material to which they pertain. For example, table handling and subscripting should be discussed along with the OCCURS clause in the Data Division.

In brief the main topic headings for a four-week COBOL course could be the following:

FIRST WEEK. (1) Equipment Specifications, (2) Input-Output Instructions, (3) Data-Handling Instructions, (4) Special machine considerations, (5) Sample problem (Machine level).

SECOND AND THIRD WEEKS. (1) COBOL: (a) Environment Division, (b) Data Division, (c) Procedure Division, (d) Sample problem; (2) Basic COBOL options, (3) Sample problem.

FOURTH WEEK. (1) More advanced COBOL Options, (2) Programming logic, (3) Techniques, (4) Sample problem.

At the conclusion of the COBOL programming course the student should be familiar with the COBOL compiler and the relationship between source and object program. He should understand the importance of the environment and data descriptions in obtaining an efficient object program.

The trainee should be periodically evaluated during the first six months after completing the formal course. This can be done by grading home study assignments supplied by the manufacturer. In addition, it is also incumbent on the manufacturer to supply his users with programming aid publications and course materials which give the programmer the tools necessary for continued study in techniques and applications.

In summary, a well conducted four-week COBOL programming course should enable the graduate to contribute immediately to the company's programming efforts.

An Advanced Input-Output System for a COBOL Compiler

C. A. Bouman
Radio Corporation of America, Cherry Hill, N.J.

File Control Processor

COBOL, and business-oriented computing languages in general, have had a very salutary effect on Input-Output systems. The very-machine-oriented or application-oriented input-output system has been thrown to the wind and the truly-problem-oriented system has taken its place. Reading at the logical record rather than the machine block level has made what always has been a common input-output requirement of business-oriented problems more fully recognized. Need for object time efficiency has been realized and met more fully from understanding of the total problem.

We at RCA resolved to get full value from our COBOL input-output system for our 601 computer. We had secured valuable insight into the needs of an English language input-output system from implementation of the 501 COBOL Narrator compiler. We wished to retain all of the desirable features of the 501 I-O system, yet expand greatly on the efficiency and the range of the new system. We elected to name this I-O system the File Control Processor.

Fundamental Objectives

Some of our fundamental objectives were:

1. Minimum object time memory.
2. Maximum object time speed.
3. Ability to use the system for *implementation* of 601 COBOL as well as a system for 601 COBOL object programs.
4. Ability to incorporate the system into the 601 Assembly System to enable the Assembly System to produce programs using the File Control Processor when the COBOL verbs OPEN, READ, WRITE and CLOSE were encountered in the program being assembled.
5. Ability to use the system for implementation of the 601 Assembly System.
6. Ability to implement all types of batching.
7. Ability to open any file in any direction at any time.

One thing was clear. To achieve all of these objectives, it would be necessary to have an interpretive system, i.e., a system of subroutines which would execute logical reading or writing regardless of which file was being manipulated. This presented problems of object time speed to overcome which we needed to develop new features, such as the Read Service and the Master Queuing techniques which are described below. Use of the interpretive mode also presented problems of object time memory. To overcome this, the old technique of segmentation was used.

One reason that it was desirable to have the program interpretive was that the program would need to be implemented quickly so that it would be available for use in building the Assembly System and the COBOL Compiler.

A second and very important reason for having the program interpretive was because of segmentation. When segmentation is used, there is no gain achieved when reducing the size of any but the largest segment occupying a common memory area. If generative techniques were used, a separate READ and/or WRITE routine would exist for each file. This would require that the amount of

memory involved could not be allocated until object time. This prevents a balance from being achieved between the sizes of separate segments and can, in the long run, require as much as or more memory than interpretive techniques.

Fundamental Object Time Functions

The fundamental object time functions required of an I-O system are:

1. OPEN FILE
2. READ FILE
3. WRITE FILE
4. CLOSE FILE
5. END REEL-INPUT (tape swap, etc.)
6. END REEL-OUTPUT

Frequency of execution of opening and closing of files and end-of-reel procedures is extremely low for most applications. When these functions are performed, it is desirable to trade time for memory.

Assume a case where four files are to be opened, the same four files are to be closed, and two END REEL procedures are to be executed (the Master file is two reels long so that one END REEL-INPUT procedure and one END REEL-OUTPUT procedure is required). This case would require a total of ten accesses to these routines. These routines, on the other hand, require an aggregate of approximately 1000 words of memory. By calling in these routines from the library tapes each time they are required (segmentation), these 1000 words of memory can be saved. This can be done at a cost of approximately 60/1000 of a second (less in the case of successive calls) per call on the 601. The total time required for the example presented where a total of ten accessed were required would be $\frac{6}{10}$ of a second for the run. This tradeoff is very desirable and has been made in the 601 I-O system with great savings.

There are other reasons why segmentation of these functions is desirable. Once these subsidiary functions are segmented, it is of little significance how large they are as long as they are smaller than the READ-WRITE segment. The READ-WRITE segment is always in memory except when opening, closing, etc. are being done. This allows more space in the auxiliary segments for READ servicing, complete operator instructions and error messages.

Minimization of Object Time

The COBOL language can be considered as the language of a theoretical computer. A COBOL compiler bridges the gap between this theoretical computer and an actual one. Each verb of this theoretical computer has, however, an object-time operating speed depending on the compiler and computer for which the program is compiled. The state of the art in compiler design and demands on object-program efficiency have increased greatly in the immediate past and will continue to increase in the immediate future. The I-O system for the 601 attempts to minimize the amount of object time required, not only by efficient object-time logic, but also by conceptual advances which can be used at the system level. Some of these advanced techniques are described here.

1. *Read Servicing.* When files are opened and closed, much work can be executed which would otherwise need to be executed whenever each read occurred. This results in significant reductions in object running time. One type of READ servicing used in the 601 System is READ-WRITE multipath switches. These switches are set in the parameters for a file which will direct it to the correct logical READ or WRITE routine or an EXCEPTION routine, depending on its state when used for reading or writing. Explicitly, there are two switches—one for reading and one for writing. A scematic is given in Figure 1.

Prior to opening a file, the switches in the file parameters for that file are set to EXCEPTION so that if either a READ or a WRITE occurred prior to opening, control would be transferred to an EXCEPTION routine which would print out the error condition. When opening a file as input, the READ switch is set to the batching routine called for by file parameters. The WRITE switch remains set to EXCEPTION so that attempts to execute logical *writes* when a file is opened as input will result in object-time error printouts. When an *optional* file is opened as input and the file is not present, the READ switch is also set to the EXCEPTION routine. This technique eliminates all exception case burden time. It also allows any file to be opened as input forward, input reverse, or output at any given time.

2. *Reading Reverse.* Input files, in 601 COBOL, may be read either forward or reverse. When a file is opened without the reversed option, subsequent reads will supply the next logical record starting at the beginning of the file. When a file is opened *with* the reversed option, subsequent reads supply the *previous* logical record starting at the end of the file.

3. *Reopening Files.* Consistent with implicit COBOL definitions, files in 601 COBOL may be opened in any manner, closed and reopened in any manner. A file may be opened as output, for example, written, closed and reopened as either input forward or input reverse. A second example for its use would be where a file must be scanned

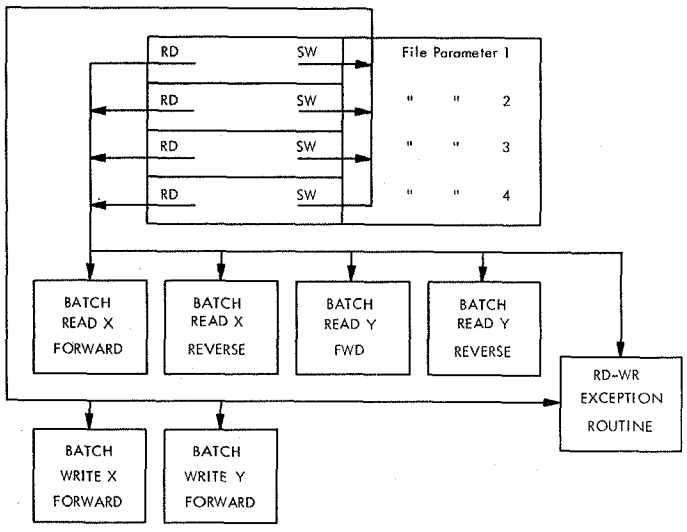

FIG. 1

twice. In this case it could be opened forward, read, closed, then opened reverse, read and closed. A third example would be where an input file is later used as an output, as in a sort. This eliminates redescribing the file and also conserves memory.

4. *Multi-file Reels.* Many times it is quite desirable to have one tape contain more than one file of information. A multi-file reel may take the form of files going to off-line devices, files used for intermediate results between runs, small reference files, or combinations of the three. This is a feature which, in many cases, has not been fully exploited in previous compilers. It has its most significant value where a mutiple run system is employed. An inventory system, for example, may have a number of files to be printed or punched after all runs for the system are completed. It may be desirable, on a system of this sort, to save the original transaction file. Where this is the case, files containing intermediate or final results can be described as second or successive files on the back of this same reel of tapes. This reduces the amount of tape loading and unloading and reduces the number of tape drives required.

5. *Master Queuing Techniques.* When it is determined that a machine buffer has become available, it is necessary to determine whether an alternate area is available which can accept a new physical block from an external medium or an alternate area is full which can be written to an external medium. Repeated testing of these conditions for each file in the program whenever a machine buffer becomes available is normally quite costly in machine time and takes away from the gain achieved by keeping input-output devices moving. An advanced technique for programmed scanning using jump switches has been developed for the 601 system which eliminates approximately 90 percent of this scanning time. A schematic of this is given in Figure 2.

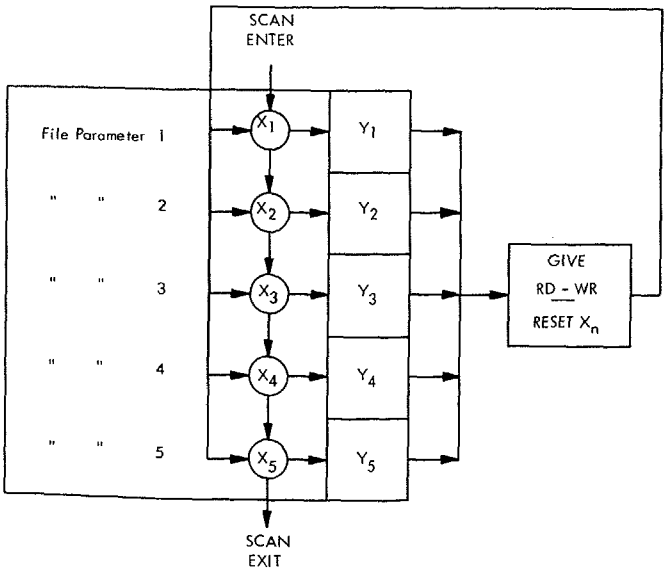

FIG. 2

Whenever a machine buffer becomes available, transfer of control is given to SCAN ENTER. Switch X_1 contains a jump to X_2 if no alternate area is available. If an alternate area is available, X_1 contains a jump to Y_1. Y_1 stores X_1 as an exit to the routine which gives the physical READ or WRITE instruction and gives control to the physical READ-WRITE routine. The Physical READ-WRITE routine, after giving the instruction and resetting the switch to "no alternate area available" transfers control back to X_1. This reduces time on the 601 from 37μsec to 3μsec per file scanned. This is very significant in large programs where many files are frequently described.

6. *File Priorities.* Many production runs of COBOL object programs feature processing against reference or master files. Where this is true, it is desirable to have the master file read and written at maximum speed where at all possible. Subsidiary files such as transaction files can then be sandwiched between master file reads and writes when the program becomes compute bound, when the balance between reading and writing of the master file is not uniform, or where a secondary machine input-output channel is available. The 601 COBOL input-output system provides this feature.

7. *Address Modifier Assignment.* Minimum data movement in the 601 I-O system is allowed through Address Modifier Assignment. Address Modifiers allow logical records to be accessed in the area where they are read into from the external media or from where they are to be written out to an external medium (alternate areas). It is a means by which the logical record areas for input and output files are "floated". A schematic of this is given in Figures 3 and 4.

When no address modifiers are assigned to an input file, the record is moved by the I-O system to a logical record area designated for that file each time a READ verb is executed for a COBOL program.

When an address modifier is assigned to an input file, no movement of data takes place on the logical READ. Instead, the address modifier assigned is updated to contain the addresses of the left-hand-end and the right-hand-end of the record area by the I-O system. The record is then accessed directly from the read-in area using the address modifier assigned. This technique eliminates moving data except for the transfer from input to output record areas.

It is important to note, regarding this technique, that one data movement is always required to control input-output for regrouping of batched files regardless of machine features involved. Optional assignment of address modifiers requires that a machine feature be assigned for this purpose only where the size of the file and/or the amount of other computing time involved make this assignment desirable.

8. *Variable Length Batching Techniques.* (a) Variable Length Records—Significant advantages in tape passing time can be achieved through elimination of leftmost zeros of numeric fields and rightmost spaces of alphabetic and

alphanumeric fields. This is accomplished in the RCA 601 computer, as in other RCA computers, by machine-oriented features which make handling of these fields efficient. Records of this type are termed *Variable Length Records.*

(b) Mixed Record Files—Handling of files containing variable length records is a separate problem, however, from handling of the records containing variable length fields. When any record within a file can be larger or smaller than any other regardless of whether any specific type of record can itself vary in size, the input-output system must do the work. Files of this nature are usually termed *Mixed Record Files*, or more simply, *Mixed Files.*

(c) Mixed Record File Batching—The fundamental problem in files of this nature is batching. If two types of fixed length records are within a file and one record is 100 characters in length while the other is 1000 characters in length, it would be very inefficient to specify batching as two records per batch. The inefficiency arises because it is not known in advance which mix of records is to occupy a batch. Because of this, memory would need to be allocated for the worst case—two records of 1000 characters each for a total of 2000 characters. If two successive records of 100 characters each were to be output, the batch size would be 200 characters and memory space, tape space and tape time would be wasted.

Efficient methods have been developed for the 601 input-output system which will pack these records into a batch area until there is room for no more in the area. Twenty records of 100 characters each or two records of 1000 characters each, or a combination of both, for example, can be packed automatically by the input-output system without requiring the length of the record within the record itself. This is, of course, in addition to almost every other conceivable type of batching.

NO ADDRESS MODIFIER ASSIGNED

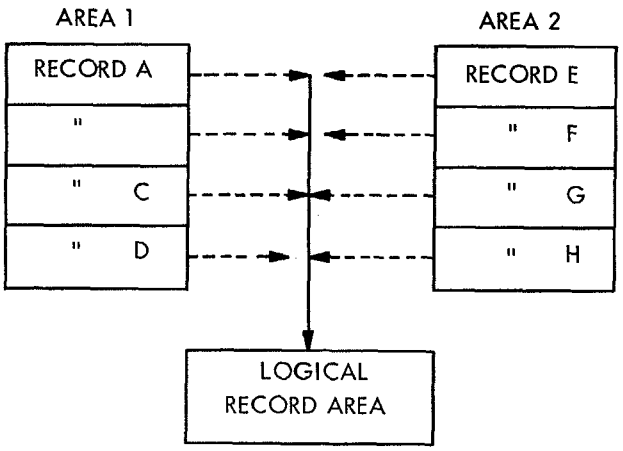

FIG. 3

ADDRESS MODIFIER ASSIGNED

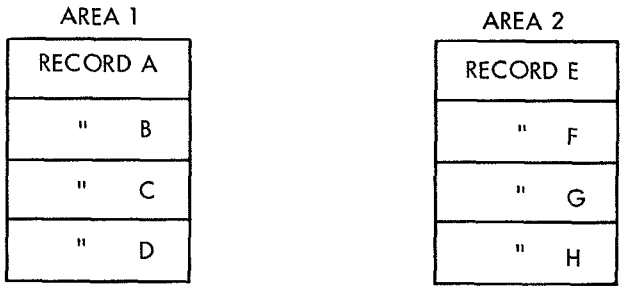

FIG. 4

Timing Objectives

Compute time for handling input-output of magnetic tape operations is an important factor in the evaluation of any input-output system. A programmer who codes his own input-output routines for a particular program normally makes a point of *not* timing the resultant speed particularly if it might not look too good. He is quick to forget that data movement should be included in the amount of compute time for input-output servicing. He also does not normally count the compute time required to provide the maximum use of simultaneity simply because he does not make this type of provision.

Speed objectives for the 601 input-output system are based on the total compute time to READ or WRITE a batched file, including one COMPARE of a 16-character criteria field. The actual formula for speed evaluation is as follows:

$$\frac{\left(\mathrm{LRT} + \frac{\mathrm{RCT}}{2^*} + \left(\mathrm{bst} + \frac{\mathrm{CMT}}{2^{**}}\right)\mathrm{ncr}\right)\mathrm{rb} + \mathrm{BT}}{(\mathrm{ncr}\cdot\mathrm{rb})\mathrm{ttc} + \mathrm{gt}} = \frac{\mathrm{CT}}{\mathrm{TT}}$$

* Comparing is by each logical read. No comparing is needed for writing.

** Because the system requires only one data movement for regrouping of batched files, moving may be done on each logical read or each logical write, but not both.

where

LRT = LOGICAL RECORD TIME/RECORD
bst = MACHINE BUFFER SERVICE TIME/CHARACTER
CMT = CHARACTER MOVE TIME/CHARACTER
ncr = NO. CHARACTERS PER RECORD
RCT = RECORD COMPARE TIME/RECORD
rb = RECORDS PER BLOCK
BT = BLOCK TIME/BLOCK
ttc = TAPE TIME PER CHARACTER
gt = GAP TIME
CT = TOTAL COMPUTE TIME
TT = TOTAL TAPE TIME

While LRT, RCT and BT vary in actual application, these variances are not great and an average can be taken for each. CMT varies based on whether the record is character, half-word (four characters), or word (eight characters) oriented. Because records should be word oriented when speed of processing is desired, word transfer time is used.

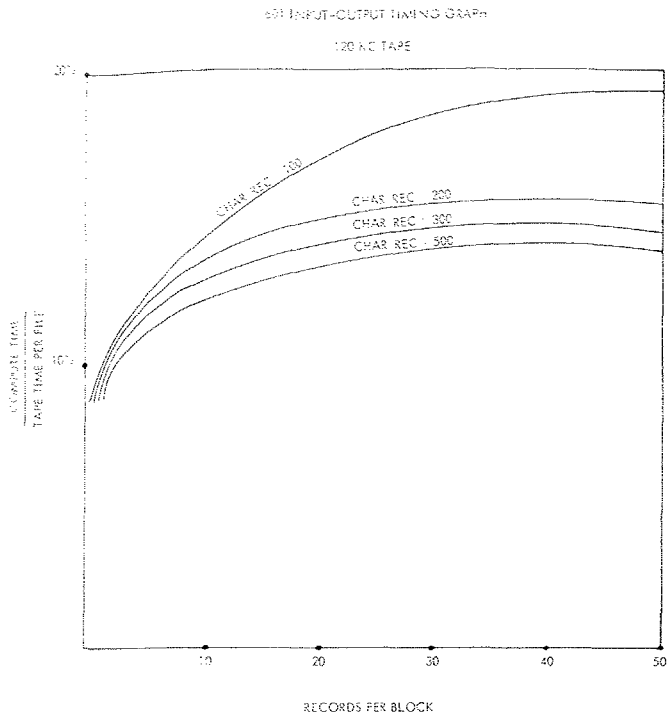

FIG. 5

This leaves five variables. The variables ncr and rb vary by the type of batching. The variables bst, ttc and gt vary by the type of tape (66KC or 120KC).

Approximate figures for LRT, RCT BT and CMT are as follows:

LRT = 75μs RCT = 30μs BT = 500μs CMT = .4μs

For 120KC Tape:

ttc = 8.3μs gt = 7400μs bst = 7μs

For 66KC Tape:

ttc = 15.0μs gt = 5500μs bst = 9μs

For 120KC tape, the formula reduces to

$$\frac{(90 + .9\text{ncr})\text{rb} + 500}{(\text{ncr}\cdot\text{rb})8.3 + 7400} = \frac{\text{COMPUTE TIME}}{\text{TAPE TIME}}$$

which can be graphed as shown in Figure 5. The graph 66KC tape takes the same basic form except that the percentages are less because the tape is not as fast.

It must be pointed out that the graph and formula shown include *machine* buffer service time in addition to basic input-output service time and one comparison of a criteric field. Although machine buffer service time is not programmed, it is a function of tape time and is therefore included. Formulas and graphs of this sort are an excellent method for evaluating whether a given I-O system fits a given computer.

Memory Requirement Objectives

By tying the I-O System into the library tape (segmentation), very reduced memory requirements are realized. Because of this fact this extended system requires only an approximate 500 words of memory.

An Introduction to a Machine-Independent Data Division

J. P. Mullin

Radio Corporation of America, Cherry Hill, N. J.

Of all the problems facing COBOL in the future, perhaps the most challenging is establishing a machine-independent Data Division. At the present time, COBOL provides a certain amount of flexibility in describing data so that the features of particular computers can be employed to achieve efficient object programs. While this approach provides a method for optimizing COBOL programs, programmer knowledge of individual computers is required in order that the merits of certain hardware features (synchronization, fixed vs. variable length data design, etc.) can be adequately measured.

In approaching the question of a machine-independent Data Division, the initial objectives of any system should be:

(A) A minimum of information required of the programmer

(B) No machine-oriented phrases needed in the source language

(C) A computer process to automatically determine
  (1) the sequence in which elementary items are arranged
  (2) whether fixed or variable length techniques should be used
  (3) the need for synchronization and other data design conventions (item separators, word marks, etc.) which permit optimum use of the object computer.

The relative efficiency of data design is a function of the procedures which operate on the data. The current concept of compiling, however, does not provide a mechanism for analyzing the frequency with which each procedural statement is performed at object time. One approach to permitting a compiler to determine where efficient data design is desirable could include some indication in the Procedure Division to indicate the main flow of the object program. Based on this information, a compiler could determine the