E-2089

FINAL REPORT ON PHASE I OF CONTRACT NAS 12-140 DSR 55-27600

by

James Pennypacker

February 1967

GPO PRICE $ ______

CFSTI PRICE(S) $ ______

Hard copy (HC) ___

Microfiche (MF) ___

ff 653 July 65

N68-32102
(ACCESSION NUMBER) (THRU) 1
28
(PAGES) (CODE) 08
CR-96154
(NASA CR OR TMX OR AD NUMBER) (CATEGORY)
FACILITY FORM 602

INSTRUMENTATION LABORATORY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge 39, Mass.

RQ-49250

E-2089

FINAL REPORT ON PHASE I OF CONTRACT NAS 12-140 DSR 55-27600

by

James Pennypacker

February 1967

INSTRUMENTATION LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE 39, MASSACHUSETTS

Approved: Eldon C Hall Date: 3/16/67
ELDON C. HALL, DIRECTOR, DIGITAL DEV.
APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: David G. Hoag Date: 16 Mar 67
DAVID G. HOAG, ASSOCIATE DIRECTOR
INSTRUMENTATION LABORATORY

Approved: Ralph R. Ragan Date: 17 Mar 67
RALPH R. RAGAN, DEPUTY DIRECTOR
INSTRUMENTATION LABORATORY

ACKNOWLEDGMENT

This report was prepared under DSR Project 55-27600, sponsored by the Electronics Research Center, Cambridge, Massachusetts, of the National Aeronautics and Space Administration through Contract NAS 12-140.

The publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or conclusions contained therein. It is published only for the exchange and stimulation of ideas.

E-2089

FINAL REPORT ON PHASE I OF CONTRACT NAS 12-140 DSR 55-27600

ABSTRACT

This report summarizes the work performed under a NASA contract to develop a computer aided design facility. The aim of the computer aided design facility is to provide the individual designer of digital electronic circuits with a tool to facilitate the design process. Typically, schematic drawings are to be automatically generated and signal and wire lists are to be printed. The facility incorporates a central data processor in which the design data is stored in a central file as the design proceeds; the data is immediately available for immediate recall in different forms specified by the designer. The threaded list structure of the central data file is described, as are the methods of manipulating data within the central file. The capabilities and limitations of the developed facility are examined, and a critique of the system is included. The overriding limitation to the practicality of the developed facility is the requirement of entering data via punched cards; strong consideration should be given to the implementation of an input device such as a light pen and display surface.

by James Pennypacker
February, 1967

PRECEDING PAGE BLANK NOT FILMED.

TABLE OF CONTENTS

Section		Page
1	An Overview	7
2	Data Structure	8
3	Virtual Memory Structure	16
4	Utilization of the Data Structure	17
5	Critique of the Present File Structure	19

Appendices		
A	Examples and Descriptions of Macro Instructions	21
B	Schematic Drawing Program	27

PRECEDING PAGE BLANK NOT FILMED.

FINAL REPORT ON PHASE I OF CONTRACT
NAS 12-140 DSR 55-27600

1. An Overview

The computer aided design program is an attempt to provide the designer with a tool to facilitate the design process in four major areas. The most trivial improvement is to decrease to near zero the amount of time required for design modifications to be documented. The second goal of computer aided design is to centralize all the design data; specifically the design is stored in the memory of a central processor. Coincident with the reduction in time for implementing the design modifications, the designer always has available in the processor's memory the most up to date design information. The third aim of computer aided design is to allow the designer access to any subset of the design data in any of several different formats. For example, signal lists, wire lists, schematic drawings, mechanical drawings and the like are to be automatically provided to the designer upon his request; such outputs of course should reflect the latest design information. The final and probably the most significant aim of the computer aided design is to provide the designer a means of "experimenting" with his product as the design evolves. In the case of digital design, such experimenting might take the form of logical simulation, either at a macro or micro level.

The heart of any computer aided design system is the storage of the design information in the central processor; the practicality of computer aided design is contingent upon the characteristics of this central data file. First and foremost, the central file must be large enough to store all the required design information. Secondly, the central file must be organized so that the flow of information into and out of the central file is easily accomplished. The file structure must be flexible enough to accommodate different types of design data as specified by the designer; in other words, the central file must be made easy for the designer to use as an aid for his particular problems. The data file structure must be organized in such a manner that subsets of data can be readily retrieved by the designer depending upon the desired form of data presentation. Finally, the data stored in the central file must be capable of being processed by programs in the central processor; this capability must be provided if the designer is to be allowed to experiment with or manipulate the evolving design.

To illustrate the concepts under discussion a typical type of design problem will be discussed; this particular design has been used as a "straw-man" to test the capabilities and usefulness of central file. The model design consists of an arbitrary logical circuit design. For the sake of simplicity, the only logical elements used in the design are 3-input NOR gates. Ordinarily the logical circuit is sketched in schematic form on several different drawings with each drawing being assigned a unique identification. The logical design is illustrated on the schematic by a number of interconnections which identify how the terminals of the various gates are electrically connected. Typically, each different electrical interconnection is referred to as a signal and is given a name such as GROUND; thus by specifying the names of the signals on each terminal of each NOR gate, the entire logical design can be specified.

With just the simple design information described in the above paragraph stored in the central file, the designer can pose a number of meaningful problems. For example, the designer might want a listing of all gates which are tied to a certain signal. A more specific problem would be to identify those gates which were tied to two particular signals, perhaps with some specified assignment of NOR gate terminals. The designer may reasonably expect a listing of all signals or gates which appear on a specific drawing; conversely it may be that the designer would like to know on which drawing a particular gate will be found. The above examples are only typical of the types of problems confronting a designer; certainly the central file should be able to help in this type of activity.

2. Data Structure

Because of the importance of the structure of data storage, the major portion of the effort expended on the computer aided design program was focused upon the development of a working data file; this development included not only the organization of the data structure but also included programs to manipulate the data structure.

The requirements placed upon the data structure can be most easily satisfied if the data file is a list-type structure. In such a structure, data elements are not necessarily stored consecutively in memory; associated with each data element is a "pointer" or address of where the next data element is stored. The advantages of this type of structure will be come obvious as the details of the data structure are described. To facilitate the description of the data structure a number of terms must be defined and described. The data structure is stored in virtual memory which appears to the program to be an independent memory of some two million locations or addresses. Addresses in virtual memory are numbered consecutively, starting with address zero. A pointer is an address in virtual memory at which a desired piece of design data which is to be stored in the central file. In terms of the design model previously described, values stored in the file would include signal names, drawing identifications, NOR gate identification, and drawing co-ordinates of a particular gate.

Values may also include remarks, such as the statement: "This NOR gate has been subjected to a severe overload". In general, a value is any piece of information which the designer wishes to store and to retrieve at a later date.

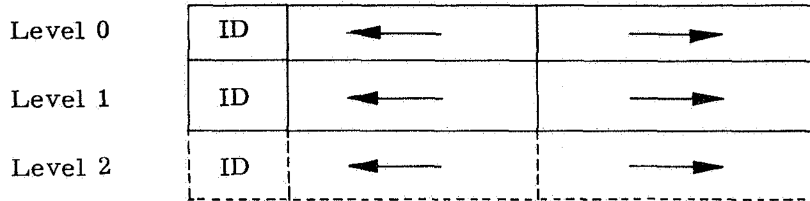
Associated with each value are either four or six pointers; this association of pointers is termed a facet and is depicted in Fig. 1. As indicated in the figure, a facet is composed of two or three levels, each of which contains an ID, a left pointer and a right pointer. A string is a grouping of facets which are interconnected by pointers; in general, the right pointer of level 1 points to level 1 of the next facet in the string while the left pointer of level 1 points to the previous facet of the string. Certain design data are actually classes of data; e.g., the list of signal names is composed of many signals. If a value is composed of subsets of data, the facet associated with the value contains a level 2. The level 2 pointers point to the first and last facets on the string which contains the subset of data; the facet with the level 2 pointers is known as the head facet of the string. The right pointer of level 0 of a facet always exists and points to the value of the facet. The structure of the facet and the purpose of the pointers is illustrated in Fig. 2.

To facilitate terminology in a later discussion, it is convenient at this time to define a super head facet of string X as the head facet of the string containing the head facet of string X.

The ID serves only to identify each level within the facet and whether or not a level 2 exists.

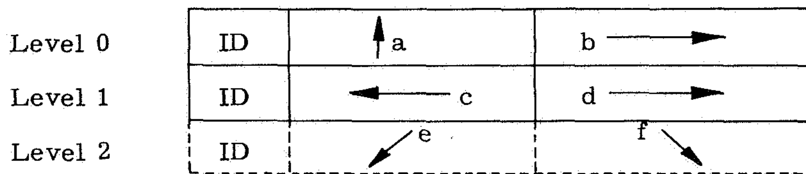
All of the data which is stored in the central file is stored via the threaded list structure described above. It is to be noted that the threaded list structure implies a hierarchy of classes of data which is defined by the position of the string on which data is threaded. The entire concept of this form of threaded list structure can best be understood through the use of an illustration; let us construct a simple file for the logical design of a digital processor.

The highest class of data to be stored is the entire file; therefore a facet whose value is "FILE" and which contains a level 2 is created. This facet is often referred to as the top of the file. The design information to be stored includes types of logical components, schematics and signals to interconnect the components. To store this information three facets are constructed and connected by their level 1 pointers; the string is closed through level 2 of the FILE facet. Because these facets are connected to level 2 of the FILE facet, it is often stated that they are strung immediately below the top of the file. The values of the three newly created facets are TYPE, DRAWING and SIGNALNAME and each facet will contain a level 2. In our simple example, two types of logical components, NOR and AND, will be used; thus two new facets must be constructed such that they are connected by their level 1



Pointers indicated by arrows.

Fig. 1 Facet structure.



a: Pointer to head facet.

b: Pointer to value associated with this facet.

c, d: Pointers to previous and next facets on this string.

e, f: Pointers to last and first facets on string immediately below this facet.

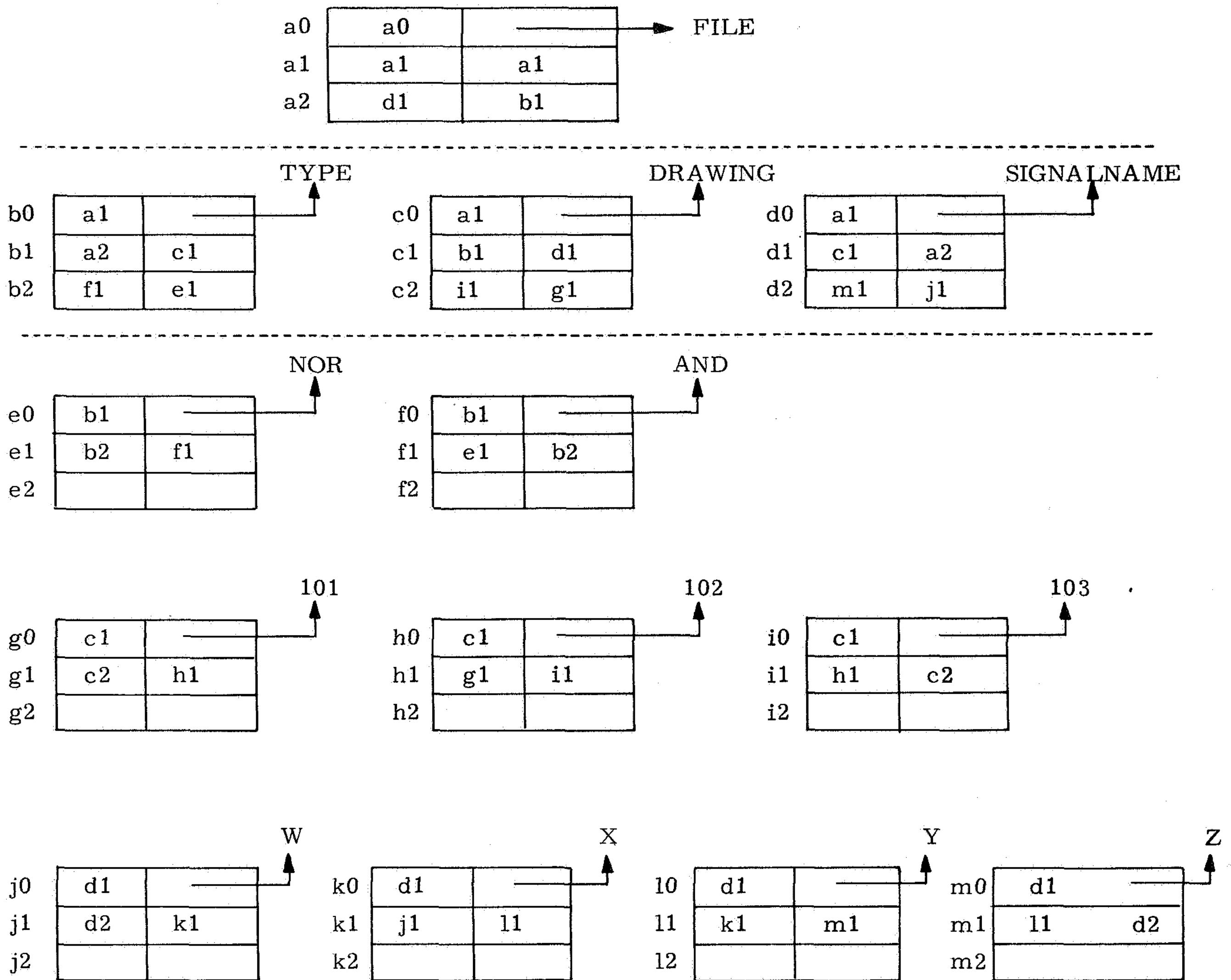
Fig. 2 A general facet.

pointers and the level 2 pointers of TYPE. Similarly, let us have three drawings, numbered 101, 102, 103 and four signals identified by W, X, Y, and Z. The resulting file structure except for the ID characters is detailed in Fig. 3 and presented in conceptual form in Fig. 4.

Figures 3 and 4 illustrate how classes of information are stored in the data file but it remains to be shown how the logical design is to be stored. Consider for example the design information pertaining to a particular NOR gate with three inputs and one output. To distinguish this NOR gate from all the other NOR gates it will normally be identified by a number e.g., 37. The gate will appear in some drawing, say drawing 102, at some co-ordinate position, u, v. The inputs and outputs of the gate will be connected to the terminals of other gates; this logical interconnection can be specified by giving names to the signals which appear on each of the gates terminal. For illustrative purposes, let the signals on input 1, input 2, input 3 and the output be called W, X, Y, Z respectively.

The above data indicates that for a NOR gate in our design, six classes of information are required and the design is completely specified by a particular value for each of the information classes. To store this design information in the data file, a <u>bead</u> is constructed, a bead being simply a contiguous grouping of facets. For the NOR gate in our illustration, each bead will contain six facets, one facet for the particular value of each class of information. The position of each facet within the bead is used to imply the class of information being represented by the facet's value; this convention is merely for convenience in file processing and does not restrict the generality of the file structure. A bead for a general NOR gate is illustrated in Fig. 5 and the bead for the particular nor gate described above is shown in Fig. 6. Note that none of the facets in the bead contains a level 2; because the values represented by the bead facets in this case are not classes of data, there is no string below any of these facets.

As implied in Fig. 6, the strings passing through the facets in a bead also pass through facets in other beads. Thus all design components which have a common property - e.g., all components appearing on drawing number 102 - are tied together. It is to be emphasized that in general each string passing through a bead is independent of all other strings passing through the bead. While the file thus becomes a maze of strings, it never becomes hopelessly tangled; by following a string all the information of the class represented by the string can be recovered. To recover information from the file, three different types of procedures exist. To recover information from a string it is only necessary to follow the pointers of the string and extract the value from every facet. To change strings, one can move from a level 1 of a facet to level 2 of the same facet (assuming a level 2 exists) and vice versa. It is also possible to change strings by moving from one facet within a bead to another facet within the same bead.



a0
a1
a2
b0
.
.
.

} Addresses in virtual memory.

FILE, TYPE etc. are values.

Figure 3.

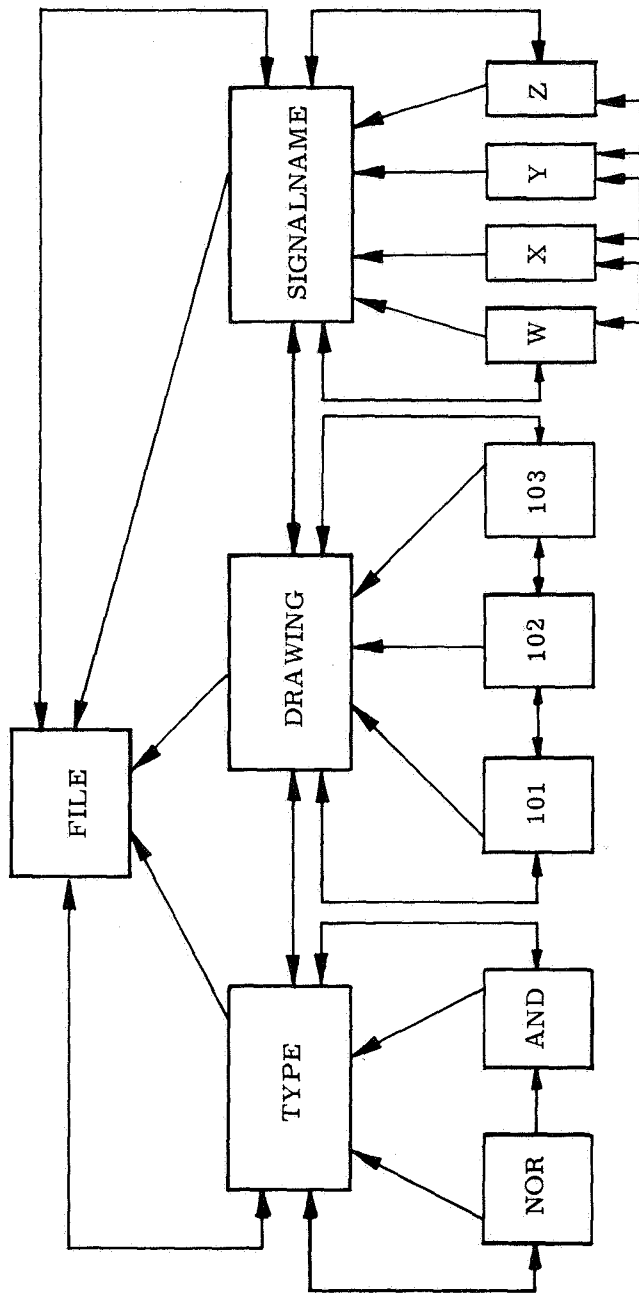


Figure 4

Facet identifying type

Facet naming the signal on Input 1

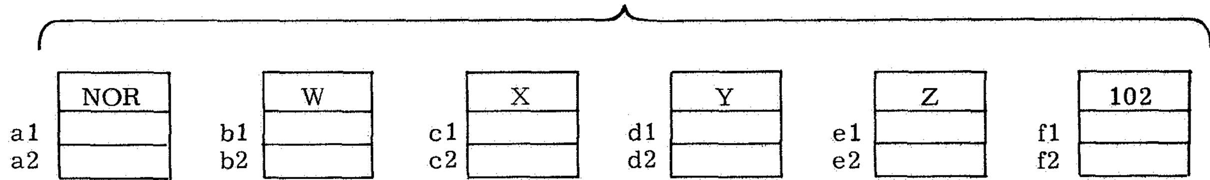
Facet naming the signal on Input 2

Facet naming the signal on Input 3

Facet naming the signal on Output

Facet identifying drawing

Fig. 5. General bead structure for NOR gate.



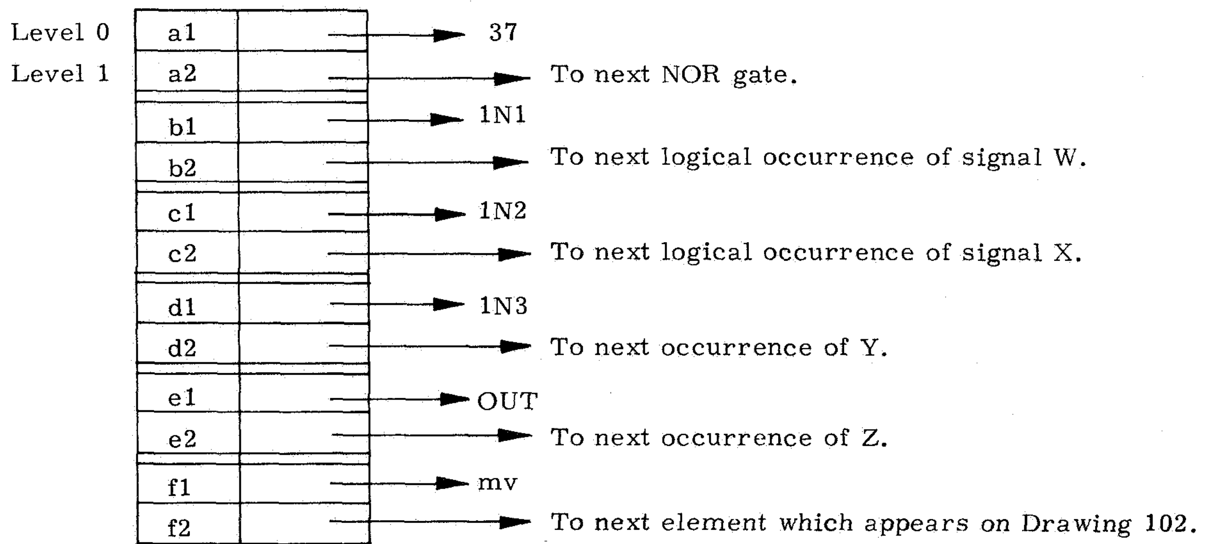


Fig. 6. Detailed structure for bead described in text.

One significant refinement to the data structure previously described is that a number of values can be associated with any facet. The resulting data structure is sufficiently general to store the required data for a large class of design problems. It is, of course, possible to recover the data by many different procedures and to manipulate this data to produce wire lists, schematics, etc. Approximately fifty macro instructions which are used to construct and manipulate the data structure have been programmed in a combination of Minneapolis-Honeywell 1800 assembly language and MAC, a higher order language similar to FORTRAN. Examples of these macro instructions and a description of their operation is included in Appendix A.

3. Virtual Memory Structure

The utilization of pointers to locate information in the file requires that in terms of the pointers the data be directly addressable. Because of various practical considerations - specifically the size of the file and the organization of the MH-1800-it is necessary to simulate a virtual memory in which the design data can be stored. To the file structure, the virtual memory looks like two million consecutive words starting with address zero. Each word in virtual memory is directly addressable simply by giving the number of the desired word relative to address zero. In terms of the file structure which has been described, a word of virtual address contains either one level of a facet or one value.

Because of the size of the virtual memory it is not possible to store the entire data file in the high speed core memory of the MH-1800; rather most of the file is at any point in time stored on a bulk storage disc memory. To facilitate the flow of data between the high speed core and the disc, the virtual memory is divided into 32,768 pages, or sections, each of which contains 64 words. During the running of the programs which utilize the file, a maximum of fifteen different pages of virtual memory is available in the high speed core. A virtual memory supervisor records which pages are in core and the locations where the pages are stored. An "age" is assigned to each page depending upon the usage of each page; that page which was most recently addressed is assigned the youngest age. As a running program requests an address in virtual memory, the supervisor identifies the page on which the page is found and locates the page in memory. If the requested address is currently in core, the running program has immediate access to the data file and the ages of the pages are updated to reflect the most recent request. When the requested address is not in core, the supervisor locates that page in core with the highest age, stores this page on the disc, locates the requested page on the disc, stores the page in core and updates the ages of the pages in cores. It should be clear that the use of ages to control the transfer of pages to and from disc is one of several algorithms which can be used to minimize the flow between disc and core, a goal which is dictated by the amount of time lost during such transfers.

4. Utilization of the Data Structure

The initial intent of the computer aided design program was that the various users would write their own specific programs which would utilize the file structure as desired. As the file developed, it became apparent that a number of specific file functions would facilitate the utilization of the file for the user. Consequently, several different file manipulations have been developed, each of which can be called by a single punched card; in general, these punched cards in addition to data cards provide the only set of input cards required for data runs. The set of functions which has so far been developed is described below; because the description is intended to be of a conceptual nature, the detailed format of the punched cards is not explained.

The card "NEWFILE Name" creates an initial data file whose top facet has the value "Name". This value serves to distinguish this particular data file from every other data file currently in the memory of the central processor; it is thus necessary that "Name" be unique. Also provided by this function is a list of working space which can be used only by this particular file and which can be expanded as required. As far as the user is concerned, this function creates and stores in virtual memory a facet with levels 0, 1, and 2; all the user's data will be strung below the level 2.

The instruction "WHATFILE Name" must be the first input card of every user's job which requires the use of an existing file. "Name" locates the file and identifies those portions of the entire data file which will be required by this run. The function essentially prepares the data file for the following data cards.

The instruction "FOLLOWPATH A, B, C, ..." is used to traverse the data strings within the file. The operation of the instruction is as follows:

1. The string passing through level 2 of the top facet of the file is located.
2. Each facet on the string is examined until the facet with value A is found.
3. The string passing through level 2 of the facet containing value A is inspected until a facet containing value B is found.
4. The above process is repeated for each value on the input card.
5. The instruction terminates after locating (and remembering) level 1 of the facet containing the last value on the input list.

The effect of this instruction is to locate a string which threads a particular class of data by following strings with specified names.

The instruction card "CREATESTRING Class" must be preceded by a FOLLOWPATH instruction. The result of CREATESTRING is to create a facet with level 2 and with value "Class". Level 1 of this facet is inserted into the string which passes through level 2 of the facet resulting from FOLLOWPATH. The instruction

essentially creates a new class of data as a subset of an existing class of data and is used primarily as an aid in creating a file.

The instruction card FORMAT must be immediately followed by data cards with data punched in a specific order. The FORMAT instruction defines the structure of a specific type of bead, the type being specified on one of the data cards. Also specified by the data cards are the number of facets to be included in the bead, the number of levels of each facet, and the FOLLOWPATH instructions required for each facet's super head facet. (See page 9). The format card with its data cards must be included in the input data deck before any beads of the specified format are utilized; the instruction need be given only once for each type of bead as a model of the bead is created and permanently stored for future reference.

The actual process of storing design information in the file can be accomplished with an ADDTOSTRING instruction card followed by appropriate data cards. The instruction creates a bead of the requested type (using the information previously entered via the FORMAT card) ties the bead into the file structure and inserts values for each fact. The head facets for each facet of the bead are specified on one of the data cards; values which are to be associated with each facet are included on another data card. At the present time, then, the bead is inserted in its entirety. Data cards for as many beads as desired may follow one ADDTO STRING instruction, a feature included to reduce unnecessary card punching by the user.

The current method of easily extracting design data from the file is limited in scope but quite flexible in application. The instruction GENERALOUTPUT is designed to deliver information to a user's program for specific processing. Before this instruction can be used, the user must have located a "starting point," i.e., a level 1 of some facet; this can be accomplished via a FOLLOWPATH instruction. Following the GENERALOUTPUT instruction are data cards which specify from which facets of a bead the values are to be extracted. Also included in the data cards is a number specifying the number of hierarchy levels below the "starting point" from which data is to be extracted. For every string on this level which is eventually threaded upward to the "starting point," every bead is examined and the data extracted from the specified facets. The data is of course delivered to the user's program in a pre-defined format.

The preceding instructions have been used to produce two types of useful design documentation; lists of signals and their terminals and schematic drawings. The programs developed for producing these outputs are typical of those which a user would be expected to provide when using the file for his own purposes. The signal list program is trivial in concept, consisting principally of logic to control the format of the printed page; the data is used directly as delivered by the GENERALOUTPUT instructions. The schematic drawing program, on the other hand, includes a fair

amount of logical processing, utilizing the delivered data to generate schematic drawings on an off-line plotter. An algorithm is included in the program to supervise the drawing of interconnection lines between the logical elements such that lines and gates are not overlaid. A description of the schematic drawing program is included in Appendix B. Obviously, for a user to generate such a program requires a detailed knowledge of programming; there appears to be no way out of the dilemma that the data file is intended to be versatile tool for designers who are not also proficient programmers.

5. Critique of the Present File Structure

The data file structure which has been developed and implemented is versatile in the types of design data which can be stored and retrieved in various forms. However, there are a number of drawbacks to the system as presently implemented. The most significant drawback is the excessive amount of computer time which is required to manipulate the data structure. The large amount of computation time required can be attributed to three causes: the language employed in programming, the use of pointers to retrieve data and the logical organization of the developed routines. There is no method of determining the allocation of time delay to these causes, but it is felt that the delay caused by program organization is insignificant, that the utilization of pointers causes a definite but minor delay, and the programming language is responsible for over 95 per cent of the delay experienced. The language used, MAC, is a FORTRAN like language with no capability for handling aphabetic information. The problems introduced by using this language for developing the file are numerous; its use was justified because the language is easy to use. By using MAC, with the full foreknowledge of its limitation, it was possible to develop a complete and sophisticated file structure; an accomplishment that a more basic machine language would not have permitted in a short period of time.

A second limitation of the existing file is that the data format must be punched on cards; the result is a clumsy data format that can only be changed as other input devices become available. Part of the difficulty arises, however, because of the threaded structure of the file; for each data value the string on which it is to be placed must be specified.

Because of the mechanization of the file on the MH-1800, the data is stored in an inflexible and undesirable format. Because of the ID characters, only 42 bits of the 48 bit word length are available for data storage. As a result, the data cannot be processed in its stored form by the central processor (by MAC). This limitation is conceptually insignificant but the problems of practical implementation are such that they should be avoided at all costs in any future development.

To expand the capabilities of the file currently requires a major effort of programming in both MAC and 1800 basic language. The file structure itself and the

manipulating instructions cannot be readily modified. A file developed in a higher language and which can be systematically modified by a few instructions is greatly to be desired. Similarly, a simple language which the file users could employ in handling their specific problems is an absolute necessity if the file structure is ever to be commonly accepted as a useful tool.

The final major drawback to the current system is that the system is custom designed for an MH-1800 computer and therefore is not universally available. This restriction also has catastrophic implications when and if the MH-1800 machines are replaced. The obvious solution to this problem is to develop the system completely in a universally available language; such a solution would also alleviate the problems described in the previous paragraph.

APPENDIX A

EXAMPLES AND DESCRIPTIONS OF MACRO INSTRUCTIONS

IN THE FOLLOWING DESCRIPTIONS, THE WORD VAD REFERS TO VIRTUAL ADDRESS.

LINKLFT

CALL LIST IS X.
W IS BINARY VAD (21 BITS), RIGHT JUSTIFIED.
ROUTINE EXTRACTS LEFT POINTER FROM CONTENTS OF X.
RETURN LIST IS LINK, WHERE LINK IS 21 BIT VAD, RIGHT JUSTIFIED.

LINKRGT

CALL LIST IS X.
X IS BINARY VAD (21 BITS), RIGHT JUSTIFIED.
ROUTINE EXTRACTS RIGHT POINTER FROM CONTENTS OF X.
RETURN LIST IS LINK, WHERE LINK IS 21 BIT VAD, RIGHT JUSTIFIED.

READLFT0

CALL LIST IS A. A IS VAD IN BINARY (21 BITS), RIGHT JUSTIFIED

ROUTINE READS THE LEFT POINTER FROM LEVEL 0 OF THE FACET CONTAINING VAD A. A MUST BE PART OF AN EXISTING FACET AND MUST NOT CONTAIN VALUES OR REMARKS.

RETURN LIST IS B WHERE B IS THE LEFT POINTER IN BINARY (21 BITS), RIGHT JUSTIFIED.

READLFT1

CALL LIST IS B
B IS VAD IN BINARY (21 BITS), RIGHT JUSTIFIED.

ROUTINE READS THE LEFT POINTER FROM LEVEL 1 OF THE FACET CONTAINING VAD B. B MUST BE PART OF AN EXISTING FACET AND MUST NOT CONTAIN VALUES OR REMARKS.

RETURN LIST IS LINK, WHERE LINK IS LEFT POINTER FROM LEVEL 1 IN BINARY FORM (21 BITS), RIGHT JUSTIFIED.

WRITERGT1

CALL LIST IS A, B
+, B, ARE EACH VAD IN BINARY (21 BITS), RIGHT JUSTIFIED.

ROUTINE WRITES A INTO LEVEL 1 OF FACET CONTAINING B. A IS WRITTEN INTO FACET AS RIGHT POINTER OF LEVEL 1. B MUST BE PART OF AN EXISTING FACET AND MUST NOT CONTAIN VALUES OR REMARKS.

THERE IS NO RETURN LIST.

WRITERGT2

CALL LIST IS A, B
+, B ARE EACH VAD IN BINARY (21 BITS), RIGHT JUSTIFIED

ROUTINE WRITES A INTO LEVEL 2 OF FACET CONTAINING B. A IS WRITTEN INTO FACET AS RIGHT POINTER OF LEVEL 2. B MUST BE PART OF AN EXISTING FACET AND MUST NOT CONTAIN VALUES OR REMARKS. IF LEVEL 2 OF THE FACET DOES NOT EXIST ALREADY, THE ROUTINE WILL CAUSE A PROGRAM EXIT.

THERE IS NO RETURN LIST.

SINK

CALL LIST IS A, A VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED.

ROUTINE OPERATES AS FOLLOWS.

1. IF A IS LEVEL 1 AND THERE IS NO LEVEL 2, ROUTINE RETURNS A MAC CONTROL CHARACTER = -1 AND VAD A.
2. IF A IS LEVEL 1 AND THERE IS A LEVEL 2, ROUTINE RETURNS A MAC CONTROL CHARACTER = 0 AND VAD OF LEVEL 2.
3. IF A IS LEVEL 2, ROUTINE RETURNS A MAC CONTROL CHARACTER = -2 AND VAD A.
4. IF A IS LEVEL 0 OR CONTAINS REMARKS OR VALUES, ROUTINE CAUSES A PROGRAM EXIT.

RETURN LIST IS K, VAD. K IS CONTROL CHARACTER IN MAC FORM AS EXPLAINED ABOVE. VAD IS IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED.

BUBBLE

CALL LIST IS A, A VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED.

ROUTINE OPERATES AS FOLLOWS.

1. IF A IS LEVEL 2, ROUTINE RETURNS A MAC CONTROL CHARACTER = 0 AND VAD OF LEVEL 1.
2. IF A IS LEVEL 1, ROUTINE RETURNS A MAC CONTROL CHARACTER = -1 AND VAD A.
3. IF A IS LEVEL 0 OR CONTAINS REMARKS OR VALUES, ROUTINE CAUSES A PROGRAM EXIT.

RETURN LIST IS K, VAD. K IS CONTROL CHARACTER IN MAC FORM AS EXPLAINED ABOVE. VAD IS IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED.

CREATEBEAD

CALL LIST IS BEADSIZE, FACETSIZE, STRING, LAVS

BEADSIZE IS THE NUMBER OF USABLE REGISTERS DESIRED IN BEAD, A MAC INTEGER WHICH MUST BE GREATER THAN 2.
FACETSIZE IS THE NUMBER OF THE HIGHEST LEVEL REQUIRED FOR THE FIRST FACET OF THE BEAD, A MAC INTEGER. THUS FACETSIZE MAY TAKE THE VALUE 2 OR 1 ONLY, FOR A FACET WITH OR WITHOUT A LEVEL 2 RESPECTIVELY.
STRING IS THE VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF A REGISTER WHICH IS ALREADY ON THE STRING TO WHICH THIS BEAD IS TO BE ATTACHED.
LAVS IS THE VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF LEVEL 0, 1 OR 2 OF THE LAVS HEADBEAD.

ROUTINE CREATES A BEAD OF THE REQUESTED SIZE, PROVIDING THAT THERE IS SUFFICIENT SPACE IN LAVS AND THAT THE REQUESTED SIZE IS GREATER THAN 2. IF EITHER OF THESE CONDITIONS IS VIOLATED, A PROGRAM EXIT WILL RESULT. THE FIRST REGISTERS OF THE BEAD ARE MADE INTO A FACET, WITH OR WITHOUT A LEVEL 2 AS SPECIFIED BY FACETSIZE. LEVEL 0 WILL ALWAYS EXIST. THE LEFT POINTER OF LEVEL 0 WILL POINT TO LEVEL 1 OF THE STRING HEADBEAD. THE RIGHT POINTER OF LEVEL 0 WILL ALWAYS POINT TO ITSELF. LEVEL 1 WILL ALWAYS EXIST (IF FACETSIZE DOES NOT EQUAL 1 OR 2, THE PROGRAM CAUSES AN EXIT) AND THE POINTERS WILL POINT TO THE PREVIOUS AND NEXT INSTANCE OF THE STRING. IF LEVEL 2 EXISTS, BOTH LEFT AND RIGHT POINTERS POINT TO THEMSELVES. THE FACET IS STRUNG AS THE FIRST INSTANCE ON THE STRING AFTER THE HEADBEAD. THE REMAINING REGISTERS OF THE BEAD WILL HAVE AN ID OF 20 WITH RANDOM LEFT AND RIGHT POINTERS.

RETURN LIST IS ADR, THE VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF LEVEL 1 OF THE STRING HEADBEAD.

FACETN
CALL LIST IS ADR, N.
ADR IS A VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF ANY REGISTER IN THE STRUCTURE OF THE BEAD.
N IS A MAC INTEGER, GREATER THAN 0.

ROUTINE LOCATES THE NTH FACET OF THE BEAD CONTAINING ADR. IF THERE

IS NO NTH FACET IN THE BEAD, THE ROUTINE CAUSES A PROGRAM EXIT.

RETURN LIST IS LEVEL 1, THE VAD IN BINARY (21 BITS) FORM, RIGHT
JUSTIFIED, OF LEVEL 1 OF THE NTH FACET.

DELETESTRING

CALL LIST IS ADR, LAVS.

ADR IS THE VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF ANY
REGISTER THREADED BY THE STRING WHICH IS TO BE DELETED. ADR
MUST BE EITHER A LEVEL 1 OR A LEVEL 2.

LAVS IS THE VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF LEVEL 1
OR 2 OF THE LAVS HEADBEAD.

ROUTINE PERFORMS THE FOLLOWING FUNCTIONS.

1. ALL LEVEL 0S OF THE STRING ARE SET TO EMPTY. ALL VALUES
AND REMARKS POINTED TO BY LEVEL 0S ARE SET TO EMPTY.
2. ALL LEVEL 1S OF THE STRING ARE SET TO EMPTY.
3. ALL LEVEL 2S OF THE STRING ARE SET TO EMPTY AND ALL FACETS
STRUNG BELOW ANY LEVEL 2 ARE SET TO EMPTY.
4. IN THE FACET WHERE THE STRING PASSES THROUGH LEVEL 2,
I.E. THE STRING HEAD FACET, THE LEVEL 2 POINTERS ARE SET
TO THEMSELVES BUT LEVELS 0 AND 1 REMAIN UNCHANGED.
5. IF ANY FACET WHICH WAS SET TO EMPTY IS THE FIRST FACET OF
ITS BEAD, THE ENTIRE BEAD IS RETURNED TO LAVS.

THERE IS NO RETURN LIST.

INTERSECT

CALL LIST IS SIZE, N_1 , $LIST_1$, N_2 , $LIST_2$, ... N_{SIZE} , $LIST_{SIZE}$, WRITEINDEX

SIZE IS A MAC INTEGER SPECIFYING THE NUMBER OF LISTS WHICH ARE TO BE
INTERSECTED. SIZE MUST BE AT LEAST 2 AND MUST BE SMALLER THAN 10.

N_I IS THE MAC INTERGER SPECIFYING THE NUMBER OF ENTRIES IN THE ITH LIST.

LIST IS A MAC INTEGER SPECIFYING THE MAC DATAFILE READ COUNTER FOR THE
BEGINNING OF THE ITH LIST.

WRITEINDEX IS A MAC INTEGER SPECIFYING THE MAC DATAFILE LOCATION INTO
WHICH THE INTERSECTION IS TO BE WRITTEN.

THE ROUTINE FORMS A LIST OF THOSE ENTRIES WHICH ARE COMMON TO ALL THE
LISTS SPECIFIED IN THE CALL FILE. THIS LIST OF INTERSECTIONS IS WRITTEN
INTO THE MAC DATAFILE COMMENCING AT THE LOCATION SPECIFIED BY
WRITEINDEX. EACH OF THE LISTS OF THE CALL FILE MUST BE ARRANGED SUCH
THAT ITS ENTRIES ARE IN ASCENDING NUMERICAL ORDER. THE INTERSECTION
LIST WILL ALSO BE IN ASCENDING ORDER.

RETURN LIST IS N, A MAC INTEGER SPECIFYING THE NUMBER OF ENTRIES WHICH ARE WRITTEN INTO THE INTERSECTION LIST.

GIVEDEPTH

CALL LIST IS ADR, A VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF LEVEL 0, 1 OR 2 OF ANY FACET IN A BEAD.

ROUTINE FINDS THE NUMERICAL POSITION OF THE FACET CONTAINING ADR IN ITS BEAD. THE FIRST FACET IN A BEAD IS NUMBER 1, THE SECOND FACET IS NUMBER 2 ETC.

RETURN LIST IS DEPTH, A MAC INTEGER EQUAL TO THE POSITION OF THE FACET CONTAINING ADR.

EXTRACTVALUE

CALL LIST IS X.
W IS BINARY VAD (21 BITS), RIGHT JUSTIFIED.
ROUTINE EXTRACTS VALUE FROM CONTENTS OF X.
RETURN LIST IS VALUE, WHERE VALUE IS 42 BIT WORD, RIGHT JUSTIFIED.

ORDERSTRING

CALL LIST IS ADR, A VAD IN BINARY (21 BITS) FORM, RIGHT JUSTIFIED, OF AN INSTANCE OF THE STRING.
ROUTINE ORDERS ALL FACETS ON THE STRING ACCORDING TO THE FIRST 7 CHARACTERS (42 BITS) OF THE VALUE OF THE FACET.

THERE IS NO RETURN LIST.

PRECEDING PAGE BLANK NOT FILMED.

APPENDIX B

SCHEMATIC DRAWING PROGRAM

This program draws logical schematics off-line on a Calcomp plotter. The plotting parameters are calculated by a general output program called DRAWSCHEMATIC which is called as a subroutine by the threaded-list manipulating program, which also requires the existence in the file of data specifying a drawing - for each gate, input and output signal names, position on the schematic, and a drawing number are required.

When a control card called *DRAWSCHEMATIC is read, the file-manipulating logic searches all data bearing the specified drawing number, and looks for notches which indicate a connection is to be made on the drawing.

The drawing routine is given first the locations of all gates, and then point pairs to be connected. To make connections without drawing over another line, all lines must be stored as they are drawn. To save memory space, this storage is done using typically a 10" x 16" grid, with one memory register storing information about one block of the 160-block grid. Lines must be drawn orthogonally; each register stores the presence or absence of a line at 16 ordinate values for horizontal lines, and 16 abscissa values for vertical. To prevent overlap, it is necessary to scan block-by-block the reserved lines along the route desired, and avoid entering a block at a coordinate value where a line has been stored.

This technique does not produce drawings with the aesthetic elegance that would be possible with, say, a 1/16" grid instead of the 1" grid, but it takes 1/16 the storage and is 16 times as fast.

One of the disadvantages of this program is that since it checks and draws lines a block at a time with no look-ahead, it can get trapped in a box or loop back on itself.

Another program for drawing schematics, which is presently about half completed has a look-ahead capability. It plans a trial line from the first point given to the second, and when it has made a connection, the line is neatened up by looking for short cuts and loops. The parameters of this line are stored, and the process is repeated, starting at the second point given and working toward the first. Then the two lines are compared, and the one with the fewest turns is actually plotted.

E-2089

DISTRIBUTION LIST

Internal:

R. Alonso
L. Baxter
M. Birnbaum
D. Bowler
J. Cirignano
R. Filene
A. Green
Eldon Hall
A. Hopkins
J. Pennypacker (10)
H. Thaler
L. E. Larson
D. G. Hoag
R. R. Ragan (w/letter of transmittal)
J. Kingston (letter of transmittal)

External:

National Aeronautics and Space Administration (50+1R)
Electronics Research Center
575 Technology Square
Cambridge, Massachusetts
ATTN: KC/Computer Research Laboratory