded in functions. Thus, it is possible to ask for great circle distances within a specific distance of some point in a single query. For example:

> Retrieve airfields with country = Brazil, GCD (Brazilia $\leq$ 2000).
> This query asks for airfields in Brazil which are within 2000 miles of the city Brazilia by the great circle distance.

Two flexible printouts to queries such as this are provided. One has a vertical orientation of data and one a horizontal orientation. It is also possible to entitle these printouts, such titling being introduced through the query language statements. Stored statement routines are also available for storing completed statements. These may be recalled at a later time for additional use. As part of this capability statements with built-in

blanks can be stored for later recall. At that time, the particular values appropriate for the moment can be filled into the blanks and the resultant statement then run. This insertion of values into blanks is accomplished through the aforementioned Computer Communication Console. This step serves to illustrate an additional source of 473L flexibility, namely, the complementarity of the push-button and query language techniques, since both are integrated and can be used in conjunction with each other.

The 473L files are, like its query language, similar to the COLINGO files. Thus, they are both formatted and hierarchically structured. Several types of file structures are allowed in the 473L system, mostly to make best use of the characteristics of the storage media; for example, serial versus random access. Examples of these structures are: packed data, blocked data, fixed or variable length entries, and combinations of these structures. The attributes of the items in the file are distinguished by type; such as, fixed or variable length, single or multiple value, etc. In addition, there are flexible provisions for checking and internally encoding the file data as desired. The system also maintains a data description file which contains all information required to process the data files. This data description file is part of a generalized file maintenance program which will take a wide range of possible data values, convert them and store them in the structured data base. Both the data and the data descriptions, including a dictionary of all external versus internal names, are kept in the data base. Having available the descriptors and the dictionary as well as the data, then, the query language can effectively retrieve as well as store data. This generalized file maintenance program with its data description file and file dictionary corresponds to the COLINGO file dictionary and is yet another source of system flexibility, as it is in COLINGO.

Some capabilities which are or soon will be part of the 473L system that COLINGO does not have are: multiple time-shared user consoles, remote I/O, priority assessment and automatic interruption of a query for a more important task, and distinction (and even simultaneous use) of operations activity and training exercises. Most of these capabilities, however, are advantages to be gained from operation on a larger computer under a more powerful executive. Therefore, 473L and COLINGO are essentially comparable in terms of information system flexibility and both represent an advanced state-of-the-art in information retrieval system design. The major advantage that 473L has over COLINGO, is not so much in terms of system flexibility as it is in terms of efficiency, 473L being the more

efficient of the two. This is best exemplified by a comparison of the two query languages. Specifically, qualification of a data record in 473L can depend on a function derived from several property values in the record. (This is also possible in NAVCOSSACT.) This can be done in a single pass. COLINGO, in contrast, would require two passes of the file, one pass to compute the function based on the appropriate property values, and a second pass to determine the correct qualification from the computed values. Similarly, there is a degree of efficiency represented by the 473L (and NAVCOSSACT) SUM operator not found in COLINGO. This operator permits simultaneous summing of several properties while making only one pass at the file. These two examples exemplify the greater efficiency, if not power, of the 473L system.

F. THE ACSI-MATIC SYSTEM

ACSI-MATIC is an acronym for a proposed information storage and retrieval system which was to be designed and implemented by RCA for the Department of the Army, Assistant Chief of Staff, Intelligence. It is primarily an information retrieval system, although it provides a good measure of generality and self-sustaining control. The prototype was implemented on a Sylvania 9400 computer.

The system was intended to provide a common repository of information which could be used by intelligence analysts with different area responsibilities. The system was also designed to reduce the amount of effort required of analysts to evaluate and file new information. This latter function was to be achieved by automatically collating new and old information on the basis of the information content. All information stored was to be indexed as thoroughly as possible by the system to provide the analysts with an efficient and comprehensive retrieval capability.

The unit of information storage in the ACSI-MATIC file structure was the Information Record of which there were different types, such as Military Organization, Personnel and Installation. Data coming into the system might be logically filed in more than one type of Information Record. All data filed by the system were indexed and cross referenced by the system. Each Record type had a set of allowable information that could be stored in it. The formation of new information records was handled both explicitly as the result of a direct order, and implicitly, as a result of a gradual acquisition of information.

To provide a means for the system to accept and manipulate information without excessive external coding by the people preparing the data to be input, a glossary of common terms was stored permanently within the system. To provide information to the system about the relationships between terms, the total number of terms was sub-divided by the users into functional categories. A glossary was set up for each category. In general, the alphabetic composition of a term contains no relational information. Hence, the users established tree structures for all terms within a glossary. The position of a term in the tree was specified by a binary code called Flexicode, which has the property that both the vertical and horizontal coordinates of a term are well specified. Further, given any two codes, a third code can always be inserted between, in either direction. An entry in a glossary, therefore, is a pair; namely, an English term and an associated Flexicode. The glossaries are maintained in alphabetic order.

The aforementioned Flexicodes are maintained in "hierarchical" order in the Hierarchy Index List. Each entry in the list contains the Flexicode for the term it represents, plus a pointer to each Information Record in the file containing information about the term.

The system receives messages and orders from the analysts by way of a paper tape reader. System responses were on a monitor typewriter and high speed printer. All elements of information in a message or order were tagged to make them identifiable by the system. For example, information about an individual would be handled as follows:

> PER:: NAM:: SMITH, JOHN H.:: JOB:: CIVIL ENGR:: AGE: 42::
> The tag PER identifies the type of information being dealt with - personnel. The tag PER has a number of allowable sub-tags, including NAM, JOB, AGE, with obvious meanings. The character "::" is a delimiter to make scanning easier.

Each message or order also contained information about the message (order) and the data contained therein. Examples of these are:

- Name of person putting in data
- Source document identification
- Source document data

Security classification.

Orders were designed to provide the analysts with the capability to explicitly define some types of system processing. These included:

- Establish a new Information Record
- Change an Information Record
- Perform Thesaurus Maintenance
- Report (print any of a number of standard reports)
- Query (interrogate the system files in complex manner).

Messages were designed to be the method of adding new information to the system files on a continuous basis. It was not necessary for the person preparing the message to know anything about the state of the system files, but only to properly format and tag the information being entered.

An initial data base of known information was to be entered using analyst orders. For each Information Record type there were criteria to determine when new types were to be established as data were received. Each element of information filed in the system was cross referenced as thoroughly as possible. For example, data input via an analyst order established a Record for an installation at location L. Part of the data kept with this Record was a list of all technical personnel, which included the name Smith, J. H. . At the time the order was given and Record established, a reference to the Record was made in the Hierarchy Index List entry for Smith J. H. . At a later time some information entered the system via a message which reported that a Smith, J. H., a civil engineer, had been given an award for outstanding work. In the List entry for Engineer, Civil, a reference was then made to Smith, J. H. and conversely, under Smith, J. H. a reference made to Engineer, Civil. The system now had two pieces of information about Smith, J. H.: (1) he was at location L; (2) he was a civil engineer. At some point there might be enough information on Smith to warrant the establishment of a Record, type Personnel, for Smith. If this was done, the List entry for Smith would point to the Record of the installation at L and to the Record for Smith. The List entry for Engineer, Civil would also point to these Records. The previous references which were only to the name Smith and

the name Engineer, Civil were thus replaced by references to collections of information about each name. This example illustrates the dynamic quality of the files which were intended to grow as a function of the data being input without explicit orders from the analysts. The decision making processes and the actual indexing procedures were more complex than those illustrated.

The system was designed to handle complex Boolean expressions which specified retrieval requirements. Basic retrieval operations involved manipulating, when possible, the indices rather than the data during the selection process. For example, consider the query, "List all civil engineers at installation at location L." The List entry for Engineer, Civil has a pointer to the List for each man known to be a civil engineer. Each of these entries has references to the Records which contain information about the man named. These references are the disc locations of the Information Records. The installation at L has a Record stored at a known disc location. Therefore, if this address appears in a List entry for a man known to be a civil engineer, he is known to be at installation L.

The programming system for ACSI-MATIC was composed of an input-command package, an executive control package, an input-output package, and a collection of utility packages.

The input-command package accepted and interpreted commands for the execution of major system functions and for establishing the appropriate programs to effect the execution of those functions.

The executive control package for ACSI-MATIC furnished a collection of functions to enable the linking, loading, binding, and execution of system tasks. In addition, functions to enable parallel processing up to sixty-four tasks and dynamic allocation (i.e., requests for more space generated at execution time) were available. Basic blocks for loading, linking, and parallel processing control were kept in terms of chains of descriptions; i.e., lists of descriptive information were organized via chain links to reflect program relations, resource allocations, and priority for execution.

The ACSI-MATIC programming system furnished to the user a variety of utility packages. For example, requests for retrieval of items from the disc file by name were accepted and handled by the Thesaurus Access

Package. In addition to retrieving data from the files by name via indices, this package furnished all of the machinery for adding and deleting entities of the files. Techniques for interlocking accessing during maintenance and for preserving accessed information in core for subsequent use were developed.

A second utility package was a comprehensive sort-merge package which was available to the user. This program allowed sorting on the basis of complex user-furnished ordering criteria.

A programming language and an assembly program were also developed. The assembly produced relocatable code and sufficient descriptive information to satisfy the requirements of the dynamic allocation and linking algorithms.

PART III. MAN-MACHINE INTERFACE SYSTEMS

In this section systems which have concentrated on the enhancement of man/machine communication through the development of various types of response techniques are discussed. Each of the systems mentioned herein has contributed to man/machine interface techniques either by introducing a new dimension along which communication can occur, or else by developing new techniques for improving the communication along familiar dimensions. Thus, SKETCHPAD has added the graphical dimension to computer interfacing and BASEBALL has demonstrated the feasibility of using ordinary English, albeit of a constrained variety, as a query language. The Culler-Fried and JOSS Systems, in contrast, have introduced new techniques for enhancing communication along a familiar dimension of man/machine communication, namely mathematics. In each case it is important to note that the system designers had in mind a wide class of users operating within a broad spectrum of problem areas. Thus, the techniques they have introduced are quite general in nature and of wide potential use.

A. SKETCHPAD

The development of SKETCHPAD marks an advance in the exploration of the dimensions along which an operator may communicate with the computer. The particular dimension explored by SKETCHPAD is graphical and the basic aim of the system has been to make rapid interaction between man and machine possible through graphical means. SKETCHPAD is fundamentally a drawing system which has been implemented on the TX-2 computer at the M. I. T. Lincoln Laboratory. It places at the disposal of the user "paper" in the form of a cathode ray tube and "ink" in the form of a hand-held photo-cell called a light-pen. In addition, it places at the user's disposal a multitude of graphical operations which enhance his ability to make complicated line drawings from relatively simple subparts. It provides the user with the ability to copy his light-penned drawings onto a permanent pen and ink plotter. Finally, the SKETCHPAD system has associated with it a tape library in which previously drawn figures may be stored for later use.

A SKETCHPAD drawing sequence may typically begin with the operator pointing the light pen at the display system and simultaneously using a "draw" button, "draw" being one of the primitive functions provided by the system. The computer then constructs a straight-line extending from the point the light pen originally identified to whatever subsequent point has been chosen by the user. The user does not employ the light pen as he might a pencil and ruler. He simply identifies an initial point and activates the "draw" function. The computer then constructs the line. In particular, when the "draw" button is pressed, the computer sets up two end points and a line segment in storage. One of the end points remains attached to the light pen, however, and the subsequent pen motions move this second point. This action of establishing pairs of points and a line segment in store may be repeated several times. Thus, to construct an hexagonal figure one would repeat the operation six times, once for each side of the figure. In order to close the lines into a six sided figure, it is merely necessary to return the light pen to a point near the end of the first line drawn. The figure will automatically close on itself. The drawing may then be terminated by a control on the light pen.

In this fashion an irregular hexagon has been constructed. A better view of the range and scope of SKETCHPAD can be gained when one attempts to construct a regular hexagon. To construct a regular hexagon, use can be made of the irregular hexagon just constructed and other primitive functions available in SKETCHPAD. To make the irregular hexagon regular, one could employ the following strategy. One depresses a "circle center" button while identifying a center with the light pen. Then one identifies a subsequent point to mark a radius and presses the "draw" button. SKETCHPAD will construct an accurate circle. Having established a circle, one may point to a corner of the irregular hexagon and then, by depressing a "move" button, bring one corner of the hexagon into the circle. The lines converging at that corner will follow the light pen in rubber band fashion into the previously drawn circle. Each corner may be thus moved inside the circle and the irregular hexagon can thus be circumscribed.

To make the hexagon regular, one points to one side of the hexagon, depresses a "copy" button and points to another side. SKETCHPAD will then produce a regular hexagon, if possible, after five such repetitions. A depression of a "delete" button erases the circle, and the user has constructed a regular hexagon.

Thus, several primitive functions have been identified and illustrated, namely "draw," "center circle," "move," "erase," "copy," etc. With these functions an irregular hexagon was constructed and from it, a regular hexagon. Subsequent use of these and similar functions make it possible to construct extremely large hexagonal mosaic patterns such as one might find on hexagonal graph paper, for example. A professional draftsman would require two days to construct a 900 hexagon mosaic on a 30 in. by 30 in. plotter. Including the time to decide upon the above strategy, SKETCHPAD required only half of an hour. The example just discussed illustrates the fundamental way in which use of SKETCHPAD differs from normal pen and ink drawing. In the above example the user created a rough approximation of a figure and then, by the addition of constraints to that figure, relied upon the computer to systematically apply those constraints to each part of the initial rough figure. It is, in fact, the contention of the SKETCHPAD designer that this process of a rough approximation followed by the imposition of new conditions is essentially the design process that any designer might go through in the creation of a finished design from a basic idea. As new requirements are created, SKETCHPAD supplies the mechanical means whereby those requirements may be imposed upon that which has been done, thus relieving the designer of much of the detailed work required by the imposition of the newer constraints.

The various design activities for which SKETCHPAD may be used are, perhaps, obvious. Thus, it is of utility in the production of highly repetition drawings, drawings where accuracy is required. Moreover, it has been demonstrated in drawing mechanical systems such as linkage suspensions and in manipulating them to observe the resultant motions.

In addition, the use of SKETCHPAD as an input program for other computation programs is suggested. Thus, with this system, truss bridge designs can be generated with designated load points specified. Subsequent processes can compute the resultant forces based upon the truss bridge design and the specified loads. In this fashion SKETCHPAD can be made an engineering design tool. Similarly, the use of SKETCHPAD in circuit design is certainly suggested, though the computation required by electronic circuitry goes beyond the capacity of SKETCHPAD at the moment.

The SKETCHPAD Program, which was written over a one year period and required approximately one and a half man-years of effort, comprises 12,000 words of executable program, 20,000 words of 'ring structure'

[illegible] 30,000 words of input-output buffer. The TX-2 [illegible] in which it operates is a single-address, binary digital computer with an unusually large memory. It is an experimental machine, however, and as such contains certain distinctive features. These are: (1) simultaneous use of in-out machines through interleaved programs; and (2) flexible, 'configured' data processing. In addition, the TX-2 includes automatic memory and arithmetic overlap, a 'bit'-sensing instruction (the operand is one bit), addressable arithmetic element registers, 64 index registers, indirect addressing and magnetic tape auxiliary storage.

In summary, the use of SKETCHPAD exists in: (1) making small, systematic changes in existing drawings; (2) in gaining scientific and engineering understanding of operations that can be described graphically; (3) as a topological input device for complex patternings such as required by networks and circuitry; (4) for highly repetitive drawings requiring recursively generated sub-pictures has been demonstrated. That the system has been made to draw electrical, mechanical and even animated drawings qualifies it as a general-purpose, user-on-line,man-machine communication technique.

B. BASEBALL

Whereas SKETCHPAD demonstrated the feasibility of man/machine communication through the medium of graphical display, BASEBALL may be viewed as demonstrating the feasibility of man/machine communication through the medium of a relatively flexible query language, flexible with respect to the typical query language available in contemporary retrieval systems such as those discussed in Part II. The query language of the BASEBALL system is, in fact, a small, but nonetheless varied subject of ordinary English. And since every potential user of a system can be supposed to know this subset already, there exists with respect to the BASEBALL user no requirement for training in its query language. Thus anyone who speaks English and is familiar with baseball can query the system with a minimum of training.

BASEBALL is a computer program that answers questions posed in ordinary English about data in its files. The program consists of two parts. The linguistic part reads the question from a punched card, analyzes it syntactically, and determines that information which is given about the data being requested. The second part, the processor, searches through

the data for the appropriate information, processes the results of the search, and prints out the answer.

The program is written in IPL-V, an information processing language that uses lists, and hierarchies of lists, called list structures, to represent information. Both the data and the dictionary are list structures, in which items of information are expressed as attribute-value pairs, e.g. Team = Red Sox.

The program operates in the context of baseball data. At present, the data are the month, day, place, teams and scores for each game in the American League for one year. In this limited context, a small vocabulary is sufficient, the data is simple, and the subject-matter is familiar.

Considerable flexibility is permitted in the wording and the form of input questions. The program accepts any grammatical question that is made up from words found in the BASEBALL dictionary, and that adheres to the following restrictions. Questions are limited to a single clause. By prohibiting structures with dependent clauses the syntactic analysis is considerably simplified. Logical connectives, such as and, or and not are also prohibited, as are constructions implying relations like most and highest. Finally, questions involving sequential facts, such as "Did the Red Sox ever win six games in a row?" are prohibited. The designers note that within these restrictions, complete freedom is allowed with regard to word order, phraseology, verb voice, etc., and that the user is free to achieve considerable variety and complexity in question format. There is considerable leeway between simple questions such as "Who did the Red Sox lose to on July 5?" and more complex questions such as "Did every team play at least once in each park in each month?" Some sample questions that the program handles successfully are:

Did every team play all of the teams at least once in May 1959?
Had any team scored more than 8 runs by May 15?
How many games were lost by a 2 run margin to the A's?
What were Detroit's winning margins at home before June 1959?
What were the scores of the Senators losing home games?
How often did Baltimore beat New York at home in August?
How many home games did New York lose in June by one run?
When did the Red Sox win by more than 7 runs?

The concept of the specification list is fundamental to the operation of the BASEBALL program. This list can be viewed as a canonical expression for the content of the question. It represents the information contained in the question in the form of attribute-value pairs such as Team = Red Sox. The spec list, which is generated from the question by the linguistic part of the program, governs the operation of the processor. For example, the question "Where did the Red Sox play on July 7?" has the specification list:

Place = ?

Team = Red Sox

Month = July

Day = 7

The dictionary definitions, which are expressed as attribute-value pairs, are used by the linguistic portion of the program in generating the spec list. A complete definition for a word or idiom includes a part of speech, for use in determining the phrase structure of the question; a meaning, for use in analyzing content; an indication of whether the entry is a question-word, e.g., who or how many; and an indication of whether a word occurs as part of any stored idiom.

The meaning of a word can be expressed in one of several forms. It may be a main or derived attribute with an associated value. For example, the meaning of the word Team is Team = (blank), the meaning of Red Sox is Team = Red Sox, and the meaning of who is Team = ?. The meaning may designate a subroutine, together with a particular value, as in the case of modifiers such as winning, any, six, or how many. Some words have more than one meaning; the word Boston may mean either Place = Boston or Team = Red Sox. The dictionary entry for such words contains, in addition to each meaning, the designation of a subroutine that selects the appropriate meaning according to the context in which the word is encountered. Finally, some words such as the, did, play, etc., have no meaning.

For testing of the system, the BASEBALL data are organized in a hierarchical structure, like an outline, with each level containing one or more items of information. Relationships among items are expressed by their occurrence on the same list, or on associated lists. The main

heading, or highest level of the structure, is the attribute Month. For each month, the data are further subdivided by place. Below each place under each month is a list of all games played at that place during that month. The complete set of items for one game is found by tracing one path through the hierarchy, i.e., one list at each level. Each path contains values for each of six attributes, e.g.:

Month	=	July
Place	=	Boston
Day	=	7
Game Serial No.	=	96
(Team	=	Red Sox, Score = 5)
(Team	=	Yankees, Score = 3)

The parentheses indicate that each Team must be associated with its own score, which is done by placing them together on a sublist.

The processing routines are written to accept any organization of the data. In fact, they will accept a non-parallel organization in which, for example, the data might be as above for all games through July 31, and then organized by place, with month under place, for the rest of the season. The processing routines will also accept a one-level structure in which each game is a list of all attribute-value pairs for that game. The possibility of hierarchical organization was included for generality and potential efficiency.

The BASEBALL program is organized into several successive, essentially independent routines, each operating on the output of its predecessor and producing an input for the routine that follows. The linguistic routines include question read-in, dictionary look-up, syntactic analysis, and content analysis. The processing routine include the processor and the responder.

In the question read-in routine, a question for the program is read into the computer from punched cards. The question is formed into a sequential list of words.

In the dictionary look-up stage, each word on the question list is found in the word dictionary and its definition is copied. Undefined words are printed out. The list is then scanned for possible idioms; contiguous words that form an idiom are replaced by a single entry on the list, and an associated definition from the idiom dictionary, a separate dictionary, is entered beside it. At this point, each entry on the list has associated with it a definition which includes a part of speech, a meaning and certain other indicators, such as sub-routine markers, etc.

The syntactic analysis is based on the parts of speech, which are syntactic categories assigned to words for use by the syntax routine. There are 14 parts of speech and several ambiguity markers.

First, the question is scanned for ambiguities in parts of speech, which are resolved in some cases by examining the adjoining words, and in other cases by inspecting the entire question.

Next, the syntactic routine locates and brackets the noun phrases, the prepositional phrases and the adverbial phrases. The verb is left unbracketed.

Following the question bracketing stage, a routine determines whether the verb is active or passive. This routine inspects the last two verbal elements in the question, and sets a verb-voice marker for use later in the program. Next, the subject and object of the verb are located and labelled. This routine makes use of bracketing information, word order and verb-voice in order to determine subject and relative clauses in the question.

Finally, the syntactic analysis checks to see if any of the words is marked as a question word. If not, a signal is set to indicate that the question requires a <u>yes</u>/<u>no</u> answer.

The content analysis stage uses the dictionary meanings and the results of the syntactic analysis to set up a specification list for the processing program. First subroutines are executed which have been inserted in the dictionary look-up stage. Here ambiguities, such as whether <u>Boston</u> is a team or a place, are resolved. Also attribute-value pairs are modified in appropriate ways. Thus, <u>what team</u> becomes Team = ? in this portion. Similarly Team = (blank) in the phrase <u>each team</u> is altered to Team = each.

After the subroutines have been executed the question is scanned to consolidate those attribute-value pairs that must be represented on the specification list as a single entry. Next, successive scans will create any sublists implied by the syntactic structure of the question. Finally, the composite information for each phrase is entered onto the specification list. Depending on its complexity, each phrase furnishes one or mor entries for the list. The resulting specification list is printed in outline form, to provide the questioner with some intermediate feedback.

The next stage is the processor stage. The previously prepared specific tion list indicates to the processor the part of the sorted data which is relevant for answering the input question. The processor selects the relevant subset of data, and, in stages if necessary, culls from it the information the spec list requests. The processor produces this answer, in the form of a list structure, for the last stage, the responder.

The core of the processor is a search routine that attempts to find a match, on each path of a given data structure, for all the attribute-value pairs on the specification list; when a match for the whole specification list is found on a given path, these pairs relevant to the specification list are entered on a found list. A particular specification list pair is considered matched when its attribute has been found on a data path and, eitl the data value is the same as the spec value, or the spec value is ? or ea in which case any value of the particular attribute is a match.

The found list produced by the search routine is a hierarchical list struc containing one main or derived attribute on each level of each path. Eacl path on the found list represents the information extracted from one or more paths of the data. Each path on the found list may thus represent a condensation of the information existing on many paths of the search da

No attempt has yet been made to respond in grammatical English sentenc Instead, the final found list is printed, in outline form. For questions requiring a yes/no answer, YES is printed along with the found list. If the search routine found no matching data, NO is printed for yes/no ques tion, and NO DATA for all other cases.

Questions which are raised by the BASEBALL program but not answered concern the feasibility of list structures when dealing with data bases that contain hundreds of files, and not just one file as in the BASEBALL program. If list structures are not feasible, then the specification list

would have to be drastically changed. Further, no experimentation is at present available to determine whether, given a choice between a subset of ordinary English and some typical query language such as the COLINGO language or the 473L query language, the user would choose the English subset.

The relevance of the BASEBALL program, therefore, from the point of view of computer usage is that it underscores the necessity for considering ordinary English and its attendant translational difficulties along with the more traditional kinds of query languages and their attendant translational difficulties when confronted with the problem of computer control through some access language.

C. DIRECT ENGLISH ACCESS AND CONTROL (DEACON)

The Direct English Access and Control (DEACON) system is an experimental information retrieval system designed by GE TEMPO. Its goal has been to demonstrate the feasibility of computer control through a system that the designers term "largely unconstrained English." In this sense, then, the DEACON work may be viewed as a continuation of the aforementioned BASEBALL effort.

The DEACON "breadboard," the first designer's model to come out of this effort, consists of a set of programs and data for the GE 225 computer. The current data base contains 3000 items which are accessed by 400 words of vocabulary of which 100 are purely function words. This ratio of 1:4 is expected to increase to 1:20 in a future system with a projected vocabulary of 40,000 words.

The DEACON breadboard data base is list structured and the principle routines and syntactic-semantic rules are programmed in LAP, a list-processing language developed at GE.

The strategy for accessing the list structured data base is contained in the designer's treatment of the syntactic-semantic rules. In BASEBALL a sentence is first analyzed by the syntactic analysis routine to determine its grammatical structure and then is analyzed by the content analysis routine to provide that sentence with a "semantic" reading which is meaningful with respect to the data base. This content analysis routine is programmed in terms of the explicit format of the data base and is dependent on this format. In DEACON, on the other hand, the syntactic

analysis is merged with the semantic analysis into a single stage and each input sentence is subjected to a semantic analysis which provides a syntactic-semantic reading, with respect to the DEACON data base. It is noteworthy, however, that both the sequence "Is the Forrestal in London?" and the sequence "In London Forrestal?" mean the same thing in the DEACON system.

Thus, whereas the BASEBALL program was syntactically too constrained the DEACON program is syntactically too unconstrained, at least with respect to its professed goal of ordinary English.

The syntactic-semantic analysis routines are written in forms of generalized types of data structures, the final identification of which is made in terms of the specific sentence being analyzed. As a consequence DEACON is less dependent upon a rigid data base format.

The similarities between BASEBALL and DEACON are striking. Both make use of list-structures and list processing languages. Both are off-line with sentences introduced on punched cards. Both are accessed and controlled by ordinary English, though of varying degrees of constraint. Both contain dictionary look-up, semantic and syntactic analyses and search routines based upon the results of the preceding routines and finally print-out routines which present the result of the search routines to the user. The major differences between the two programs are the collapsing in DEACON into a single stage of two separate stages in BASEBALL, namely the syntactic and semantic stages, and the greater dependence in the one (BASEBALL) than in the other (DEACON) on a rigi data base format.

Problems which remain for the one system, therefore, remain for the other. These problems are (1) how to effect efficient file up-dating; (2) how to effect efficient access in data bases storing 25,000 files and not just one file; and (3) how to generate new files.

D. JOHNNIAC OPEN SHOP SYSTEM (JOSS)

JOSS is an experimental on-line computational system developed by the RAND Corporation as a problem-solving tool for its scientific/engineering personnel. The system has two separate functions, each of which can be used together or apart. The first of these functions is that of an elaborate printing desk calculator. Thus JOSS can add, subtract,

multiply and divide. It can compute exponents, square roots and trigonometric functions. Further, it is possible to request the fraction, integer, digit and/or exponent part of a number. Logarithms are to the base e in JOSS and decimals carried to the ninth place.

Requests for all of these functions can be made simply and easily once the user has familiarized himself with the keyboard symbols for the various JOSS arithmetic functions. Thus by typing in '2 + 2', JOSS replies '4'. Similarly, '/' will cause JOSS to divide, '.' to multiply and an '*' to raise to an exponent. Thus 'TYPE 3*3' will cause JOSS to reply '3*3= 27.' The expression ' TYPE sqrt (81)' will cause JOSS to reply 'Sqrt (8 1)= 9', i.e., the square root of 81. By enclosing the number to be square rooted in parentheses, it is possible to build up complex expressions. Thus 'TYPE sqrt (sqrt(2*2*2))' will cause JOSS to reply 'sqrt (sqrt(2*2*2))=2'.

All of these functions can be expressed in a single complex function, of course, and in a very short time it is possible for the user to master the 'calculator' function in JOSS.

The second function in JOSS is that it can also act as a 'stored-program' computer. This means that it is possible to enter a statement in JOSS and have it stored away for later use. Thus, by prefixing a statement with an identifying number, JOSS is thereby instructed to store away the sequence following the prefixed number and to be ready to recall that sequence on demand. A typical statement to be entered as a stored program statement is:

1.1 Type X, sqrt (x), log (k), exp (x).

JOSS will store this information. A subsequent command will cause JOSS to execute the statement to whatever value of X the user requires. Thus the statement:

Do step 1.1 for X=1(1)100

instructs JOSS to take the values of X as being from 1 to 100. JOSS will begin to compute the square root, log and exponent of each number from 1 to 100 as specified in the stored program statement 1.1 above.

It is possible at any time to interrupt JOSS and specify a format in which the output of some stored-program statement is to be printed. Thus, the JOSS user can have the various values arranged in a matrix with each number from 1 to 100 at the head of a row and each function to be computed at the head of a column. Demands for spacing can also be introduced. Thus, JOSS can be requested to insert a blank line every five or ten lines, etc.

It is also possible to name formats which have been defined by the user and to request that the output of some stored-programs be printed in the defined format which can thus be reviewed by name. A typical statement with format specified is:

1.1 Type X, sqrt (x), log (x), exp (x) in form 1.

It is worthwhile to note that JOSS is primarily used as a combination of its two functions, namely as a desk calculator in combination with a stored-program computer. This is largely because these combined functions allow the user constantly to exercise judgment during a session on JOSS. This he does by initiating a program, interrupting it, changing it, altering its format, re-specifying its values and its functions and so on. Numerous error messages as well as techniques to avoid catastrophic errors resulting in loss of prior work are abundantly supplied.

The JOSS system operates solely in the interpretive mode. This is to say, with each input statement from a remote typewriter, it picks up the statement, scans it, executes it immediately and then proceeds to the next statement. JOSS is able to complete this cycle on the average of 15 times every second. Because JOSS is time-shared (there are eight remote consoles), the user is occasionally required to wait. The maximum period a user would have to wait, when the system is being used to full capacity, is sixteen seconds.

The amount of space allotted to each JOSS user is quite small, about a page per user. This is due to two factors, the first being the limitations of the Johnniac computer, an old and rather small vacuum tube machine. The second reason for space limitations, however, is that the JOSS designers were simply not interested in large memory capacity. Thus JOSS was intended for users with small problems who would take maximum advantage of the problem-solving capability of a machine with which it was possible to interact easily and quickly. Production computing

interaction of the sort attainable with larger more capacious machines was neither desired nor sought after by the JOSS designers. Now that the interaction capability has been demonstrated, however, attempts to increase computation space per user as well as the number of possible simultaneous users are in process.

The goal of JOSS to develop a tool for problem solving, rather than production computing, has been oriented toward the design of hardware and software for a specific application, namely scientific and engineering computation. In the design of this hardware/software package speed and power have been sacrificed for smooth operation and a relatively transparent algebraic user's language. In this sense, then, JOSS is like STRESS and COGO discussed in Part IV. However, JOSS differs in that the applications class that it addresses itself to, viz., general scientific and engineering computation, is less restrictive than the applications class encompassed by STRESS and COGO. Secondly, JOSS represents an integrated software/hardware design oriented toward a specific application whereas STRESS and COGO are strictly software systems executed on a conventional machine, the IBM 7090.

A major limitation of JOSS is that at the end of a session on the machine the program vanishes. Subsequent operation at a later date on the same program requires complete re-entry of the entire program.

E. TRW TWO-STATION, ON-LINE SCIENTIFIC COMPUTER (OLSC) (ALSO "CULLER-FRIED SYSTEM")

The Culler-Fried System is a computer program developed by TRW Space Technology Laboratories for the RW 400 Computer and Display Analysis Console to provide a tool for the on-line investigation of scientific/engineering problems which can be stated in terms of classical mathematics.

Three principal features, independent but interacting, characterize the system:

Functional Orientation

The programming structure is such that in the computer, as it appears to the user, functions (sets of 101 points) rather than individual numbers constitute the data elements while the

repertoire of "commands" consists of operations on functions (e.g., arithmetic, differential, and integral operations).

Control and Display Capability

Central to the operation of the system is a control console having a number of push buttons or keys, which allow for user control of the computer, and two 17-inch CRT oscilloscopes (with line-drawing capability) which provide direct graphical representation of computational results. An 8-inch CRT with alphanumeric capability and a Flexowriter provide numerical output when required.

Console Programming

A simple procedure allows the user to construct, directly at the console, new subroutines, using as building blocks an initial set of hand programmed subroutines, plus any subroutines previously created by this console programming procedure.

The system incorporates two standard keyboards, one associated with operators, the other with operands plus a keyboard for digit entries. Each operation is carried out by pushing a single key of the operator keyboard. These operations may be at any of a variety of logical levels, for example:

(1) Basic machine language commands;

(2) Elementary arithmetic;

(3) Functional mathematics;

(4) Basic display operations;

(5) System management and data transfer operations.

Since a complete system includes all of these levels, it was convenient to associate all keys with operations for any one level, with a simple means for changing "levels," i.e., changing en masse the significance of all the operator keys. A small subset of operator keys designated as

"level indicators," analogous to case shift on a typewriter is used to effect level changes. Corresponding to the variety of operator levels, there is a similar diversity in the nature of the "operands." In the context of level 2, for example, they are treated as ordinary numbers, while in the context of level 3 they are, as explained above, functions (represented by their values at 101 points). Each key on the operand keyboard corresponds to a storage allocation in some mass storage device, such as magnetic drum or disc, and data transfers between this and the computer are effected with the operator keys LOAD and STORE.

If A is the label on a particular operand key, then pushing, in sequence, the operator key LOAD and the operand key A brings the data "in" A -- be it a number, a vector, or any other list -- into the computer's magnetic core memory, while pushing the keys STORE A accomplishes the converse.

Experience with this system has shown the on-line approach to be of most help in so-called "fixed point" problems, i.e., those which have the form

$$f = T f$$

where f is an unknown function to be determined (or, equally well, several functions) and T is an operator sufficiently complicated in structure (non-linear, singular, etc.) to cause difficulties in both analytic and numerical approaches. The designation "fixed point" is associated with the concept of f as a "point" in "function space," and T as a mapping of that space onto itself.

If, by any means, one can find an f such that the operation T f reproduces f, one has clearly a solution of the above equation. The on-line system with its graphical display facility and light pen in essence makes it convenient to explore the function space in search of a fixed point. Because of its unusual flexibility, the system facilitates an experimental approach in which the strategy of the search in the function space is varied by the user on the basis of the information he obtains.

Certain shortcomings of the Culler-Fried system have been noted. For example, there is no way for a user to examine the sequence of subroutine steps which constitute the operations performed by a button. The user must therefore document his subroutines with pencil and paper. Future efforts are being initiated to remedy such difficulties.

PART IV. SPECIAL PURPOSE PROGRAMMING SYSTEMS

In this section two special purpose problem-oriented language systems are discussed. Each of these languages and associated computer programs illustrates an approach to problem solving which gains in flexibility and ease of use precisely because the problem areas addressed are extremely narrow and highly specific. Thus STRESS is a problem-oriented language that can be used by structural engineers as an aid in describing the kinds of calculations which must be performed in almost every structural problem. Heretofore, these calculations had to be performed manually, or else described with difficulty in FORTRAN-like languages. The fact that the problem area is highly constrained coupled with the willingness on the part of the designer to sacrifice generality for the sake of ease of use by the engineer enables designers of problem-oriented language systems to provide extremely satisfactory tools for specific classes of users. It is not surprising that, in the light of the extreme specificity of each problem-oriented language, there is little or no carry over from one problem class to another.

A. STRUCTURAL ENGINEERING SYSTEMS SOLVER (STRESS)

STRESS is a special purpose programming system designed as a machine aid to engineers in the solution of structural engineering problems on a digital computer. The STRESS system consists of a special purpose language in which a structural problem can be easily and naturally described. Further, STRESS contains an associated processor, or compiler, which accepts well-formed problem statements in the STRESS language and ultimately produces appropriate solutions.

The STRESS programming system, according to its designers, should not be viewed either as a general-purpose program nor a compiler in the conventional sense. Instead, they note that it combines features of both. Thus, they explain that from the point of view of the user, STRESS resembles a compiler in that it consists of a source language unintelligible to the computer and a processor which translates the source language into machine-coded operations. However, unlike most compilers,

STRESS makes no provision for the compilation of an "object program."

A typical STRESS problem is one in which the nature and size of a particular structure, say, the frame of some building, is specified, along with the loads that the structure is to be subjected to, a statement of the procedure to be followed in the solution, and a statement of the results desired. Modification of any of this information may also be requested in order to obtain additional results for slightly altered problems.

A STRESS input consists of problem-oriented statements using common engineering terminology. In the STRESS language the difference between data statements and procedural statements has been obliterated and the majority of statements contain both data and process directives.

The STRESS input involves a specification of a particular problem. The full specification contains: (1) a header statement which initiates the routine and provides a title; (2) a series of size descriptors which state the size of the problem to be handled in terms of numbers of joint, supports, members and loadings; (3) a statement of the procedure to be used in solving the problem; (4) a set of structural data descriptors within which it is possible to specify necessary information about the geometry, topology, mechanical properties and the presence of local releases (hinges, roller, etc.) of the structure; (5) a set of statements which permit rapid evaluation of alternate designs; and (6) a set of termination statements.

Within the framework of these statements the designers of STRESS have tried to maintain communication in the engineer's language while providin a concise form of input. The result is a relatively easy-to-use special purpose programming language and problem solving system tailor-made to the needs of the structural engineer.

The internal system of STRESS performs four basic functions: (1) input, (2) compilation, (3) execution, and (4) modification. The input phase is initiated by the header statement mentioned above and is terminated by th termination statement, a phrase such as SOLVE, or SOLVE THIS PART. These and all statements in between are presented to the computer (an IB 709/7090/7094 data processing system with 32,768 storage locations) through punched card inputs. The initial phase includes scanning of the

punched cards for the identification of the various parts of the program; that is, the labels and the data in the form of numerical values. This information, once identified, is stored and the parameters specified by the labels are set and other appropriate preliminary steps undertaken. At this point if any portion of the statement is unacceptable, an identifying error message is printed out. The designers have tried to provide as much diagnostic information as possible.

In the second, or compiling phase, all editing, checking and compilation functions are performed. Additional error checks are performed, and if the problem is found to be executable, the remaining compilations are performed.

In the execution phase, a control program essentially calls in and executes the appropriate sub-routines, checks again for errors, and terminates if an error occurs. After execution the system returns again to input phase.

In the modification phase, the input phase is resumed, such resumption being triggered by a key phrase in the initial input statements. In this phase the modification statement is examined to determine the appropriate actions to be performed and these actions are executed with the modified information being superimposed on or merged with the previous data. An unlimited number of modifications are allowed.

From this brief description of STRESS it should be apparent that the STRESS system amounts to an extremely useful and flexible programming language for use in a highly structured problem environment, namely that of structural engineering. The strategy behind the creation of STRESS has been the development of a special purpose language tailored to the needs of a specific user and so constructed as to be easily translated into machine control. The purpose of STRESS is not the enhancing of computer usage through greater generality of programs and program structures however. Rather it is to facilitate the use of digital computers for the solution of structural engineering problems.

B. COGO-90

COGO-90 is the name of a civil engineering problem oriented language and programming system. The basic difference between COGO-90 and STRESS, is essentially in the problem environment to which each addresses

itself. Hence, COGO-90 is used in the solution of geometric problems arising out of land surveying, highway design and bridge geometry, whereas STRESS deals with structural engineering problems. Thus COGO-90, like STRESS, is a useful and flexible programming languag for use in a highly specific problem environment. And its purpose has been, like STRESS, solely to facilitate the use of digital computers in highly specific problem area.

The COGO program currently runs on the IBM 1620 computer, and executes simple macro calls on functions in coordinate geometry. It accepts as input sequences of macro calls with specified parameters. These macros are functions normally used by civil engineers to calcul areas, point coordinates, line lengths, etc., in support of field surveying and similar problems. Any logical sequence of macros may be used. All calculations may be printed out. Some typical COGO functi are illustrated below.

STORE 51 2000.0 5000.0

(Record the point known as "51" as having coordinates x=2000, y=5000).

LOCATE/AZIMUTH 22 51 2236.07 116 34 00

(Record point "22" as being 2236.07 feet from point 51 at an azimuth of 116°34'00". Record and output the coordinates of point 22).

Similarly, angles, areas and other information can be declared or derived from existing data. COGO is designed to be used primarily b engineers working from a plot plan who desire quick answers to one-s surveying problems. COGO is not capable of extensions except by di coding new macro functions. While relatively unsophisticated, COGO a useful and inexpensive engineering tool.

COGO-90 was developed for the IBM 709, 7090 and 7094 computers. is based on the original COGO system developed by C. L. Miller for t IBM 1620. In addition COGO versions have been or are being develop for the IBM 7040, 7070 and 705, the CDC 1600 and 3600, the Burrougl 5000 and the PDP 6.

COGO-90 occupies approximately 20K of core; it contains several hundred FAP instructions and several thousand FORTRAN instructions. The program required approximately 1 man year to develop. However, a large number of the COGO-90 programs were identical to those of the 1620 version. Indeed, it is estimated that, were COGO-90 written without the benefit of 1620 COGO, it would have taken from three to five years to develop.

PART V. TIME-SHARING SYSTEMS

The ability of a system to time-share (i. e., to make a computing facility available to a multiplicity of users simultaneously) is a mode, or aspect, of operation of a system; it is essentially independent of the area of application, or other features of the system and can be achieved in several ways. It is similar, in this respect, to the user-on-line mode of operation and, in fact, most time-sharing systems are also (at least partly) user-on-line systems, though the converse is not gener ally true. Time-sharing systems have been built with widely different properties and features, and the determination of the most suitable set of features for a system must be made without regard to whether or not the system will be made to time-share. By way of indicating the range of computing facilities that have been time-shared, consider the JOSS system discussed in Part III. The facility made available to the user is essentially that of an elaborate, responsive desk calculator. In contrast to this is the MAC system which gives the user access to a comple IBM 7094 which he may use, wisely or poorly.

The main question concerning time-sharing is whether or not it should be done, and if so, how. The matter of the relative efficiency of time-sharing versus batch processing, for example, is still an open issue. However, many types of computing centers, particularly university environments, have found time-sharing to be the best way to make a cor puter available to a large number of users. Time-sharing seems justified, even desirable, where many short, one-shot programs are involve where nearly equal computer time is to be given to each user; and wher no real-time requirements on the system might be compromised. In other cases, it may be preferable to use a more elaborate multi-progra ming scheme with user priorities, or, alternatively, a simple batch processor.

Certain extra hardware (and therefore extra expense) is required in a time-sharing system. For user-on-line operation, users' stations suc as typewriters, display consoles, etc., must be provided; and in genera

sufficient buffering, control, and I/O channel capacity must be available. A prime requirement for time-sharing is fool-proof memory protection, whether it be done by software -- i.e., interpretive execution, as in JOSS -- or hardware, as it must be for an efficient large-scale system. Given this required hardware, almost any computer system can be made to time-share. In fact, simple forms of time-sharing occur as a matter of course in some of the general purpose systems examined in Part VI.

Among the major concerns of the time-sharing designer are the development of proper algorithms for scheduling, the development of routines capable of restoring programs interrupted at some prior stage to their proper place in the computer upon being recalled by the user, the development of sufficiently sophisticated executive routines to minimize the delay time experienced by the user between his input and the system's response. Thus, the time-sharing designer is concerned with generalization of his computer in the sense in which it is available to a large number of users simultaneously. Other considerations of generality, however, such as increasing the dimensions along which man/machine communication is possible, or enhancing the flexibility of program-linkage techniques, or extending the range of problem areas the computer is capable of handling, may also be objectives of time-shared projects. Indeed, many time-sharing system designers believe that the use of time-sharing system techniques is the only way in which tools such as JOSS, SKETCHPAD, and Culler-Fried will ever be economically feasible. Two such projects are described below.

A. PROJECT MAC

The system goal of Project MAC is regarded, by its designers, as the development and operation of a community utility capable of supplying computer power to several users simultaneously with the least number of constraints. Thus, the MAC system is viewed as a public utility which provides each of its users with the equivalent of a private computer whose capacity is adjustable to individual needs.

The primary terminals of the MAC system are 40 model 35 Teletypes and 28 IBM 1050 teletypewriters.

Each can dial, through the M.I.T. private branch exchange, either the IBM 7094 installation of Project MAC, or the similar installation of the

M. I. T. Computation Center. The supervisory program of the two computer installations accepts or rejects the call. Each installation can provide service to as many as 24 simultaneous users.

To provide long distance access, the MAC system is connected to the TELEX network operated by the Western Union Company and will be connected shortly to the TWX network operated by the American Telephone and Telegraph Corporation. The TWX terminal can reach approximately 65,000 Teletypes, and the TELEX network provides access from terminals in Europe as well as in the United States.

The heart of the MAC system is the Compatible Time Sharing System (CTSS), a programming and executive system which allows for multiple user access while also permitting conventional batch-processing loads to be run. This is one of the principal design features of the CTSS. The CTSS includes executive, scheduling, debugging, assembler-compil and input-output facilities. The programming languages presently available in the system for the use of individuals are FAP, FORTRAN, MAD COMIT, LISP, SLIP, a limited version of ALGOL or two problem-orien languages for Civil Engineering, COGO or STRESS.

The system will continue to add new language facilities and other utility programs and programming aids.

The users of the MAC system include faculty and students of a dozen academic departments, and research staffs of five major research laboratories. The disciplines represented range from engineering to psychology, from physics to management, from metallurgy to political sciences.

In a typical programming session at a terminal, the user first logs in, giving his identification. He can then type in a subroutine, perhaps usin the MAD language, and then call for a printout of his input, edit it to correct errors, and call for a MAD compilation. The resulting binary program, possibly with other programs previously compiled, can then b loaded and executed. If the run is unsuccessful the user can request post-mortem data to assist in locating the fault. If necessary he may request the source program and recompile it, perhaps repeating this several times. To terminate the session the user logs out, at which tim he receives from the supervisory program accounting data indicating how much actual computer time he has used. Users' programs and data are

stored in the disc files of the system, together with compilers and other public programs. Thus, a user can interrupt his work when he wishes, and start again where he left off at his next session at a system terminal, hours or weeks later.

The equipment configuration of the MAC computer installation is illustrated in Fig. 1. The IBM 7094 central processor has been modified to operate with two banks of core memory, each consisting of 32K words, and to provide facilities for memory protection and relocation. These features, together with an interrupt clock and a special operating mode (in which input-output operations and certain other instructions result in traps) were necessary to assure successful operation of independent programs coexisting in core memory. One of the memory banks is available to the users' programs; the other is reserved for the supervisory program of the time-sharing system. The second bank was added to avoid imposing severe memory restrictions on users because of the large supervisory program, and to permit use of existing utility programs (compilers, etc.) many of which require all or most of a memory bank.

The basic point with respect to the MAC system in particular, and time-sharing in general is that as a strategy for imparting to a computer a program configuration with a high degree of generality it leaves much to be desired. The construction of its highly efficient and comprehensive supervisory programs merely make it possible for more than one user to have available a large scale conventional computer at the same time. But these programs do not contribute at all in making that computer anything more than conventional. Nor does the time-sharing strategy make available to the user highly sophisticated problem solving techniques. But it was never intended as such. Rather it is a responsive design strategy oriented toward providing a large number of users with simultaneous access to a large scale conventional computer. The development of more sophisticated user-on-line techniques, of general purpose programming and executive routines, of problem-oriented languages, and of information processing techniques remain as problems to be solved within a multiple access, time-shared system.

B. THE SDC TIME-SHARING PROGRAM

The SDC time-sharing program is similar in scope to Project MAC and is designed to make available a tape-oriented AN/FSQ 32v computer to 10-15 users simultaneously, each of whom operates independently under control

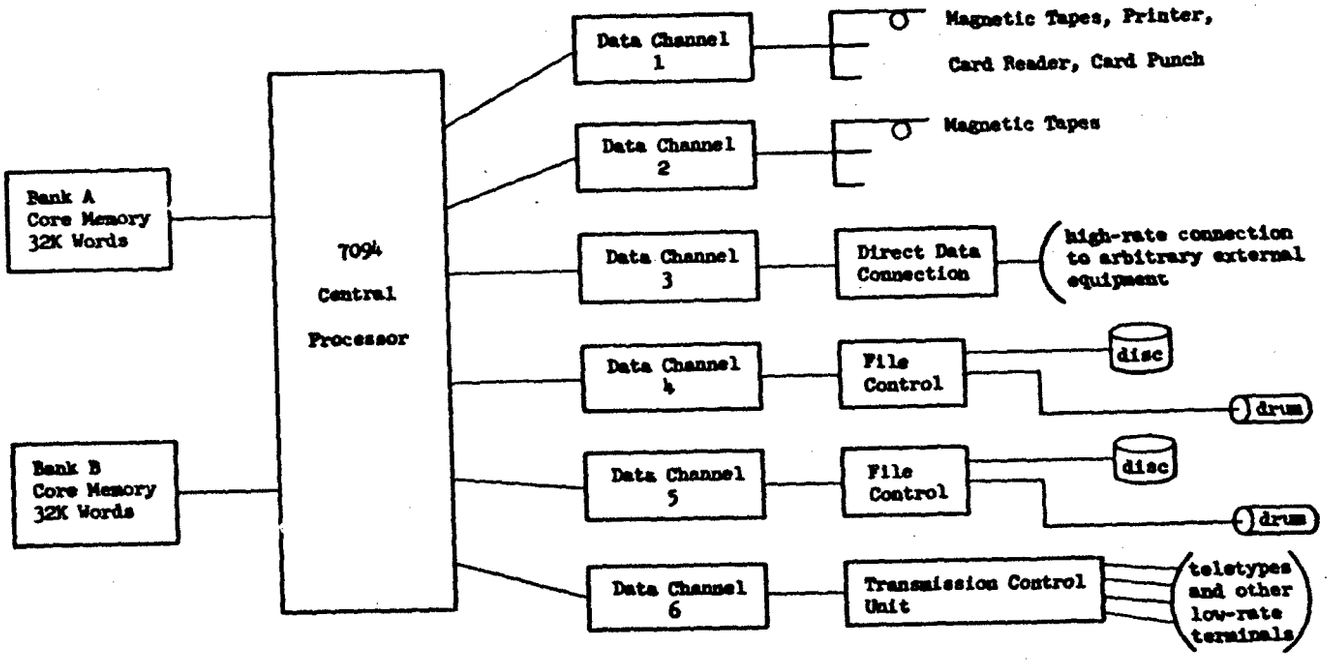

Figure 1. Project MAC Time-Shared Computer Configuration

of an executive system.

The executive system permanently occupies 16,384 words of core memory. It also utilizes around 6,000 words of drum and about 9,000 words of disc permanently for storage of additional programs and data. Input memory is used by the system for buffering between the PDP-1 and the Q-32. A maintenance program (called FIX) occupies an additional 2,000 words of core, 16,000 words of drum, and one tape drive permanently. Also, the remainder of the drum storage is utilized for storage of active object programs, and thus is not available for data use by object programs. The remainder of the computer (core, disc, tapes) is essentially available for the object program's use.

The scheduling algorithm maintains two queues, one for short compute, highly interactive users (determined empirically). These users operate in an essentially round-robin fashion, getting 400 ms. maximum per turn. Since the probability of more than one of these users requiring computation at the same time is quite small, these users generally get almost instantaneous response. The second queue is composed of users who have long compute requirements. They operate with a much longer basic turn, each one being interrupted only by the occasional requirements of the other queue of users, thus making the amount of swap time required fairly minimal. Of the 11,000 executive instructions, the scheduler requires only 500 of these. (The remainder of the executive program's space is occupied by data.)

The programming languages available are JOVIAL, IPL-V, LISP, SLIP, and machine language. The RAND Tablet is now part of the hardware complex.

The difficulties involved in the development of the time-sharing system have been primarily due to the constantly changing hardware and software requirements. Both the program and hardware have maintained a steady rate of change since the beginning of the system, so that although in general, schedules have been met, and use of the system has been quite productive, numerous problems have required a fairly constant work force and level of activity. Since this has been an experimental as well as productive environment, this situation is not unexpected.

The system design represents two major and relatively distinct efforts. First, the basic executive, described above, required six people (generally quite experienced) ever since the early design phase in 1962. The system was operational eighteen months later and has evolved ever since, requiring roughly the same level of six people. In addition to this basic executive, a large number of "service" routines have been programmed and made available to both programmer and non-programme users. These include compilers, editing routines, interpreters, aids to calculation, and other useful routines. In general, each one of these routines required one or two people. The time-sharing project has averaged around six or seven people on this kind of work for two years.

Figure 2. CHARACTERISTICS OF EQUIPMENT IN THE SYSTEM
DEVELOPMENT CORPORATION COMMAND RESEARCH LABORATORY

COMPONENT	NUMBER	CAPACITY/SPEED	TOTAL
AN/FSQ-32 COMPUTER			
Main Core Memory . Cycle time 2.5 μ sec. . 48-bit word	4	16,384 words	65,536 words
Input Core Memory (Buffer) . Cycle time 2.5 μsec.	1	16,384 words	16,384 words
Drum . Access time 11.5 ms. . Word transfer rate 2.75 μsec.	5[1]	139,264 words	417,792 words
Disc File . Access time 225 ms. . Word transfer rate 12 μsec.	16 discs	262,144 words	4,194,304 words
Tape Drives (729-IV)	16	112-1/2 ips	
Card Reader	1	250 cpm	
Card Punch	1	100 cpm	
Printer	1	150 lpm	
Typewriter	2	100 wpm	

[1]Currently 3 units, 2 additional scheduled for 3/65.

Figure 2. (CONTINUED)

COMPONENT	NUMBER	CAPACITY/SPEED	TOTAL
ASSOCIATED COMPUTERS (ON/OFF-LINE)			
PDP-1			
. Shares input core memory of Q-32			32K words
. Cycle time 5 μsec.			
. 18-bit word main core memory	1	4K words	4K words
1401-D			
. Core memory	1	4K char.	4K char.
. Printer	1	600 lpm	
. Tape drives (729-IV)	2[2]	112-1/2 ips	
. Card reader[3]	1	800 cpm	
I/O DEVICES			
Teletypes and Typewriters			
. Model 33 Teletypes	17[4]	100 wpm	
. Model 28 Teletypes	6	100 wpm	
. TWX data sets (remote users)	8	100 wpm	
. Soroban typewriters	3	100 wpm	
. Telex data sets	1	100 wpm	
Display Consoles	6	2K char. max. (per console)	
. Light pens			
. Vector-generator capability			
. Graphic tablet	1	5K points/sec.	
. Keyboard	1	100 wpm	
Telephones			
. Links for simultaneous conversations	6	100 wpm	
. Phones	35		
Recording by PDP-1			

[2] Switchable from Q-32

[3] Scheduled operation 2/65

[4] Capacity increased to 23 by 1/65

PART VI. GENERALIZED DATA MANAGEMENT SYSTEMS

In this section two systems are discussed which represent an approach to software development that can best be described as generalizations of the preceding points of view. These two systems, ADAM and LUCID, have attempted to construct within the confines of one computer system environment programs which incorporate the sophistication and power of general purpose executive routines, the job specific concentration exhibited by the functional systems, the flexibility and power of user-on-line systems, the narrow band specificity of problem oriented languages and, to a lesser extent, the special sense of generality implied by time-sharing. The motivation behind this undertaking is the necessity for devising a computer based system which exhibits the kind of flexibility required of an engineering tool. In this case the goal of the engineer is the design of large scale computer-based systems. Hitherto, he has been without the requisite tool to facilitate his design task. ADAM and LUCID are attempts to fill this gap.

A. ADAM (ADVANCED DATA MANAGEMENT)

ADAM (Advanced Data Management) is a current effort by the MITRE Corporation to develop a system of generalized programs which anticipate most of the features of an on-line, real time multi-access information processing system. The purpose of ADAM is to facilitate the rapid and economic conversion of an IBM 7030 computer and its peripheral typewriters and displays from one alternative system configuration to another. This capability permits alternative design decisions to be tested and verified, e.g., utility of graphical input capability or adequacy of a query language, by the experimenter without incurring the usual reprogramming costs associated with making modifications to conventional computer programs.

The essential feature of ADAM is its generalized implementation of data management functions which are considered to be common housekeeping functions in most systems, i.e.:

(1) Data Base Creation

(2) Data Base Alteration

(3) Data Base Augmentation and Reduction

(4) Data Base Regrouping

(5) Data Base Searching

(6) Data Base Analysis

(7) Input Message Analyses

(8) Output Message Creation

(9) Information Formating

(10) Report Generation

(11) Display Generation

(12) Information Routing

(13) Event Sequencing

(14) Process Implementation

In ADAM, generalized programs for implementing these functions are written without explicit reference to data attributes, message format, display makeup, query language, etc., which are considered the "specifics" of the application. Variation in data base organization, qu language syntax, sequence parameter representation, etc., can be controlled by the experimenter.

The operating ADAM system accepts messages from and sends output on-line input-output devices, such as typewriters, display consoles, printers and magnetic tapes. Messages may query the data contained ADAM; add, change, or delete data; add or delete programs; or cause operation of programs which have been previously added. Data or pr grams added to the system become part of the system, to be written

the next system tape and remain part of the system until replaced or deleted.

Data in ADAM are contained in ADAM files, and the creation, maintenance, processing, and querying of the files are the major purposes of the system. In addition to files of problem data, the system contains a file of routines (both system routines and problem-specific routines), a file of languages in which messages may be stated, a file of formats for outputs, and other special purpose files.

Operations basic to the use of ADAM may be broadly classed into:

File Generation	Creation of new files.
File Processing	Queries, modifications to existing files, computations on file data.
Problem-Specific Processes	Operation of 7030 programs (for file manipulation or calculation) which are specific to a problem as opposed to general-purpose constituents of ADAM.
Program Preparation	Compilation or assembly of 7030 programs for use within ADAM and their insertion into the system.

File Generation, Processing, and Problem-Specific operations all result from message inputs through on-line input devices or the MCP* input tape. Program Preparation requires compilation of programs off-line by the pre-processors or post-processors followed by on-line additions to the routine file.

New input languages or modifications to languages may be made through off-line assembly by the Language Assembly Program followed by on-line additions to the language file.

1. File Generation

The heart of ADAM data management is the ADAM file and the rolls corresponding to each file. All data objects, language specifications, routines, formats, etc., are carried in files. A file is a named collection of like

*ADAM operates in conjunction with the Master Control Program (MCP) for the IBM 7030.

things. For example the "airfield" file would contain information about several specific airfields. Each airfield, e.g., Logan, Hanscom, etc., would be an object in the airfield file. The various data about each object are called properties. Typical properties for the airfield file would be Location, Number of Runways, Elevation, etc. The actual data in a file are the values of these properties, e.g., Boston, 6, 2,0 feet, etc. The set of properties for an object in a file must be the same set as for all other objects in that file. Of course, the values of these properties may change from object to object. The set of all property values for one object in a file is called an entry in that file. ADAM files are structured serially by entry, the first entry being the values of the properties that occur once per file, e.g., file name, file size, date updated, etc.

A roll is a type of dictionary or directory that associates external name with their more concise internal representations, usually some small integer. This integer is called the principal value of the name. Synony for external names are effected by a many to one mapping of names to integers, i.e., all synonymous external names have the same principal value. Additionally, a roll will have subsidiary values which give additional information about the things named in the roll. Within a roll, principal values are unique.

Each file has associated with it, but physically separate, an object roll and a property roll. The object roll contains the names and principal values of each object in the file and subvalues which give the object's location in that file. The property roll has the names and principal val of the properties in the file, and for each property, several subvalues giving type, size, unit of measure, location, legal range of values, etc Files may also use rolls to hold the alphanumeric values of some of the file properties (e.g., red, green, up, north, or some other non-numer value) and carry only the principal values of these values in the file its Many files may share such a roll. Rolls are not used only by files; the system keeps several rolls for its own internal use.

Property types are fixed-length (integer, floating, small-range or low precision floating, or alphanumeric) and variable-length. Examples of the latter are query-valued (some processing is invoked to produce the value), raw (arbitrary string of bits such as an actual alphanumeric string as opposed to a principal value standing for such a string), and repeating-group. A property of the repeating-group type is simply a

collection of sub-properties, which may themselves be of repeating-group type nested to arbitrary depth.

Any data amenable to structuring may be introduced into ADAM. New files may be generated from external data, existing ADAM files, or both.

2. Queries and File Processing

The principal feature of ADAM which it shares with no other generalized data management system is its translator.

The translator translates a message in accordance with a language, selecte from a number of languages contained in the language file. The selection is based on a language specification determined by the recognition rules. Translation involves transforming the syntax of the message into a specification of operations to be performed and transforming the names of files, etc., in the message from their external form to a condensed internal representation. Translation of the syntax occurs with reference to the specified language. Translation of names requires directories of names or rolls and are used as required by the translator. The translated message becomes a process table, which is a set of pseudo instructions which specify various subroutines that effect the processing specified by the message. The process table may call for combinations of file processing and problem programs in essentially any order. The processor interprets the process table by examining each pseudo instruction in order and calling upon the appropriate subroutines to perform the action.

This function of the ADAM system permits several potential users to specify the syntax of a query language relevant to their own problem area. The sentences generated by each of their respective grammars, however, will be acceptable to the ADAM translator without the necessity for its being reprogrammed. In different words, the ADAM translator is a syntax driven compiler which provides the same translation capability to any number of different users without reprogramming. Thus, the ADAM translator may also be viewed as a query-language modelling facility which allows for the study of query languages without building software prototypes Such facilities as these exemplify what is general in generalized data management systems.

3. Routines

An ADAM routine is a piece of code which obeys ADAM rules for comm cating between routines and data. Routines are prepared in DAMSEL (a procedural language available in ADAM) or FORTRAN. In either case, some off-line processing is required to produce a binary form of t routine with its communication information included. This binary routin is what is then inserted into the ADAM system. Theoretically, routines and their communication links may be directly prepared also via the SMAC and STRAP assemblers. Routines are all relocatable and are size limited only by the amount of available core when they are to run. All ADAM routines are kept in a special file called the "routine file."

4. System Control

The system control philosophy of ADAM is to accept input messages as they arrive, recognize them, and place them according to some priority scheme in a job queue. An ADAM message is any string of characters from an input device that ADAM can recognize. ADAM will recognize a input language whose recognition rules are given to the system. When a running job is completed, the top of the job queue is examined and the appropriate routine is called (problem program, translator, query analyzer, etc.) and enjoined. When the job is started, it runs to comple tion, being interrupted only for recognition and stacking of input messag This implies that only one program (other than utility routines) can have the machine at one time. The amount of multi-programmin g of user programs is minimal. Requests for output are handled immediately if the channel and peripheral gear are available; otherwise, they are queu up and sent out when possible. All system routines (input/output handling, job scheduling, memory allocation, etc.) are subroutines and may call each other as needed.

Streams are an artifice whereby any ADAM data structure can be considered of arbitrary size, existing over more than one storage media (e.g., part in core, overflowing onto disk). Two routines are provided for the handling of streams. Other control routines available automatically to the system are a disk and tape allocator, core allocator, routine loader, and roll controller.

Thus, ADAM is a system which performs data management functions by general techniques independent of the form of data or calculations requi

in any specific application. ADAM treats as data the specifications of a specific system design. Through its operator language, designers are able to dynamically modify system specifications resulting in a convenient evolution in design. Furthermore, because the designer has access to current system specifications in ADAM, ADAM itself can be used to prepare reports on the design and can generate on-line displays for assisting users to both use and modify the system.

B. LANGUAGE USED TO COMMUNICATE INFORMATION SYSTEM DESIGN (LUCID)

LUCID is a user-oriented language and macro system for the design and evaluation of data retrieval and manipulation systems in a laboratory environment. The system is being written for the AN/FSQ-32 computer at the System Development Corporation, Santa Monica, California. The objectives of this system are to provide the information system designer with an integrated set of closed JOVIAL procedures and a convenient means for arranging them in appropriate complex hierarchies under a control program to provide data base maintenance and on-line data-base query capabilities.

The LUCID system is designed to operate from a Master Tape. In the Phase I version of the system there are four system files on the Master Tape. These files, (1) System Control, (2) LUCID Translator, (3) On-Line Query System, and (4) Interpreter, contain not only the appropriate LUCID system routines for performing designated functions, but in addition, they contain tables which are a complete parametric description of the user's system under investigation. Because LUCID programs are highly generalized, they must be supplied with certain parameters by the computer center staff even before the system can be employed as a design tool. Thus, the System Control program provides an initial parameter load capability which specifies input media, output media, available tapes, symbolic representations of LUCID operations, comments to be communicated to the designer, etc. Once the system is initialized, experimental file structures, input data, temporary files and processes for the manipulation of the data can be easily described in the LUCID language by the experimenter. These descriptions are translated by the LUCID translator into binary tabular form called OPAQUE. The OPAQUE tables are then examined by the Interpreter which calls in the necessary macro routines for building and processing files, thus implementing a specific system design.

The most salient characteristic of the internal structure and design philosophy of the LUCID system is the number of tables used. In addition to the OPAQUE tables, others such as communication tables, control tables, temporary storage tables, and a table of operators are generated automatically by the system.

LUCID admits six levels of data structure: data base, file, table entry, item, and bead. Four structures of data tables (linked, serial, parallel, and mixed) can also be accepted by the system. A linked table, not accessible to the user in the present system, is a table in which each entry provides the index for the next entry. Serial tables are serial by row and parallel tables are serial by column. The mixed table consist of two parts, a fixed-entry-length part and a variable-entry-length part. Each fixed-entry component contains a cross reference to the variable-entry part.

LUCID also has provisions for the generation and maintenance of a concordance of the data base which is, in effect, a copy of the data base with the contents listed in a different way. Some but not necessarily all of the property values are used in a normal manner. The size of the concordance varies from slightly smaller than to three or four times the size of the data base. This type of organization facilitates certain retrieval operations, but can be very inefficient.

Current capabilities include off-line file generation and system description with card-to-tape, tape-to-card being the conventional input/output mode of operation.

CONCLUSIONS

From this survey on important perspectives in advanced software development, it is apparent that several new and powerful entries have been made in the roster of software innovations. Thus, the general purpose programming and executive systems cited in this survey have contributed significantly to the notion of the "extensible" machine where storage allocation, program linkage, program segmentation, sequence control, error detection and correction and I/O control capabilities are provided as tools for building, controlling and modifying programs. Functional systems such as COLINGO, NAVCOSSACT, FICEUR and 473L have incorporated new techniques for retrieving data from large numbers of formatted files and for manipulating these files in an efficient manner. In the area of man-machine interface techniques, SKETCHPAD, CULLER-FRIED, JOSS, and BASEBALL have each contributed new ways for non-programmers to interact with computers through the development of graphical manipulation, calculation and natural English accessing capabilities, respectively. Similarly, COGO and STRESS have shown where languages tailored to specific problem classes promise an order of magnitude improvement in the problem solving ability of non-computer specialists. Finally, generalized programming techniques in the form of data management systems promise significant improvement in the flexibility of multi-user, on-line, computer-based systems.

The desirable features found in all of these systems are summarized below:

(1) Addressing the computer in a specific problem area (COGO, Sketchpad, etc.)

(2) On-line users controlling their computational processes.

(3) Referencing data objects by name.

(4) Description of data objects independently of the program

(5) Use of programming and retrieval languages.

(6) File maintenance procedures.

(7) Scheduling and monitoring functions.

(8) Creation, modification and evaluation (running) of programs and data.

(9) Time and facility sharing.

(10) Automatic allocation of machine resources.

(11) Specification of parallel paths of processing.

(12) Extensibility into various problem areas.

All of these features contribute to more effective use of computers and may certainly all be present within a given system. Indeed, many of these features exist at varying levels of use and sophistication in many of the systems that have been examined. For example, both COGO and ADAM have an executive control; one is primitive and special-purpose, other quite sophisticated and general. It is also true that the various features are achieved at different costs in different systems, e.g., it would be cheaper to include an ALGOL translator under the CL-II system than it would be under COLINGO .

Given that the features in the above list have proved to be valuable, individually and in various combinations, the approach of the designer of "next generation" systems might profitably be the following: rather than concentrating on a specific application, and building up from scratch a computer system to meet those requirements, he should start by designing a hardware-software sub-system which provides, or facilitates the provision of all (or a large subset) of these desirable modes of operation, programming environments, etc. If this can be done in a very general, and reasonably efficient, fashion, the designer will have at hand the beginnings of any number of different systems, each of which will be relatively easy to modify and expand, as requirements dictate.

In the light of this observation, it is important to note that to date effort to develop generalized systems and sub-systems exist only on a very

small scale and in only a small number of laboratories scattered across the country. As a consequence, there does not now exist a single system in which all or many of the techniques mentioned above are being integrated into one efficient configuration. And this is in spite of the fact that computer systems so designed would be operational tools of inestimable value to the military in general, and to the Air Force specifically.

And this brings us to the final observation of this survey: namely, the time is now ripe for a technical program designed to demonstrate the feasibility of generalized computer systems and sub-systems in the solving of traditional data processing problems.

BIBLIOGRAPHY

I. GENERAL PURPOSE PROGRAMMING AND EXECUTIVE SYSTEMS

AOSP (Automatic Operating and Scheduling Program)

Anderson, Hoffman, Shifman and Williams, "The D-825 - A Multi-computer System for Command and Control," Burroughs D&S Group, Proceedings FJCC, 1962.

Thompson and Wilkinson, "D825 Automatic Operating and Scheduling Program," Burroughs, Proceedings SJCC, 1963.

CL-II.

T. E. Cheatham, Jr., G. F. Leonard, "An Introduction to the CL-II Programming System," Computer Associates, Inc.

DODDAC (Department of Defense Damage Assessment Center)

Defense Atomic Support Agency, Department of Defense Damage Assessment Center, Initial System, Executive System Program Description, Technical Memorandum No. TM-WD-16/001/00, System Development Corporation, Santa Monica, California, 1 October 1964.

Defense Atomic Support Agency, Department of Defense Damage Assessment Center, Initial System, Glossary and Program Index, Technical Memorandum No. TM-WD-14/002/00, System Development Corporation, Santa Monica, California, 1 December 1962.

Defense Atomic Support Agency, Department of Defense Damage Assessment Center, Initial System, System Data Definitions, Technical Memorandum No. TM-WD-16/005/00, System Development Corporation, Santa Monica, California, 1 December 1962.

Defense Atomic Support Agency, Department of Defense Damage Assessment Center, Initial System, Data Base System User's Manual, Technical Memorandum No. TM-WD-16/007/00, System Development Corporation, Santa Monica, California, 31 Dec 1962

INTIPS (Integrated Information Processing System)

ECP 1A User's Reference Manual, Informatics, Inc., Sherman Oaks, California, 30 August 1964.

Programming Manual (RW-400) AN/FSQ-27, 2nd ed., prepared by Bert Helfinstein, Ramo-Wooldridge Division, Thompson-Ramo-Wooldridge, Inc., Canoga Park, California, 1 February 1961.

OASIS

"Support System - 1604 OASIS Program Descriptions," System Development Corporation, TM-WD-17/001/00, 1 August 1962.

II. FUNCTIONAL SYSTEMS

ACSI-MATIC

Colilla and Sams, "Information Structure for Processing and Retrieving," RCA, Comm. ACM, January 1962.

Holt, A. W., "Program Organization and Record Keeping for Dynamic Storage Allocation," ADR, Comm. ACM, October 1961.

Gurk and Minker, "The Design and Simulation of an Information Processing System," RCA, Journal ACM, April 1961.

Miller, Minker, Reed, and Shindle, "A Multi-Level File Structure for Information Processing," RCA, Proceedings WJCC, 1960.

COLINGO (Compile On-Line and Go)

"COLINGO 'D' a control language for U. S. STRIKE COMMAND," MITRE Working Paper W-06890, 4 March 1964.

"USSTRICOM COLINGO 'D' SYSTEM DESCRIPTION," MITRE Working Paper W-06821, 22 January 1964.

FICEUR (Fleet Intelligence Center Europe)

"Intelligence Data Processing System, Formatted File System Volume 4, Information System Design and Utilization," Prepa by U. S. Navy Fleet Intelligence Center, Europe, and Internat Business Machines, Rockville, Maryland, May 1963.

"Intelligence Data Processing System, Formatted File System Volume 5, Information Retrieval," Prepared by U. S. Navy Fleet Intelligence Center, Europe and International Business Machines Corporation, Rockville, Maryland, May 1963.

NAVCOSSACT (Naval Command Support System Activity)

"User's Manual for NAVCOSSACT Information Processing Sys Phase I," NAVCOSSACT Report No. 51, Naval Command Sys Support Activity, Prepared by IBM, 12 July 1963.

"Supplement 1 to User's Manual for NAVCOSSACT Informatior Processing System - Phase I," Supplement 1 to NAVCOSSAC Report No. 51, Naval Command Systems Support Activity, Prepared by IBM, 22 January 1964.

"User's Manual for NAVCOSSACT Information Processing Sys Phase I, Library Maintenance System," NAVCOSSACT Repor 52, Naval Command Systems Support Activity, Prepared by I national Business Machines Corporation, 23 August 1963.

"User's Manual for Information Processing System for the AN/FYK-1(V) Data Processing Set - Phase II," NAVCOSSAC Report No. 123, Naval Command Systems Support Activity, Prepared by IBM, 15 June 1964.

"User's Manual for the 704/7090 Information Retrieval," NAVCOSSACT Report No. 76, Naval Command Systems Suppc Activity, Prepared by Suzanne J. Berard.

TUFF - TUG (Tape Updater for Formatted Files - Format Table Tape Updater and Generator)

"User's Manual for 704/7090 TUFF MOD III, Tape Updater for Formatted Files," NAVCOSSACT Report No. 74, Naval Command Systems Activity, Prepared by Marie P. Wilk, 1 November 1963.

"704/7090 TUFF (Tape Updater for Formatted Files), On-Line Messages and Error Stops," NAVCOSSACT Report No. 75, Naval Command Systems Support Activity, November 1963.

"User's Manual for TUG - Format Table Tape Updater and Generator," Naval Command Systems Support Activity, Prepared by International Business Machines Corporation, Rockville, Maryland, 30 October 1962.

473L (US Air Force Command and Control System)

Technical Memorandum for IOC Optimization 473L IOC Phase, Technical Memorandum No. TMV-225, Prepared for 473L System Program Office by International Business Machines Corporation, Rockville, Maryland, 29 March 1963.

Operational Specification for Query Language - 473L, Operational Specification No. 473L-OS-10, Prepared for 473L System Program Office by International Business Machines Corporation, Rockville Maryland, 25 May 1964.

III. MAN-MACHINE INTERFACE SYSTEM

BASEBALL

Green, B. V., Wolf, A. K., Chomsky, C., and Laughery, Jr., "Baseball: An Automatic Question-Answerer," Proc. WESCON Vol. 19, 1961.

Wolf, A.K., Chomsky, C., and Green, Jr., B. F., "The Baseball Program: An Automatic Question-Answerer, Volume I," Technical Report No. 306, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, Mass., 11 April 1963.

CULLER-FRIED

Culler, Glen, and Fried, Burton D., "The TRW Two-Station On-Line Scientific Computer," TRW 8587-6002, RU-000, TRW Space Technology Laboratories, Redondo Beach, California, July 1964.

Culler, Glen, and Fried, Burton D., "An On-Line Computer Center for Scientific Problems," M19-3V3, Thompson-Ramo-Wooldridge, Inc., Canoga Park, California, June 1963.

DEACON (Direct English Access and Control)

Thompson, Frederick B., J. A. Craig, Gregory D. Gibbons, John W. Gwynn and Jacque S. Pruett, "DEACON Breadboard Summary," RM64TMP-9, TEMPO, General Electric Company Santa Barbara, California, March 1964.

Shaw, J. C., "The JOSS System, Time-Sharing at Rand," Datamation, November 1964.

SKETCHPAD

Sutherland, I. E., "Sketchpad: A Man-Machine Communicatio System," Technical Report No. 296, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, Massachuse 30 January 1963.

IV. SPECIAL PURPOSE SYSTEMS

COGO

Miller, C. L. "COGO - A Computer Programming System for Civil Engineering Problems," M. I. T., 15 August 1961.

Roos, Daniel and C. L. Miller, "COGO-90: Engineering User Manual," Department of Civil Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1964

Roos, Daniel, and C. L. Miller, "COGO-90 Time-Sharing Ver Department of Civil Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1964.

STRESS

"Stress: A User's Manual," Dept. of Civil Engineering, Massachusetts Institute of Technology, M. I. T. Press, 1964.

V. TIME SHARING SYSTEMS

MAC (Multi-Access Computer)

"The Compatible Time-Sharing System," M. I. T. Computation Center, Massachusetts Institute of Technology, 1963.

SDC Time-Sharing Program (System Development Corporation)

Schwartz, Jules, "Introduction to the SDC Time-Sharing System," SP-1722, System Development Corporation, Santa Monica, California, August 1964.

VI. GENERAL DATA MANAGEMENT SYSTEMS

ADAM (Automated Data Management)

Burrows, J. H., "MITRE Presentation on Automated Data Management," MITRE Corp., TM-03905, 15 February 1964.

Connors, T. L., "A Brief View of ADAM," MITRE Corp. Working Paper, 1 April 1964.

Hodgins, D. D., "General Description of: 1. Structures of Data and Programs in the ADAM System; 2. Functions of Individual Programs in the System," MITRE Working Paper, 28 August 1963

LUCID (Language Used to Communicate Information System Desig

"The LUCID System of Automatic Programming Directly from Dat Processing System Design Specifications," Working Paper No. FN-6797/000/00, System Development Corporation, Santa Monica California, 4 August 1962.

System Design Specifications for LUCID Phase I," Tech Memo No. TM-1749/000/00, System Development Corporation, Santa Monica, California, 27 January 1964.

"Volume I. LUCID Control System Design, Part 1. The Master Tape," Tech Memo No. TM-1749/101/00, System Development Corporation, Santa Monica, California, 27 January 1964.

"Volume I. LUCID System Control Design, Part 2. Parameter Load," Tech Memo No. TM-1749/102/00, System Development Corporation, Santa Monica, California,27 January, 1964.

"Volume 1, LUCID Control System Design, Part 3 Operational Control," Tech Memo No. TM-1749/103/00, System Developme Corporation, Santa Monica, California, 27 January 1964.

"Volume I. LUCID System Control Design, Part 4. Test Set-U Tech Memo No. TM-1749/104/00, System Development Corpora Santa Monica, California, 27 January 1964.

"Volume II. GENDARME Data Processing Facilities," Tech Me No. TM-1749/201/00, System Development Corporation, Santa Monica, California, 27 January 1964.

"Volume III. LUCID Program Design: The Grammar of OPAQU Tech Memo No. TM-1749/301/00, System Development Corpora Santa Monica, California, 27 January 1964.

"Volume IV. LUCID Translator Design," Tech Memo No. TM-1749/401/00, System Development Corporation, Santa Monica, California, 27 January 1964.

"Volume V: LUCID Utility," Tech Memo No. TM-1749/501/00. System Development Corporation, Santa Monica, California, 27 January 1964.

For general remarks on system design, see:

"A Multiprocessor System Design," M. E. Conway, Vol. 24, AFIPS Conference Proceedings.

"An Environment for an Operating System," G. F. Leonard, J. R. Goodroe, ACM, Proceedings of the 19th National Conference.

"Hardware/Software Interaction," E. L. Glaser, Burroughs Corporation, TR62-45.

"Design Concepts for a Programming System to Support a Large Scale Operations Center," T. E. Cheatham, Jr., G. O. Collins, Jr., Computer Associates, Inc., CA-61-1.

"Exercise and Evaluation of Command and Control Systems," T. E. Cheatham, Jr., G. F. Leonard, Computer Associates, Inc., CA-61-2.

"Evaluation of Generalized Data Systems for Use in Command and Control," W. R. Slack, Computer Associates, Inc., Final Report, Contract No. AF19(628)-4160, Electronic Systems Division, Bedford, Mass.

Information concerning all systems not included in the Bibliography was obtained by letter or personal communication.