GRAPHICAL AUTOMATIC PROGRAMMING

A. Kossiakoff

Summary

THE DEVELOPMENT, PROOFING, AND MAINTENANCE of computer programs for complex data-processing systems represents a difficult and increasingly costly aspect of modern systems design, especially for those systems requiring real-time processing. The problem is aggravated by the absence of a lucid means of representing the operations performed by the program or its internal and external interfaces and the associated communication gap between engineers and programmers.

A major step toward the solution to this problem is believed to have been achieved. It consists of an entirely new approach devised to provide systems engineers with the capability of designing entire complex data-processing programs by direct use of a modern computer graphics terminal. The name "Graphical Automatic Programming" (GAP) has been suggested for this technique. Its four principal features are summarized below.

1. *Data Flow Circuit Language*—The essential basis for the Graphical Automatic Programming technique is the configuration of the program into "Data Flow Circuits," which represent the processing to be done in a form directly analogous to the diagrammatic representation of hardware circuits. Data Flow Circuits represent a "universal language" with a form intimately familiar to engineers and at the same time directly translatable into computer programs. The Data Flow language consists of a "vocabulary" of some 30 basic data-processing "elements," each of which represents an operation equivalent to the execution of a specific set of instructions in a general-purpose computer. These "Data Circuit Elements" are configured by the designer into an engineering-type Data Flow Circuit representing the data processing desired, as if they were equivalent hardware functional elements. The designer can also assemble and define special circuit elements for his own use.

The correspondence between the individual Data Circuit Elements and actual computer instructions makes it possible for the designer to assess the approximate time for executing each circuit path and the total core required to store the instructions. This permits him to balance the performance requirements for accuracy and capacity against the "cost" in terms of memory and execution time during the initial design of the circuit. This capability can be of utmost importance in programming high-data-rate real-time systems, especially those having limited memory capacity.

2. *Application of Computer Graphics*—Each Data Circuit is designed by the engineer with the aid of a graphics terminal, operated by a time-shared computer such as the IBM 360/91. The circuit elements are selected, arranged, and connected using a light pen and keyboard and displayed in a manner similar to that used in computer design of electronic circuits.

The display program stores the circuit description in an Element Interconnection Matrix and a data "Dictionary." This is checked automatically, and any inconsistencies in structure are immediately drawn to the designer's attention.

3. *Transformation of Graphical into Logical Form*—The computer then executes a transformation program, which converts the Data Flow Circuit automatically into an operational sequence, representing the sequential action of the circuit

elements as they would be serially processed by the computer. In the next step, the computer converts the operational sequence into computer assembly code for the computer driving the graphics terminal. The program logic is checked out by using sample inputs and examining the outputs. Errors or omissions can be corrected immediately by the designer by modifying the faulty connections or input conditions in the circuit.

4. *Integration and Testing of Complex Programs*—When checked out, the circuit is assembled by the computer with other blocks of the total program. The result is checked for proper operation. At any desired stage, the individual circuits or their assemblies can be translated into the machine assembly code of the particular computer on which the operational program is to run, which can be fed directly into the assembler of the operational computer. Subsequent modifications to the program can be made by calling up the circuit to be altered, making the changes with the display terminal, and invoking a program to find and change other affected sections.

In this way an entire complex computer program can be designed, documented, and managed through the use of Data Circuit language by direct interaction between the systems engineer and the graphics terminal. It is hoped that this technique will be capable of producing system software at a fraction of the time and cost achievable by current methods.

Introduction

The general-purpose digital computer found its first practical application in the forties in calculating more extensive tables of mathematical functions than had been practicable to produce by mechanical calculating machines. Few viewed the appearance of these devices as a breakthrough in technology. In fact, there was considerable speculation as to what would be left for them to do after the computation of function tables had satisfied all reasonable requirements.

As things have turned out, the digital computer did not run out of work when the most important mathematical tables were finished. Computers were applied with great success to problems in scientific and engineering analysis requiring highly complex mathematical calculations. They were also found to be extremely useful for economically storing large masses of data sorted in a way that permits almost instantaneous retrieval of a particular set of data. This capability is being exploited in major libraries and in almost every business.

A third type of application of enormous importance has been in the automation of operating systems. The Whirlwind computer, developed to automate the handling of radar information in the SAGE continental air-defense system, was one of the historic landmarks in the evolution of very-high-speed high-capacity digital computers.

The problems associated with these three applications of general-purpose digital computers are fundamentally very different. The application as a high-speed mathematical calculator involves the transformation of a set of given parameters by a sequence of specific mathematical transformations into one or a set of solutions. The storage and retrieval of data involves the organization of a filing system with suitable indexing to facilitate rapid location of the data to be retrieved. The use of a computer to automate portions of a complex system generally does not require involved computations or elaborate data retrieval. In this application, the primary processes are the correlation and classification of data inputs, recognition of significant events or changes in input conditions to the system, and translation of these into concise information outputs or actual control signals to external devices.

The design of computer programs for using digital computers in automating real-time operating systems has turned out to be quite different and much more difficult than designing programs for the computing and data-handling applications. Thus the enormous potential impact of the use of modern digital computers in automating such systems has been impeded by the very large expenditure of manpower, and hence of time and money, in the design of satisfactory large-scale computer programs. In many instances the development of the so-called "software" (in contrast to the "hardware," or equipment) is widely regarded as the limiting factor in both time and cost of system development.

Operation and Programming of a General-Purpose Digital Computer—For those with only general familiarity with the problem of designing computer programs, a brief discussion of how a general-purpose computer operates and of aids available to programming is necessary to provide a background for the description of the Graphical Automatic Programming techniques.

The general-purpose digital computer consists of a central processing unit (CPU), a random access memory unit (Core), and peripheral "input-output" (I/O) devices. The central processing unit contains a set of circuits, each designed to execute a specific operation on one or more digital "words" which represent numbers or characters making up the data. The execution of each operation is called for by a coded word called an "instruction." A sequence of instructions calling for a series of operations is called a "program." The instructions are stored in part of the random-access memory, along with data words in various states of processing. Data are entered into the computer by means of punched cards, tape, or disk. Control signals are usually entered through special interrupt connections. Results are read out in the same form as data inputs, or as printed copy, plots, or graphic displays. They may also be used to control various devices directly.

The main task in effectively using a general-purpose computer in a given application is the development of a satisfactory computer program. Since the individual operations of the central processing unit are very elementary, a relatively long sequence of instructions must be written to accomplish most data-processing tasks. Care must be taken to insure that adequate space is reserved in memory to hold data inputs and partially processed information. Where the process involves decision points and "branches" into alternative paths, it is not unusual to make mistakes in the proper sequencing of instructions. Sequencing errors are inherently difficult to locate, so that "debugging" of the program is usually the most time-consuming part of the job.

The actual instructions stored in the computer to execute a given program are made up of binary code. The lowest language readily intelligible to a programmer is called "assembly language." This is a direct representation of machine code for each instruction, in characters which convey meaning, as for example "LDA(M)" which means "Load the contents of memory location M into the A register." The assembly language can use names to refer to memory addresses and labels for instruction locations. This notation makes it possible to write a program without assigning particular locations in memory for data files (or arrays) and the instructions themselves. Each computer model has its own assembly code notation and its own "assembler" which automatically translates the assembly language program into machine instructions.

Since the program in assembly code requires a separate instruction for each elementary machine instruction, it is very laborious to use in designing complex programs. For this reason several "programming languages" have been developed which enable the programmer to write concise "higher level" instructions. This involves development of a program called a "compiler," which translates the high-level instructions into the assembly code for a given computer. Since much of the detailed housekeeping is done by the compiler, the programmer's task is greatly facilitated.

The most widely used computer language has been "FORTRAN," which was developed by IBM for programming mathematical calculations. "ALGOL" is a more sophisticated algebraic language. For programming data storage and retrieval operations, such as those used predominantly in business applications, "COBOL" is widely used. More recently IBM combined the best features of FORTRAN and COBOL into "PL/I" (Programming Language One) to handle both types of program. In addition to these general-purpose languages, a variety of higher level programming aids have been developed to handle special applications.

Problems in Design of Real-Time Programs—Unfortunately, while these higher level languages are very helpful in programming computers for use in mathematical analysis and business applications, they do not lend themselves to the design of real-time programs for complex automated systems. In such applications, the program has to provide for accessing and outputting data at times required by the system timing, often in a period of a millisecond or less, and it must have a system of priorities which interrupts lengthy operations in favor of those requiring immediate action. It must be subject to external commands by operators, to adapt the processing priorities or modes to changes in the operational environment. The higher level languages obscure the relation between the operation called for and the time required for its execution, and hence can inadvertently produce a program which later proves to require unacceptably long processing times. "Timing" in scientific or business programs generally only affects cost. In high-data-rate real-time systems timing may control success or failure.

Automated systems must often accommodate

large variations in the volume and rate of data inputs, and in their quality or noise content. The use of a higher level language obscures the memory requirement for storing the program code and data. The resulting inefficient use of the memory is often a limiting factor on data-handling capacity. In such systems the use of assembly language, in which the execution time and memory required for each instruction is immediately apparent, is more satisfactory in insuring that the program meets all system requirements, despite the increased labor involved in the detailed coding. These characteristics make the design of computer programs for real-time systems vastly more difficult and tedious than the preparation of programs for batch-type computational tasks.

An even more basic difficulty in the preparation of effective programs for computers which serve as permanent elements in complex automated systems is the "communication gap" between the engineers and the programmers. The design specifications on the program are prepared by engineers to fit the characteristics of the data inputs and the rate and accuracy requirements of the processed outputs. Capacity is often dictated by operational factors. At the time he has to make these specifications, the system engineer does not have effective means to estimate reliably the complexity of the program that will result. The programmer, in turn, has little discretion in altering the specifications to meet the limitations on computer capacity and processing times. Accordingly, the development of the system computer program is effectively an open-loop process, and often results in an oversized and unbalanced product after an inordinate expenditure of effort and time.

Data Flow Circuit Language

The fundamental new concept which constitutes the essential basis of the techniques of Graphical Automatic Programming is the representation of a computer program in a "language" consisting of circuit networks in a form directly analogous to diagrams used by engineers to lay out electronic circuits. This representation focuses attention on the "flow" of identifiable *data* inputs, quantized in the form of digital words, through alternative paths or "branches" making up the total data-processing network. The switching of data flow at the branch points of the network is done by *control* signals generated in accordance with the required logic. These control signals are equivalent to "jump" instructions in the digital program. The term "Data Flow Circuit" will be used in referring to this representation.

The development of the Data Flow Circuit representation has required the definition of a set of "elements" which constitute the building blocks of the circuit. Each element has a dual meaning: to the engineer it represents a transfer function and to the programmer it represents a set of computer operations. Before describing these building blocks, it is illuminating to compare the general form of a typical computer program with its equivalent Data Flow Circuit.

A computer program representative of a practical example of real-time programming is illustrated in Fig. 1. It lists the code of one of the processes used in a program for automatic tracking of target returns from a three-dimensional search radar. This program is written for the Honeywell DDP-516 computer—a small modern high-speed machine, with a relatively simple but versatile set of instructions.

The figure shows both the DDP-516 machine code and the corresponding assembly code for each instruction. The machine code is listed in the columns of numbers on the left side of the figure, and the equivalent assembly code is listed in the middle columns of characters. The same code will be used in examples described in later sections of this paper. The text at the right lists comments written by the programmer for his own reference in "debugging" or modifying the program. The program consists of some 100 instructions. Since it does not have an obvious form or structure, a typical program such as this is difficult to follow by anyone except the programmer who wrote it.

The representation of the same process in the form of a Data Flow Circuit is shown in Fig. 2. The solid (red) lines represent the flow of *data* in the form of digital words, and thus trace the successive operations on a given data input. The dashed (blue) lines represent *control* signals transmitted to gates which activate particular operations or data paths, and thus effect branching in the operational sequence.

The polygons in Fig. 2 represent the 12 main functional elements in the circuit. The shape of the element and the number and types of signal inputs and outputs indicate the general type of function it performs, while the characters inside

```
0184                                        -INPUTS-
0185                          (HIT) = T(1) DELTA E(3) A(3) O(3) DELTA R(6)
0186                         (TD41)=ADDRESS OF ITEM IN TD STORE WITH FORMAT
0187                                R(1) N(3) A(3) O(3) DELTA R(6)
0188                         (TD51)=ADDRESS OF ITEM IN TD STORE       WITH FORMAT B(10) E(6)
0189                         (BGEL)=BRNG AND ELEV. ASSOCIATED WITH (HIT). FORMAT B(10) E(6)
0190
0191    003250 -0 02 00645       LDA*  TD51          WILL BE -1 IF NO ENTRY
0192    003251  140401            CMA
0193    003252  101040            SNZ
0194    003253  0 01 03323        JMP   HCE         NO PREVIOUS HIT ENTRY
0195                          COMPARE HIT WITH PREVIOUS ENTRY
0196    003254 -0 02 00644       LDA*  TD41
0197    003255  0 04 03407        STA   HCT4
0198    003256  0 03 03416        ANA   =177        EXTRACT DELTA R FROM PREVIOUS ENTRY
0199    003257  0 04 03404        STA   HCT1
0200    003260  0 02 00612        LDA   HIT
0201    003261  0 03 03416        ANA   =177        EXTRACT DELTA R FROM HIT
0202    003262  0 07 03404        SUB   HCT1
0203    003263  100400            SPL               SEE IF EQUAL UP TO LAST BIT
0204    003264  141407            TCA
0205    003265  0 03 03417        ANA   =|177776
0206    003266  100040            SZE
0207    003267  0 01 03403        JMP   NTE         NO, GO TO MULT. TARG. ROUTINE
0208    003270  0 02 00612        LDA   HIT         EXTRACT A FROM HIT
0209    003271  0 03 03420        ANA   =|7000
0210    003272  0 04 03404        STA   HCT1
0211    003273  0 02 03407        LDA   HCT4

0212    003274  0 03 03420        ANA   =|7000      EXTRACT A FROM PREV. ENTRY
0213    003275  0 11 03404        CAS   HCT1
0214    003276  0 01 03317        JMP   PHC2        A (PREV) G.T. A (HIT)
0215    003277  0 01 03330        JMP   HCA         EQUAL, GO TO AVERAGE COORDINATES
0216    003300  0 02 00612        LDA   HIT         INSERT HIT INTO CG TABLE
0217    003301  0 03 03421        ANA   =|7077      EXTRACT A AND DELTA R FROM HIT
0218    003302  0 04 03404        STA   HCT1
0219    003303  0 02 03407        LDA   HCT4
0220    003304  0 03 03422        ANA   =|170000    EXTRACT N AND N FROM ENTRY
0221    003305  0 06 03404        ADD   HCT1        COMBINE WITH HIT
0222    003306  0 06 03423        ADD   =|10000     ADVANCE N
0223    003307 -0 04 00644        STA*  TD41
0224                          FORM SECOND WORD OF CG ENTRY
0225    003310  0 02 00612  PHC1  LDA   HIT
0226    003311  141340            ICA               EXTRACT DELTA E FROM HIT
0227    003312  0405 74           ABS   4
0228    003313  0 03 03424        ANA   =|7
0229    003314  0 06 00602        ADD   BGEL        ADD RADAR COORDINATES       B(10)E(6)
0230    003315 -0 04 00645        STA*  TD51        STORE IN TD STORE
0231    003316 -0 01 03247        JMP*  PHC         RETURN
0232                          FOR A (PREV) G.T. A (HIT), ADVANCE N AND RETURN
0233    003317  0 02 03407  PHC2  LDA   HCT4
0234    003320  0 06 03423        ADD   =|10000
0235    003321 -0 04 00644        STA*  TD41
0236    003322 -0 01 03247        JMP*  PHC
0237                          NO PREVIOUS ENTRY
0238    003323  0 02 00612  HCE   LDA   HIT
0239    003324  0 03 03421        ANA   =|7077      EXTRACT A AND DELTA R FROM HIT
0240    003325  0 06 03423        ADD   =|10000     SET N=1
0241    003326 -0 04 00644        STA*  TD41   STORE IN TD TABLE
0242    003327  0 01 03310        JMP   PHC1
0243                          HIT COORDINATE AVERAGING (EQUAL AMPLITUDES)
0244                          BEFORE COMPUTING AVERAGE BEARING, SEE IF NEW
0245                          HIT AND PREV. ENTRY ARE ON OPPOSITE SIDES OF NORTH
0246    003330  0 02 00602  HCA   LDA   BGEL
0247    003331  0404 76           LGE   2
0248    003332  0 03 03425        ANA   =|177760
0249    003333  0 04 03404        STA   HCT1        BEARING OF NEW HIT
0250    003334 -0 02 00645        LDA*  TD51
0251    003335  0404 76           LGE   2

0252    003336  0 03 03425        ANA   =|177760  BEARING OF PREV. ENTRY
0253    003337  0 04 03405        STA   HCT2
0254    003340  0 06 03404        ADD   HCT1
0255    003341  0 04 03406        STA   HCT3        SUM OF BEARINGS
0256    003342  0 07 03404        SUB   HCT1
0257    003343  0 07 03404        SUB   HCT1        DIFFERENCE OF BEARINGS
0258    003344  100400            SPL
0259    003345  141407            TCA               ABS. VALUE OF DIFFERENCE
0260    003346  0 11 03426        CAS   =10089     COMPARE DIFF. WITH PI
0261    003347  0 01 03352        JMP   *+3         OPPOSITE SIDES OF NORTH
0262    003350  0 01 03352        JMP   *+2         OPPOSITE
0263    003351  0 01 03363        JMP   HCA3        SAME SIDE OF NORTH
0264    003352  0 02 03406        LDA   HCT3        LOAD BEARING SUM
0265    003353  0 11 03427        CAS   =20189     COMPARE WITH 2 PI
0266    003354  0 01 03355        JMP   *+1         AND ADD OR SUBTRACT 2 PI
0267    003355  0 07 03430        SUB   =40289 AS NECESSARY.
0268    003356  0 06 03427        ADD   =20189
0269    003357  0414 77     HCA2  LGL   1           SHIFT SO THAT RESULT IS DIVIDED
0270    003360  0 03 03431        ANA   =|177700    EXTRACT 10 BIT BEARING
0271    003361  0 04 03404        STA   HCT1        BY 2.
0272    003362  0 01 03365        JMP   HCA4        GO COMPUTE AVER. ELEVATION
0273    003363  0 02 03406  HCA3  LDA   HCT3        LOAD BEARING SUM
0274    003364  0 01 03357        JMP   HCA2
0275
0276                          COMPUTE AVERAGE ELEVATION
0277    003365  0 02 00612  HCA4  LDA   HIT         FIRST COMPUTE ELEV. OF NEW HIT
0278    003366  141340            ICA
0279    003367  0405 74           ABS   4
0280    003370  0 03 03424        ANA   =|7        EXTRACT DELTA E FROM HIT
0281    003371  0 06 00602        ADD   BGEL        ADD GROUP ELEV.
0282    003372  0 03 03416        ANA   =|177
0283    003373  0 04 03405        STA   HCT2
0284    003374 -0 02 00645        LDA*  TD51              GET EL. OF PREV. ENTRY
0285    003375  0 03 03416        ANA   =|177
0286    003376  0 06 03405        ADD   HCT2        ADD EL. OF HIT
0287    003377  0404 77           LGR   1           DIVIDE BY 2
0288    003400  0 06 03404        ADD   HCT1        COMBINE WITH AVERAGE BEARING
0289    003401 -0 04 00645        STA*  TD51              STORE COORDINATES IN TD STORE
0290    003402  0 01 03317        JMP   PHC2
0291

0292
0293    003403  000000       NTE   HLT
0294    003404                HCT1  BSS   1
0295    003405                HCT2  BSS   1
0296    003406                HCT3  BSS   1
0297    003407                HCT4  BSS   1
0298                          TD41  EQU   |644   ADDRESS OF TD STORE WITH FORMATW(3)A(3)O(3)DR(6)
0299                          TD51  EQU   |645   ADDRESS OF TD STORE WITH FORMAT V   B(10) E(6)
0300                          HIT   EQU   |612
0301                          BGEL  EQU   |602
0302                                END

          003410  000700
          003411  000010
          003412  003777
          003413  000030
          003414  014000
          003415  016000
          003416  000077
          003417  177776
          003420  007000
          003421  007077
          003422  170000
          003423  010000
          003424  000007
          003425  077760
          003426  014400
          003427  031100
          003430  062200
          003431  177700
```

Fig. 1—Target Coordinate Computation Circuit Program.

define its specific operation. Thus, the visual configuration of the circuit is descriptive of its general operational function.

The routing of data and control signals among

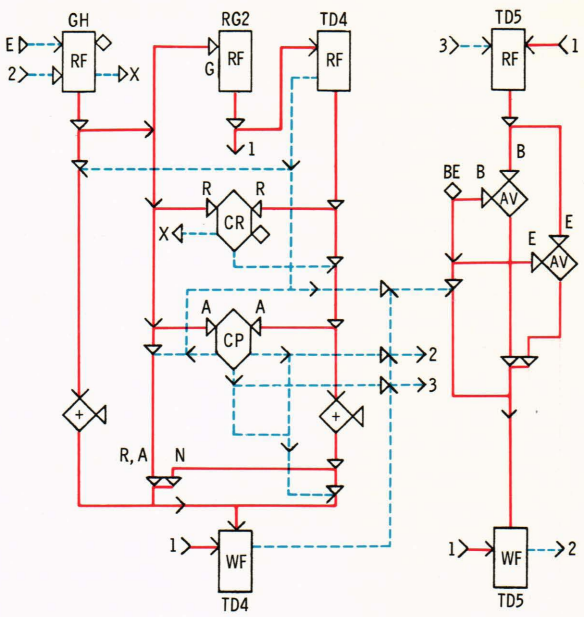


Fig. 2—Target Coordinate Computation Circuit.

the different branches of the circuit is accomplished by a secondary class of elements represented by characteristic configurations of open and closed arrowheads. A closed arrowhead at an input to a functional element labeled by a letter indicates that the input is only the part of the data word which contains the variable represented by the label.

Some of the inherent advantages of using the Data Flow Circuit representation for the programming of real-time systems can be seen from a general comparison between Figs. 1 and 2. The ability of following the operations performed on each data input in Fig. 2 makes the interaction of different variables readily visible. The ease of representing branching at decision points and tracing the resultant paths through the circuit network reveals possible logical traps to an engineer much more readily than the conventional logic flow diagram in which the path of data flow is not shown. It is easy to spot redundant operations, which can be combined.

Perhaps of equal significance is the fact that each circuit element, when used in a given computer, has associated with it a definite set of instructions, except for minor variations depending on the form of the inputs, and hence the number of words in Core, and time to execute, can be estimated quite closely at the outset. For example, it will be seen later that the COMPARE element,

designated in Fig. 2 by a hexagon marked by the characters "CP," requires four instructions for the DDP-516 computer. In general, each signal routing element requires an average of one instruction, while the main functional elements require an average of six instructions counting the preparation of data inputs. This knowledge gives the designer a measure of the size of the program equivalent to the circuit and the approximate transit time through any of the possible circuit paths. If either the size or time of the equivalent program appears excessive, the designer can seek to simplify the processing operations at the very outset so as to achieve a well balanced program.

A Data Flow Circuit is conceptually equivalent to an actual circuit constructed from a multiplicity of special-purpose digital circuit elements. This equivalence was in fact the origin of the idea, when it was realized that special-purpose digital operational elements are becoming so inexpensive that they could be used to advantage in high-data-rate real-time data processing to complement the general-purpose computer.

Digital circuits differ from analog circuits by virtue of the fact that in the former the signals are "quantized" in the form of digital "words." This means that signal transformation and "flow" occur by a series of steps rather than continuously. A Data Flow Circuit differs from an ordinary digital circuit in that the steps are further restricted to take place one at a time to correspond to the sequence of operations by the computer. Thus, while data will flow in parallel paths in a circuit network such as that shown in Fig. 2, at any given instant signals will be flowing in only one of the paths. This characteristic does not detract from the high visibility of all of the interactions in the process inherent in the diagrammatic representation.

Data Circuit Elements—As stated previously, in a Data Flow Circuit each functional element has a dual meaning. In the engineering representation it can be considered to be exactly equivalent to a hardware building block, which transforms the indicated digital inputs into a uniquely defined set of output signals. In its representation of a sequence of operations performed by a general-purpose digital computer, it corresponds to a definable set of instructions in computer assembly language.

In selecting the building blocks to be used as the functional elements of Data Flow Circuits, each Data Circuit Element was designed to meet the following criteria:

1. It must be sufficiently basic as to have wide application in data-processing systems.
2. It must be sufficiently powerful to save the designer from excessive detailing of secondary processes.
3. It must have a symbolic form which is simple to represent, which is meaningful in terms of its characteristic function, but which is not readily confused with existing component notation.

The choice and definition of the basic GAP data circuit elements is in an early stage and will evolve as experience is gained in applying the language to practical problems. At present five classes of circuit element have been defined. The first four perform direct signal transformations, and the last routes data and control signals through the circuit. These classes are defined as follows:

SENSE elements test a particular characteristic of a data word and produce one of two control outputs according to whether the result of the test was positive (Yes) or negative (No).

OPERATOR elements perform arithmetic or logical operations on a pair of data inputs and produce a data word.

COMPARISON elements combine several sensing and operator functions in a single element to accomplish frequently used data classification operations.

TRANSFER elements bring data in and out of the circuit from files in memory and from external devices.

ROUTING elements combine, split, and gate the flow of data and control signals in the circuit. Some routing elements do not themselves produce program instructions, but rather modify those produced by the functional elements to which they are connected.

Table 1 lists the elements presently defined for initial use of GAP. These include 2 SENSE elements, 11 OPERATOR elements, 4 COMPARISON elements, 6 TRANSFER elements, and 9 ROUTING elements. Others found to be widely applicable may be added to the basic vocabulary for general use. Facility will be provided for each designer to define for his own use special-purpose functions as auxiliary elements. Most of these can be built up from combinations of the basic elements, as is true of the COMPARISON elements already defined.

TABLE 1

DATA FLOW CIRCUIT ELEMENTS

SENSE ELEMENTS
- SENSE ZERO
- SENSE SIGN

OPERATOR ELEMENTS
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE
- AVERAGE
- EXPONENTIATE
- MAXIMUM
- MINIMUM
- LOGICAL AND
- EXCLUSIVE OR
- INCLUSIVE OR

COMPARISON ELEMENTS
- COMPARE
- THRESHOLD
- CORRELATE
- RANGE GATE

TRANSFER ELEMENTS
- READ FILE
- WRITE FILE
- INPUT DATA
- OUTPUT DATA
- FUNCTION TABLE
- SHIFT REGISTER

ROUTING ELEMENTS
- DATA SPLIT
- DATA JUNCTION
- DATA GATE
- DATA PACK
- DATA LOOP
- CONTROL SPLIT
- CONTROL JUNCTION
- CONTROL GATE
- SIGNAL SPLIT

Figure 3 illustrates the symbolic representation of typical circuit elements. The top rows picture one element of each of the four main functional groups, while the bottom rows illustrate four ROUTING elements. As noted previously, solid lines are used for data signals and dashed lines for control signals.

In Fig. 3 the sample elements are seen to have the following types and numbers of connections:

Element Type	*Name*	*Data Inputs*	*Control Inputs*	*Data Outputs*	*Control Outputs*
SENSE	SENSE ZERO	1	0	1	2
OPERATOR	ADD	2	1	1	0
COMPARISON	COMPARE	2	1	0	2–3
TRANSFER	READ FILE	1	2	1	1
ROUTING	DATA SPLIT	1	0	2–5	0
	DATA GATE	1	1	1	0
	CONTROL JUNCTION	0	2–5	0	1
	DATA LOOP	2	1	1	0

OPERATOR, COMPARISON, and TRANSFER elements are provided with an optional control input to serve as a gate for delaying the functioning of the element until the receipt of a control signal from elsewhere in the circuit. The READ FILE and DATA LOOP elements have a control input which serves a different purpose, namely to initiate the next cycle of the loop.

The maximum number of connections for any element is six, and for SENSE and OPERATOR elements it is four. Connections are numbered clockwise with #1 at 12 o'clock.

Operation of Data Circuit Elements—The characteristics of Data Circuit Elements can best be described by examples. Five of the elements shown in Fig. 3 will later be used in a simple circuit to illustrate the automatic translation of a Data Circuit into a computer program. The detailed operation and equivalent code of these five elements are described below.

The function of the COMPARE element is to emit a control signal from one of its three output connections in accordance with the relative magnitude of the two data inputs, x and y. As the signals in a Data Circuit flow from an output of one element to an input of another, one link at a time, the step when a given Data Element performs its function and generates an output occurs when the final input necessary for its operation arrives. In the case of the COMPARE element in its basic ungated form, two data inputs are required. When the first arrives it is put in a temporary memory location. When the second arrives, usually several steps later, the element functions and generates the appropriate output, which in this instance is a control output from connection #3, #4, or #5.

In translating the functioning of the element into computer assembly code, the conditions at the time of functioning must be noted. When the COMPARE element is activated by the arrival of the input at connection #2, the corresponding data word is in a general register AR, while the other data input is in a temporary memory location, M6. The resulting code would have the form listed below for computers having a specific "Compare" instruction. The instructions in word form are listed in the left column and the equivalent instructions in DDP-516 assembly code are listed at the right.

1. Compare AR with M6 CAS CP6
2. Jump to M3 (if AR > M6) JMP CP3
3. Jump to M4 (if AR = M6) JMP CP4
4. Jump to M5 (if AR < M6) JMP CP5

In the above, code labels are used to designate memory locations containing data, as M6 or CP6, or instructions, as M3, M4, M5 or CP3, CP4, CP5. In the rest of the paper it will be convenient to relate these labels, which are entirely arbitrary, to the notation of the corresponding element input or output connections.

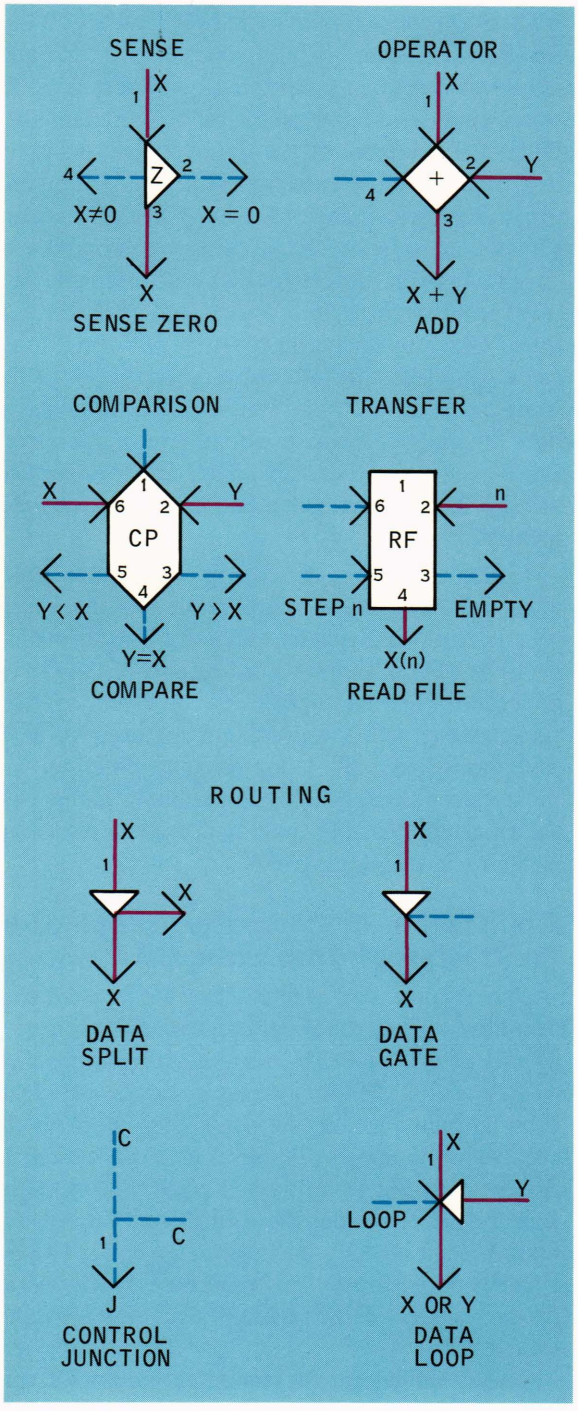


Fig. 3—Examples of Data Circuit elements.

Like most functional elements, the COMPARE element has a connection (#1) that can be used as a control gate when it is desired to make the operation conditional on a particular control output from another element. This connection saves the use of a separate gating element on one of the data inputs. When gating is used, the second data input is also temporarily stored in memory (M2) and operation of the element occurs when the gating input arrives. The code for the gated form of the COMPARE element begins with the instruction

M1: Load M2 into AR CP1: LDA CP2

followed by the same code as the ungated form. The "M1:" (or CP1:) preceding the instruction is a label used in assembly language to indicate the destination of a jump instruction.

The COMPARE element also has another form in which either control output #3 or #5 is designed as "unconnected." Output #4 then becomes a "greater than or equal to" output. In this form an output at #4 will be produced when M2 ≥ M6. Since such simple variants of an element are distinguishable in the diagram by the status of specific optional connections, the same basic element can be used for several closely related functions without ambiguity.

The READ FILE element has the function of extracting one or a series of data words from an array or file in memory. In its fully connected form it is designed to operate in a circuit "loop," extracting one word of a sequence at each turn until the file is empty. If the stepping control input at connection #5 is designated as unconnected, the READ FILE element will extract a single data word from the file location designated by the input at connection #2.

In the fully connected form of the READ FILE element, the input required to generate the code is the control input at connection #6. When this input arrives, the element reads out the data word located at the address indicated by the initial value, n, of the index, i.e. the number of items in the file to be read out, which has been stored previously at the data input at connection #2.

After the extracted word has been processed, a "stepping" control pulse is received at connection #5. This input causes the index to step to the next word on the list. If the incremented value of the index shows that no words remain, a control output appears at connection #3. If not, the next word is read out at connection #4, initiating the next cycle of the loop.

The translation of the READ FILE element into assembly language is written in a single sequence of instructions as soon as the first word is read out. The differentiation between the initial and stepping modes is done by the use of labels which indicate the entry points for the two modes.

The code for the READ FILE element is shown below in its generalized form on the left and in DDP-516 assembly code on the right. The "IRS" instruction in the 516 code stands for "increment, replace, and skip" and has the function of incrementing the contents of the indicated memory location by one and skipping the next instruction if the result is zero. "SKP" is an unconditional skip instruction. The significance of the other instructions is obvious.

1.	M5: Increment M2	RF5:	IRS	RF2
			SKP	
2.	Jump to M3 if M2 = 0		JMP	RF3
3.	M6: Load M2 into XR	RF6:	LDX	RF2
4.	Load M1, X into AR		LDA	RF1, 1

M5 is the label of the jump instruction which provides the gating input to connection #5. M6 is the label corresponding to the readout of the first word on the list. Thus an instruction calling for a jump to M6 would result in the execution of instructions 3 and 4, and eventual return to the loop at instruction 1. M1, X stands for the X'th entry in the file whose base address is in M1, and where X is the contents of the index register.

The DATA SPLIT element routes a data signal from one element to as many as five other circuit elements. The data input is temporarily stored by the code:

1.	Store AR in M1	STA	DS1

The DATA GATE is used to inhibit the flow of a data signal until a control signal is received. At that time the data input, which had been stored in a temporary memory location, M1, is reloaded into the general register for subsequent processing. The corresponding code is:

1.	M2: Load M1 in AR	DG2:	LDA	DG1

The CONTROL JUNCTION routes several different control outputs to a single element input. While it does not in general produce code, it does change the labels of jump instructions on the connected elements.

Data Preparation—The word length in most general-purpose computers varies between 16 and 36 bits. The accuracy with which a given variable is known is seldom greater than one part in 2000, which requires 11 bits plus 1 bit to designate sign. Often the accuracy of the data requires 8 bits or less. Since memory capacity is often a limiting factor in the performance of a computer as a system element, it is frequently necessary to combine or "pack" two or more variables into a single data word to economize on memory storage and access time. When an operation must be performed on a given variable, the latter must first be extracted from the data word and manipulated to adjust its sign bit and location to put it into proper form for the ensuing operation. The data preparation usually involves several mask, shift, and complement instructions.

In the Data Flow Circuit notation, such preparation is specified as a preliminary to the operation performed by each element. The format of each variable is also specified as part of the circuit definition. The manipulations involved in data preparation, which represent a major portion of the "housekeeping" labor in programming, are thereafter accomplished automatically along with the translation of the functional operations of the elements in the Data Circuit.

Application of Computer Graphics to the Design of Data Flow Circuits

The second key element in the technique of Graphical Automatic Programming is the utilization of the newly available computer-driven displays to help the designer lay out a satisfactory Data Flow Circuit, and at the same time store in the computer a complete description of the circuit as drawn. This latter step lays the necessary foundation for automating the transformation of the Data Circuit directly into computer code. The net result is an enormous saving in time in the overall process of Data Flow circuit design, checkout, and translation.

The development of computer graphics terminals enables the engineer to use the computer without writing a computer program. An example

of a modern graphics terminal is the IBM 2250, which can be driven by most of the IBM 360 computers. The display has a 10-in. x 10-in. cathode-ray-tube screen, a typewriter keyboard, a set of special control keys, and a light pen for direct interaction between the display and the operator. The operator uses the light pen to indicate the point at which he wishes a line or other symbol to appear, or the symbol which he wishes to select, erase, or otherwise operate on as he may direct by the keyboard.

Graphics terminals have greatly broadened the utility of computers as direct aids to many human tasks. By enabling the operator to "talk" with pointers and English words rather than through an elaborate code, they are revolutionizing many tasks. For example, a computer program called "ECAP," together with a graphics terminal, enables an engineer to "draw" an electronic circuit on the face of the display, punch in the component values he wishes to try, and in a few moments it gives him the salient characteristics of the circuit. If these characteristics are outside the desired limits, the engineer can adjust component values, alter connections, insert or delete components, and get essentially instantaneous feedback of the effects on performance. This technique promises to shorten the time for circuit design drastically.

In the graphic display program for the design of electronic circuits, the available components are first displayed at the bottom of the screen. They are then located in the circuit by pointing in turn to the desired component and then to the desired location on the screen with the light pen. The scanning beam in the display recognizes the location of the light pen, associates it with the component, and positions it accordingly. Elements are connected by simply pointing the pen at each of the terminals to be joined.

The successful development of such a powerful technique for the design of electronic circuits suggested that computer graphics might equally help accomplish direct and real-time transformation of Data Circuits into computer routines. The programming of the computer to accomplish this is, of course, quite different from "ECAP," but the property of communication between the engineer and computer by means of symbols and light pen is the same.

The display of a Data Circuit is accomplished in the same general manner as that described above for conventional electronic circuits. The symbols used are those defined in Fig. 3 for the Data Circuit elements, with the appropriate character code specifying the member of the element class. Figure 4 shows the display of the circuit of Fig. 2 on the IBM 2250 terminal.

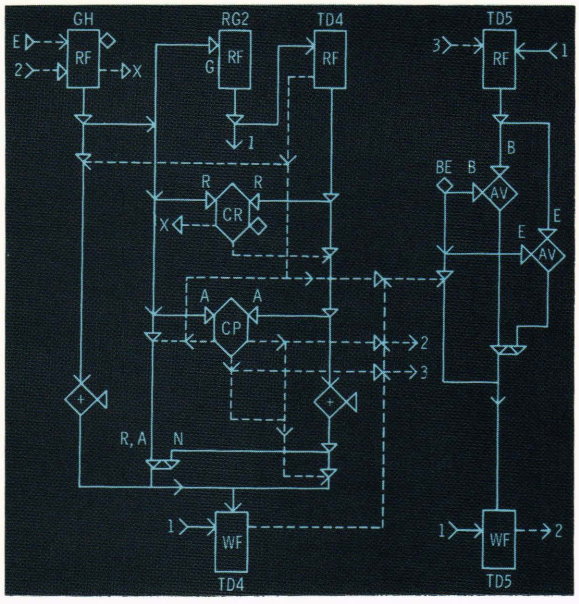


Fig. 4—Graphic display of circuit of Fig. 2.

The GAP graphic display program is designed to fulfill the following functions:

1. To display the element symbols located by the designer, storing the location of all element connections.
2. To display the data and control connections between the elements, and any special notation entered by the designer, including data preparation.
3. To associate the linked elements into an "Interconnection Matrix."
4. To check for any obvious errors in the diagram and to signal them to the designer.
5. To interact with the designer in the later stages of program generation by displaying anomalies or altering the circuit as directed.

Example of the Graphical Design of a Data Flow Circuit—The following elementary process illustrates how a simple Data Flow Circuit would be designed.

Data Inputs:

1. A number of potential target returns or

"hits" have been received by a radar during several dwells.

2. The Amplitude, A, and Range, R, of each hit have been encoded into a single word A, R.

3. The hits have been listed sequentially in a file.

Data Processing:

1. All hits whose amplitude equals or exceeds a certain threshold are to be retained and stored in another file for further processing.

2. Other hits are to be rejected.

The representation of this process in a conventional programmer's Logic Flow Diagram is shown in Fig. 5. In the figure, a potential target return is called a "HIT," and a return exceeding the threshold is called a "TRK," a mnemonic for "track." The diagram shows the steps required in indexing and the three decision branch points which occur when the amplitude is below the threshold or when either file is exhausted.

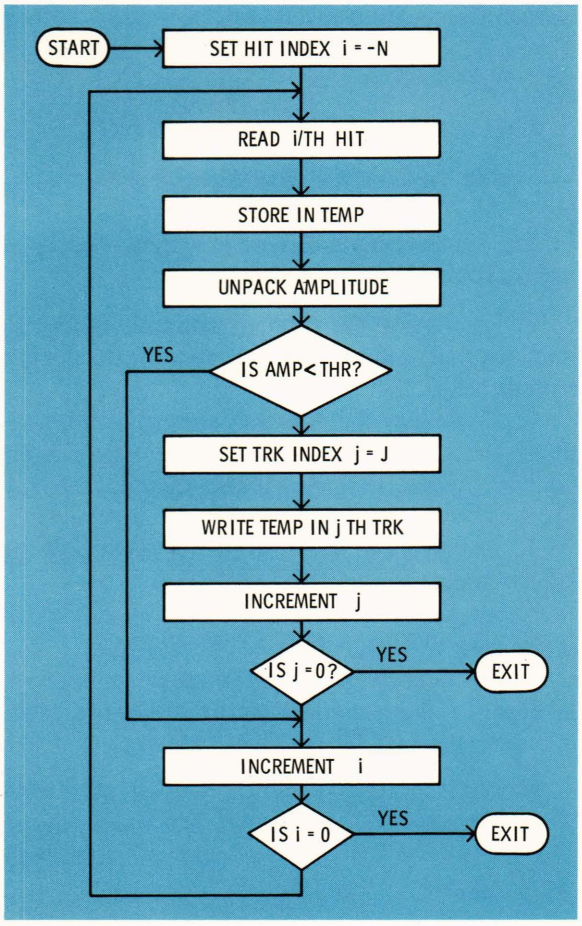


Fig. 5—Hit Sorting logic flow diagram.

The representation of this data process in Data Flow Circuit language can be accomplished by the use of three functional elements.

1. READ FILE, to extract each hit from the hit entry file.

2. COMPARE, to select hits whose amplitude equals or exceeds the threshold.

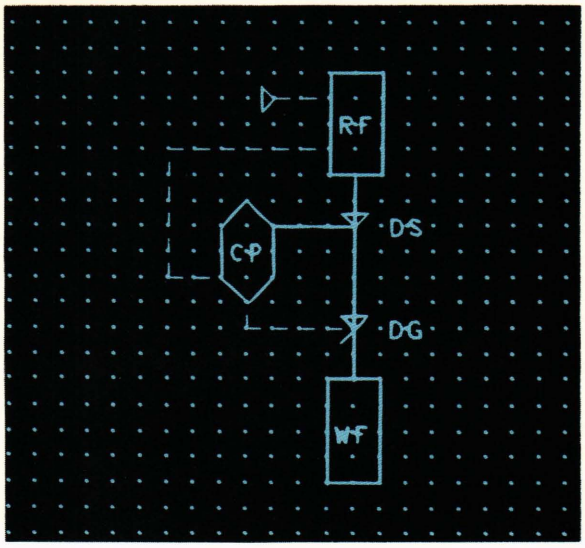
3. WRITE FILE, to enter the selected hits into another file for retention.

The designer selects the READ FILE (RF) and WRITE FILE (WF) from the TRANSFER elements and positions them on the screen with the aid of a ¼-in. grid used during circuit assembly. He positions the COMPARE (CP) element to one side to provide the path for the hit selection logic. He then selects and locates the signal ROUTING elements and connects the element inputs and outputs with data (solid) or control (dashed) lines. The ROUTING elements required are a DATA SPLIT (DS) to route the extracted hit to both the COMPARE element and the WRITE FILE element, and a DATA GATE (DG) to pass the hit for entry only if the comparison shows that its amplitude passes the threshold. The partially completed circuit diagram is shown in Fig. 6a.

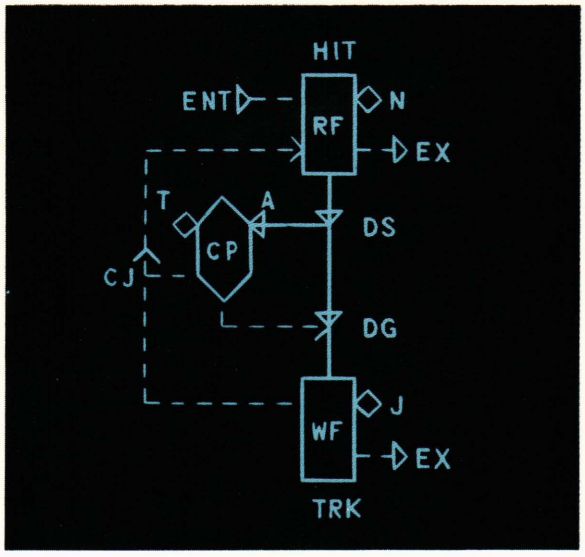
The next step is to enter the arrows marking the input end of each connection, as well as other auxiliary labels and symbols (Fig. 6b). Where data inputs are to be stored in permanent memory locations, the input is indicated by a diamond with a symbol denoting the variable. Where a data input requires preparation, such as extracting the amplitude A from the hit word (R, A), the input arrow is closed into a triangle.

In order to help him remember the data and control inputs to the different elements, the designer may type in appropriate symbols on the keyboard and place them on the diagram by means of the light pen. In Fig. 6b the file names "HIT" and "TRK" are indicated on the RF and WF elements, the number of hits "NHT" and the number of empty spaces in the TRK file "JTK" are indicated by the abbreviations "N" and "J," respectively, and the threshold and amplitude connections on the CP element are indicated by the symbols "T" and "A."

Figure 6b includes a CONTROL JUNCTION (CJ) element and a connecting link from it to the WF element that are not shown in Fig. 6a. Figure 6b also shows connections marked "EX" (Exit)



(a) Partially completed circuit.



(b) Completed circuit.

Fig. 6—Sample circuit displayed on the IBM 2250 terminal.

to the RF and WF elements. The appearance of these features illustrates how the GAP graphics program would discover formal errors or omissions by the designer in connecting the circuit elements. The computer examines each connection to see whether it has been assigned its proper function, i.e., data or control, input or output, and indicates omissions or incompatibilities by flashing or otherwise marking the connections involved. The designer would correct such errors before initiating the transformation of the circuit into computer code.

The data tables stored in the computer to generate the above circuit design on the 2250 could be converted by an automatic program into a table of logical connections represented by the circuit. A simpler approach, however, is to use the graphic routine initially for the sole purpose of remembering graphical elements, and *not* their logical connections. A connection within the circuit can then be entered by the designer after he is satisfied with the display by simply touching the light pen to each end of a link. This avoids the housekeeping overhead associated with remembering "bends" in lines representing links, and circumvents the necessity of encoding logic to handle deletions and additions of line segments.

Element Interconnection Matrix—The information concerning the configuration of the Data Flow Circuit entered by the designer is organized by the graphics display program into a table which will be called the Element Interconnection Matrix. The matrix for the circuit described in Figs. 6a and 6b is shown in Table 2. The first three columns contain the element name, reference number in the circuit, and code. The next six numbered columns are the labels of the terminations of each respective connection. Each connection which is linked to another element in the circuit is labeled with the code and connection number of that element. For example, the entry "DS1" in column 4 of row 1 means that connection #4 of RF is linked to connection #1 of DS. Since there may be several elements of a given type in a single circuit, in actual practice the labels would use the reference numbers instead of the element code letters.

In the above example, the reference number "4" of the DS element would be used instead of the code letters "DS" in labeling the link. The matrix entries also indicate when "Data Prepare" operations are to be performed on data inputs or outputs, referring to a separate list where each operation is specifically defined. Such an entry is made in column 2 of row 2, and defined in the note below the matrix.

Each connection in the matrix which is linked to an input and output from the circuit is designated by a label representing the respective file, variable, or control instruction. For example, the entry "NHT" in column 2 of row 1 stands for the data entry for the number of hits to be read out of the file named "HIT." These designations

TABLE 2
ELEMENT INTERCONNECTION MATRIX

Elements			*Linkages*						*Connections*					
Name	*Ref. No.*	*Code*	*1*	*2*	*3*	*4*	*5*	*6*	*1*	*2*	*3*	*4*	*5*	*6*
READ FILE	1	RF	HIT	NHT	EXH	DS1	CJ1	ENT	D	D	J	O	C	C
COMPARE	2	CP	——	DS3–P	——	DG3	CJ2	THR	U	D	U	J	J	D
WRITE FILE	3	WF	DG2	JTK	EXT	TRK	——	CJ3	D	D	J	D	U	J
DATA SPLIT	4	DS	RF4	DG1	CP2	——	——	——	D	O	O	U	U	U
DATA GATE	5	DG	DS2	WF1	CP4	——	——	——	D	O	C	U	U	U
CONTROL JUNCTION	6	CJ	RF5	CP5	WF6	——	——	——	J	C	C	U	U	U

P PREPARE: MASK A

are separately identified by the designer and recorded in a "Dictionary" accompanying the set of Data Circuits which will be combined into the total program.

The last column of the matrix designated "Connections" indicates the type of each connection, namely:

D Data Input
O Data Output
C Control Input
J Control Output
U Unconnected

The data in this column enable the program to make sure that an output always goes to an input and that each element has the appropriate type of connections.

Transformation of Graphical into Logical Form

The third key feature of Graphical Automatic Programming is the automatic transformation of the Element Interconnection Matrix, generated by the graphics terminal from the Data Flow Circuit, into the desired computer program. This requires the translation of the designated process represented by a two-dimensional circuit diagram into a one-dimensional sequence of computer instructions. The noteworthy facts are that this transformation can be done entirely automatically and that the resulting program is highly efficient in execution time and Core usage.

It will be recalled that in a Data Flow Circuit, as opposed to an ordinary digital circuit, signals flow in a succession of steps, each representing the transfer of a signal from an output of one element to an input of an adjoining element. The transformation of a Data Flow Circuit into an operational sequence involves putting these steps into an order which can be performed efficiently by a general-purpose computer. A remarkably simple set of rules produces a program that has high efficiency and is free from any evident pitfalls. These rules are listed in Table 3. They are divided

TABLE 3
CIRCUIT TRANSFORMATION RULES

Output Sequence Priority

1. Last Data Output
2. Last Control Output

Input Handling

Element Type	*Initial Data Input*	*Final Data or Control Input*
1. FUNCTIONAL	Put in Temporary Store	Write Function Code
2. DATA JUNCTION	Jump to Termination	——
3. CONTROL JUNCTION	——	Change Termination of Inputs to that of Output
4. DATA LOOP	Write Function Code	——

into "Output Sequence Priority" rules and "Input Handling" rules.

The logic behind the Sequence Priority rules is the following:

1. When an element produces a data output, this output is in a general register of the computer. It is more efficient to operate on this output while it is in a general register than to go to another operation, since this saves the steps of temporarily storing the output and later reloading it in a general register. Therefore, the element to which the output is connected is first examined to see if it is ready to operate.

2. When no data outputs remain, selecting the latest control output insures that any internal loop in the circuit will be closed inside of any larger loop. Entry inputs to the circuit are listed at the beginning of the program as control outputs of external circuit blocks.

The Input Handling rules are the following:

1. For the four types of functional element the rules for handling inputs were described earlier. They state that an element is activated to function by the arrival of the last input, whether a data or control signal, and that data inputs arriving previously are stored until needed.

2. The DATA JUNCTION element is equivalent to an "or" gate combining several alternative data signal inputs into a single output. This is effected in computer code by a "Jump" instruction to the point in the program to which the DJ element is connected, thereby joining the input paths.

3. The CONTROL JUNCTION element performs the same function for control signals. Since a control output is already a "Jump" instruction, this element does not require writing additional code, and is translated simply by changing the termination of each junction input to the termination of the junction output.

4. The DATA LOOP element has two data inputs—one direct and the other a feedback from the processed direct signal. Its function is to output the direct signal upon its arrival, and to hold the feedback signal until the arrival of a control feedback gating signal for processing the feedback signal through another cycle of the loop. It is translated into code in a similar manner as the READ FILE element, which also has a direct and feedback mode. This is done by placing the execution of the DL element in the Operational Sequence at the point where the direct input arrives. The function code is written to include both the direct and feedback mode with a label to identify the address of the "Jump" instruction from the feedback path.

Transformation of a Data Flow Circuit—The application of most of the above rules to the translation of a Data Circuit into computer instructions can be illustrated by going step by step through the process on the very simple circuit described in the previous section. This is shown in Figs. 7a, b, and c.

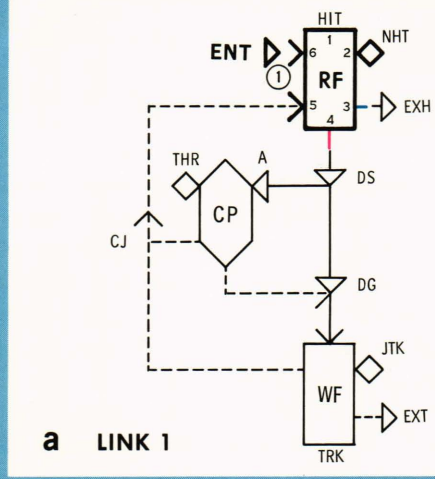
Figure 7a shows the first step in the transformation process. The process begins by tabulating and labeling the inputs and outputs connecting the circuit block to other blocks in the overall program. The external data inputs are: two connections to files—HIT and TRK, and three data word inputs—the file indices NHT and JTK, and the threshold THR. There are three external control inputs and outputs: one enter, ENT, and two exits, EXH and EXT.

It is helpful to assign a section of the computer program to the definition of the labels used in the instruction code. This section, usually called the "Linkage" or "Assembly" area, defines the labels used in referring to each external connection. The resulting assembly code, in DDP-516 language, is given in the top block at the right side of Fig. 7a. The instruction "DAC" stands for "Declare Address Constant" and serves to define the labels used in the assembly code for the circuit in terms of labels for variables and files defined for the overall program. The labels used for all memory locations are defined in terms of the element connections, as used in the Element Interconnection Matrix.

The Linkage section will later be used to connect the circuit block to others in the program. Other circuit blocks may be in a different section of the computer memory, and in the DDP-516 have to be addressed by the "indirect" memory address mode. An asterisk is used to indicate indirect addressing.

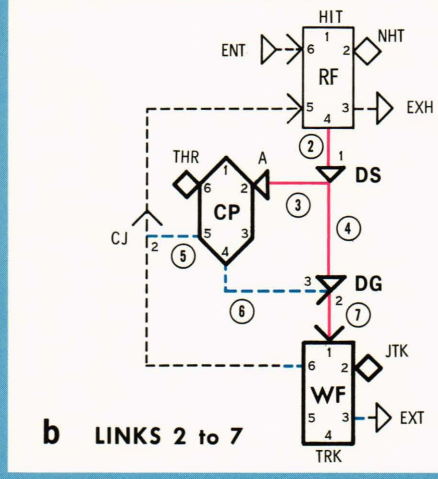
The listing of the external connections in the Linkage section produces one control output, namely from the Enter symbol to RF6. This and other outputs are listed in Fig. 7a at the right of the instruction block.

The first link to be made in the circuit is the



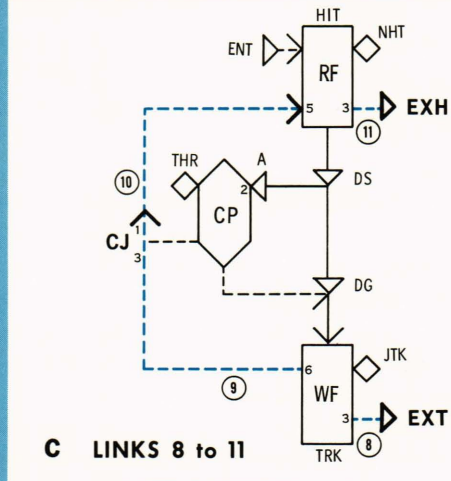
a LINK 1

	LABEL	INSTRUCTION		OUTPUTS
	RF1	DAC	HIT	
	WF4	DAC	TRK	
	RF2	DAC	NHT	
	WF2	DAC	JTK	
	CP6	DAC	THR	
	RF3	DAC	EXH	
	WF3	DAC	EXT	
	RF6	DAC	ENT	C - RF6 (1)
	RF5	IRS	*RF2	
		SKP		
		JMP	*RF3	C - RF3
(1)	RF6	LDX	*RF2	
		LDA	*RF1	D - RF4



b LINKS 2 to 7

	LABEL	INSTRUCTION		OUTPUTS
	RF5	IRS	*RF2	
		SKP		
		JMP	*RF3	C - RF3
	RF6	LDX	*RF2	
		LDA	*RF1	D - RF4 (2)
(2)		STA	DS1	D - DS2 (4)
				D - DS3 (3)
(3)		ANA	CP2	
		CAS	*CP6	
		JMP	DG3	
		JMP	DG3	C - CP4 (6)
		JMP	CJ2	C - CP5 (5)
(6)	DG3	LDA	DS1	D - DG2 (7)
(7)		LDX	*WF2	
		STA	*WF4	
		IRS	*WF2	
		JMP	*CJ2	C - WF6
		JMP	WF3	C - WF3



c LINKS 8 to 11

	LABEL	INSTRUCTION		OUTPUTS
	RF5	IRS	*RF2	
		SKP		
		JMP	*RF3	C - RF3 (11)
	RF6	LDX	*RF2	
		LDA	*RF1	
		STA	DS1	
		ANA	CP2	
		CAS	CP6	
		JMP	DG3	
		JMP	DG3	
(10)		JMP	RF5	
	DG3	LDA	DS1	
		LDX	*WF2	
		STA	*WF4	
		IRS	*WF2	
(10)		JMP	RF5	C - WF6 (9)
		JMP	*WF3	C - WF3 (8)
				C - RF5 (10)
	CP2	DEC	MKA	

Fig. 7—Hit Sorting Program.

connection of the above output to the READ FILE element. Since the index data input is already available in the Linkage, the RF element functions when the control input RF6 arrives. This step results in the following set of operations which transforms the circuit into computer code:

1. Write code for RF element in its gated form.
   List data output RF4.
   List control output RF3.

In accordance with the priority rules, the last data output, RF4, is chosen as link 2, the next step in the transformation. This and the subsequent five steps in the transformation of the circuit are illustrated in Fig. 7b. The number of each output selected, the corresponding link formed in the circuit, and the resulting block of code are shown by a circled number. The Linkage section of code is not repeated, for the sake of brevity.

The arrival of the data input to the DATA SPLIT element in this step is sufficient to cause it to function. This stores the input temporarily and produces two data outputs. The transformation results in the following operations:

2. Write code for DS element.
   List data output DS2.
   List data output DS3.

The last data output from Step 2 is DS3, and it is chosen as Step 3 in the transformation. This completes the inputs required for the COMPARE element to function, and results in the operations:

3. Prepare input CP2.
   Write code for CP element.
   List control output CP4.
   List control output CP5.

The data input to CP has to be prepared by extracting the amplitude, A. This is accomplished by a logical "and" (ANA) masking operation, which is included in the first instruction of the block of code written for the CP element.

According to the priority rules, the remaining data output, DS2, is operated on before the control outputs are. This becomes Step 4 in the transformation sequence. In order to function, the DATA GATE element requires the presence of the control input as well as the data input. Since the former has not yet arrived and the latter is already stored in a temporary location, DS1, the completion of link 4 does not result in any code but merely the entry of the temporary store label DS1 into the input of the DG element, and hence the operation:

4. Change label on DG1 to DS1.

At this point we note the distinction between a true electronic circuit and a GAP circuit. In an electronic circuit, signals in lines emanating from a branch are processed in parallel, whereas in most computers only one branch can be executed at a time. In GAP, a DATA SPLIT automatically stores the input as it arrives, then processes one branch, and later processes the other branches. The order of execution of elements connected to the branches of a DATA SPLIT is controlled by use of gates either on the elements themselves or, as in the sample circuit, by the use of the DATA GATE routing element. This element is used here so that the code for the COMPARE function will be written before the code for the WRITE FILE element.

The next output in priority is the last control output, CP5, and hence initiates Step 5. This output provides one of the two necessary inputs to the CONTROL JUNCTION element and results in no code.

Output CP4, Step 6, completes the required inputs to the DATA GATE. This step results in the operations:

6. Write code for DG element.
   List data output DG2.

Step 7 is to link data output DG2 to WF1, which causes the WF element to function. This results in the operations shown in the last block in Fig. 7b:

7. Write code for WF element.
   List control output WF3.
   List control output WF6.

The final figure of the series, 7c, shows the completion of the last four links in the circuit. Output WF6 causes the CJ element to function. The functioning of the CJ element completes the link to RF and simply changes the labels on the control outputs CP5 and WF6 from CJ2 and CJ3 to RF5. The remaining links do not write additional code since the exit connections were already included in the linkage section.

The last step is to define the label CP2 in the code for the CP element as the mask for extracting the amplitude A. This is done in the final assembly instruction in Fig. 7c. The notation DEC stands

for a decimal number. MKA stands for a number which, when translated into binary code, forms the mask for extracting A.

Element Operational Sequence—In Fig. 7 the Data Circuit was transformed directly into computer assembly code. In actual practice it is useful to divide this process into two steps.

1. Transformation of the Element Interconnection Matrix into an Operational Sequence.
2. Compilation of the Operational Sequence into Computer Code.

The first step is the really fundamental one, since it converts the two-dimensional matrix into a one-dimensional sequence. This is done by following the priority order of the circuit transformation rules. During this process some of the signal routing elements effectively disappear after establishing the sequence of operation of the functional elements and making direct interconnections among the functional elements themselves.

The result of this first step for the example discussed above is shown in Table 4. Comparing this table with Table 2 shows that the control junction has disappeared, and the connections are made directly between the functional elements to which it had been connected. The sequence of operations is given by the numbers in the third column.

The significance of the entries under each connection in the Operation Sequence is the following. The characters at the left designate the label of the memory location of a data-input or of the instruction for a jump. The status or location of each input (in memory or in a register) is designated by code letters defined below the table. This information, together with the basic definitions of each element in terms of the code for a particular computer, is sufficient to translate the Operational Sequence into computer assembly code.

The Element Operational Sequence has a form entirely independent of the computer for which the program is to be written. Thus it represents a set of high level or "macro" instructions, which can now be translated into the assembly code of any desired computer by a "compiler." This process involves a mechanical substitution of the code for each element in the operational sequence, with due regard for the mechanics of storing and retrieving data from temporary stores as indicated by the notation in the connection field, and the conversion of label notation to suit the format of the specific computer code being written.

The compact and universal form of the Element Operational Sequence means that this intermediate step in program design can be used to check the program logic using any computer code, including the one which drives the graphics terminal, such as the IBM 360. Thus, compilation of the IBM 360 version of the code enables the immediate on-line test of the entire logical design of the circuit and of its transformation into the sequence of operations. If this is successful, the program logic can be considered "debugged" for all practi-

TABLE 4
ELEMENT OPERATIONAL SEQUENCE

			Linkage/Status					
Ref. No.	*Code*	*Seq.*	*1*	*2*	*3*	*4*	*5*	*6*
1	RF	1	RF1/*	RF2/*	RF3/*	DS1/O	RF5/L	RF6/*L
2	CP	3	/U	RF4/AMP	/U	DG3/J	RF5/J	CP6/*
3	WF	5	DG2/AM	WF2/*	WF3/*	WF4/*	/U	RF5/J
4	DS	2	RF4/A	DG1/O	CP2/O	/U	/U	/U
5	DG	4	RF4/M	WF1/O	DG3/L	/U	/U	/U

Status:

*	Stored in Linkage Location	P	Data to be prepared	O	Data Output
A	Data in general register	L	Labelled Instruction	J	Control Output
M	Stored in Memory	C	Control Input	U	Unconnected

cal purposes, inasmuch as the conversion to the code for another computer involves no change in operational logic. This on-line debugging capability is an enormous advantage inherent in the use of the graphics terminal to effect direct interaction with the computer.

Integration and Testing of Complex Programs

A large data-processing program for a large-scale system can be represented as a Data Flow Block Diagram, in which each block is an individual Data Flow Circuit. Each Circuit Block can be regarded as a special "Macro" Circuit element, with data and control signal inputs and outputs connecting it to other blocks which comprise the total program. The integration of Data Flow Circuits is readily accomplished by the use of the graphics terminal and a special Integration Program in a manner similar to that used in constructing the Data Flow Circuits. This program serves the purpose of a "Linkage Editor" in computer terminology.

The representation of a Data Flow Circuit as a Program Block is shown in Fig. 8, using the Hit Sorting Program as a simple example. It is seen that the block has 8 connections, namely 4 data inputs, 1 data output, 1 control input, and

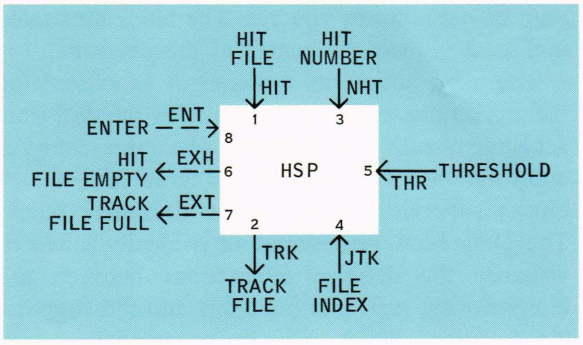


Fig. 8—Hit Sorting Program block.

2 control outputs. By reference to Fig. 7a it can also be seen that all of these connections are embodied in the Linkage Section of the code for the block. This section is equivalent to a terminal strip in a piece of electronic equipment.

The integration of program blocks into the total program is simply done by drawing the program Block Diagram on the graphics display and making proper connections between the individual blocks. In such a diagram, it is important to keep all files external to the processing circuits.

An example of such a diagram is shown in Fig. 9, which represents the Track Prediction module of the 3D Radar Automatic Tracking Program. The program blocks are represented by rectangles

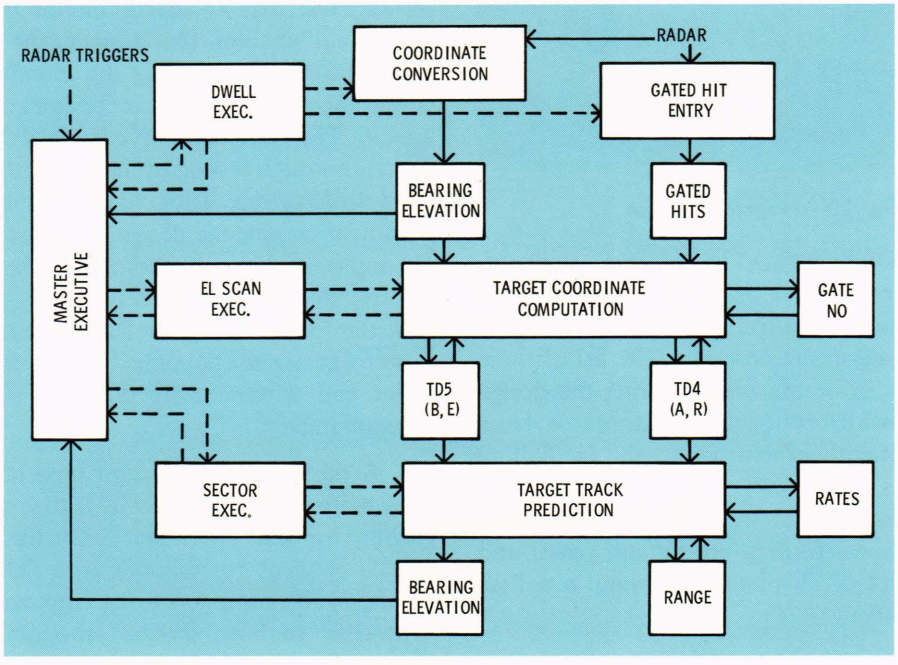


Fig. 9—Track Prediction module.

and the data files by squares. The block illustrated in Fig. 2 is near the center of the diagram.

The very important function of synchronizing the operations of the program with the real-time schedule of radar transmission, elevation scanning, and rotation is accomplished by three "Executive" blocks supervised by a master Executive block. The Data Flow representation is ideally suited to visualize the detailed interactions between the high-priority, real-time functions and the supporting functions which may be accomplished with loose scheduling.

The transformation of the Block Diagram into computer assembly code involves only the proper correlation of the block linkage labels, where all inputs and outputs are listed. Since the module is itself a "block," as seen in Fig. 10, the next higher level of program integration is done in terms of entire modules rather than blocks. In this way an orderly and flexible format for the total program can be achieved.

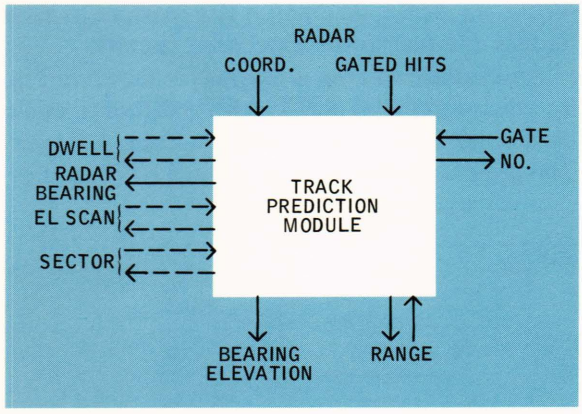


Fig. 10—Program module.

Program Dictionary—The efficient management of the process of program design requires careful definition, organization, and maintenance of all terms used in the program. The list of these properties has to be assembled during the design process and, when complete, constitutes a basis for fully documenting the program and facilitating future changes.

The data required for this list include the code names, definitions, format, constituent parts, and cross-references of all of the following in a Program Dictionary:

1. Variables and constants
2. Data Files
3. Circuit Blocks
4. Modules

In addition, areas of memory must be allotted to store each of the above.

Once the above terms have been defined, their subsequent use requires only reference by code name. The computer automatically looks up the necessary characteristics in the Program Dictionary. This saves a great deal of housekeeping by the system designer, and should eliminate a major source of error.

Program Checkout—The most laborious and time-consuming part of programming is the elimination of errors, or "debugging." It is in this area that Graphical Automatic Programming is likely to produce the greatest benefit in program design. The power of this technique to facilitate the production of a correct program stems from the following sources:

1. The Data Flow Circuit representation renders the pattern of data and logic flow highly visible and hence eliminates many errors at the source.
2. The entry of the Data Flow Circuit into the computer enables a virtually instantaneous check of any inconsistencies in the design by checking the Element Interconnection Matrix and monitoring its transformation into computer code.
3. The display of the circuit by the graphics terminal enables the designer to correct errors immediately by altering the circuit and verifying that the errors have been eliminated.
4. The GAP technique is ideally suited to rapid and thorough testing of the program at any desired level of realism. Thus, upon completion of a constituent circuit, the designer can test it by entering sample inputs and reading out the resulting outputs. He can also quickly design a test program in the form of another Data Circuit which would perform a realistic simulation of the program input and automatically compare the results with requirements.

A particularly important type of test that may be performed automatically is that of compatibility with real-time operation. Since the functioning of each element corresponds to a definite execution time in the computer to be employed, it is readily possible to have the test program simulate the execution time and monitor it against specified events.

Design of Data Flow Circuits

The representation of data-processing operations in the Data Flow Circuit form has turned out to have all of the characteristics which were sought for in a basic language for the programming of real-time systems. For those interested in how a given problem is translated by the designer into a Data Flow Circuit, the paragraphs below illustrate how one might design the circuit illustrated in Fig. 2.

The design of a particular Data Flow Circuit is best approached by constructing a "cause-effect table"—similar to a "truth table" or "decision matrix" in mathematical logic. This lists the possible combinations of input conditions and the corresponding outputs. For its application to the Target Coordinate Computation Circuit illustrated in the figure, the logic is as follows:

1. If no prior hit exists in the target data file (TD4), set number of hits = 1 and store coordinates of new hit in target data files (TD4, TD5).

2. If previous hit exists, but does not coincide in range with new hit, exit to multiple target routine.

3. If previous hit correlates in range, increment number of hits, store target coordinates of strongest hit.

This leads to the following cause-effect table in which A is amplitude, $\triangle$R is range increment within the gate, B is bearing, E is elevation.

From the cause-effect table it is evident that the following functional elements will be required.

1. READ FILE (RF) and WRITE FILE (WF) elements to read out data on new hits, previous entry, and to store updated coordinates.

2. CORRELATE (CR) element to check range correlation.

3. COMPARE (CP) element to compare amplitude of new and previous hit.

4. ADD (+) elements to increment the number of hits.

5. AVERAGE (AV) elements to combine bearing and elevation for hits of equal amplitude.

The table also helps to arrange these operations in an efficient order in the circuit. It is evident, for example, that the range correlation check should be made before amplitude comparison.

With the aid of this type of logical organization, the layout of a Data Circuit such as the one shown in Fig. 2 follows quite readily. The representation is intuitively easy to use by engineers, and makes it relatively simple to configure the routing elements to minimize the total number of instructions. For example, convergence of the data paths before storage of the updated coordinates is an obvious saving in code. This type of optimization is made much more visible by the Data Circuit representation than by the conventional sequential logic approach.

The Data Circuit language thus makes it possible for an engineer to design the data flow process, in a form with which he is intuitively familiar, to achieve the best balance between operational requirements on accuracy, capacity, and timing within the limitations of available computer speed and size. The language is also directly interpretable by a programmer, so that even without the automatic features it bridges the communication gap

Conditions (*Cause*)					
Previous entry in TD4, (A, ΔR)	None	Yes	Yes	Yes	Yes
ΔR of hit correlates with previous entry	—	No	Yes	Yes	Yes
Amplitude of hit compared to previous entry	—	—	Greater	Equal	Less
Actions (*Effect*)					
Number of hits	Set = 1	—	Add 1	Add 1	Add 1
Store in file TD4, (N, A, ΔR)	New hit	—	New hit	Previous hit	Previous hit
Store in file TD5, (B, E)	New hit	—	New hit	Average	—
Other	—	Exit	—	—	—

which currently represents one of the greatest impediments to the economical design of system "software."

Acknowledgment

In seeking to make the subject of this paper clear to all readers with a possible interest in its application, I obtained many valuable suggestions from R. P. Rich, W. H. Guier, W. N. Sweet, and J. R. Austin, which I gratefully acknowledge. R. R. Newton was particularly helpful in this regard. I wish also to acknowledge the help of D. M. White, who has been the first to actually use Data Circuits for designing SIMFAR—a major radar simulation program, and therefore has been in a position to demonstrate the utility of this concept. W. T. Pullin contributed his expert knowledge of computer graphics to the design of GAP symbols, with the result that they are both visually clear and easy to generate. W. T. Pullin and S. E. Anderson have developed a program for displaying GAP circuits on the IBM 2250 terminal.

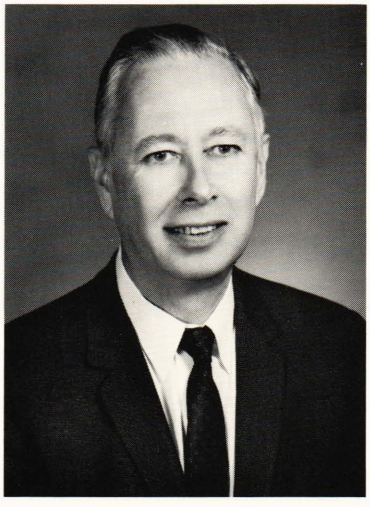
I should like most particularly to acknowledge the very significant contribution of Lee Hoevel to the development of the Graphical Automatic Programming concept. This very appropriate name and its acronym, GAP, are his ideas. As an expert programmer, he provided the first authoritative confirmation that the transformation of a Data Circuit to a computer program could in fact be carried out unambiguously and could produce efficient code. He has given much time after working hours to critical discussions of every aspect of the design of the language and to its presentation in this paper. Working with him has been a most enjoyable experience, and I look forward to collaborating with him in the further development and implementation of the Graphical Automatic Programming technique.

WITH THE AUTHOR

A. Kossiakoff, author of "Graphical Automatic Programming," is the Director of the Applied Physics Laboratory. Dr. Kossiakoff received a B.S. degree in chemistry from California Institute of Technology in 1936, a Ph. D. degree in chemistry from The Johns Hopkins University in 1938, and then spent a year as a post doctoral fellow at California Institute of Technology. He taught at The Catholic University of America (1939–42), then served with O.S.R.D., and was Deputy Director of Research at the Allegany Ballistics Laboratory, Cumberland, Maryland from 1944 to 1946.

Dr. Kossiakoff joined the Applied Physics Laboratory in 1946, and served as head of the Launching Group until 1948, when he was appointed Assistant Director. He became Associate Director in 1961, Deputy Director in 1966, and was appointed Director on July 1, 1969, when Dr. R. E. Gibson retired as Director of the Laboratory.

Dr. Kossiakoff has made important contributions to the development of advanced weapons systems



for the U.S. Navy, particularly in the design and implementation of guided-missile defenses for the fleet. From 1948 to 1951 he served as Chairman of the Panel on Launching and Handling of the Department of Defense Research and Development Board. In recognition of his work on national defense, Dr. Kossiakoff has been awarded the Presidential Certificate of Merit and the Navy's Distinguished Public Service Award.

Dr. Kossiakoff has been involved in systems engineering for many years. About two years ago he decided to learn the details of computer programming in order to get a better assessment of the problems involved in their rapidly growing applications to the automation of complex systems. He found this field so interesting that he devoted a good deal of his spare time to the design of a computer program for the automatic detection and tracking of aircraft by a three-dimensional radar. The ideas discussed in this paper stemmed from his experience with the difficulties of using existing techniques for the application of computers to complex real-time systems.

Dr. Kossiakoff is a member of the American Chemical Society, American Association for the Advancement of Science, The Philosophical Society of Washington, and the Cosmos Club.