A SET OF KARNAUGH MAP MANIPULATION COMPUTER ROUTINES FOR USE IN LOGIC DESIGN

by
Charles Martin Shub

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1968

APPROVAL SHEET

Title of Thesis: A SET OF KARNAUGH MAP MANIPULATION COMPUTER ROUTINES FOR USE IN LOGIC DESIGN

Name of Candidate: Charles Martin Shub
Master of Science, 1968

Thesis and Abstract Approved: Alan B. Marcovitz

Dr. Alan B. Marcovitz
Associate Professor
Department of Electrical
Engineering

Date Approved: May 3, 1968

ABSTRACT

Title of Thesis: A Set of Karnaugh Map Manipulation Computer Routines for use in Logic Design.

Charles Martin Shub, Master of Science, 1968

Thesis directed by: Dr. Alan B. Marcovitz

The Karnaugh map provides a convenient visual aid for the manipulation of switching functions for both the design engineer and the student of logic design. Algorithms for the minimization of switching functions by the manipulation of information displayed on a Karnaugh map are presented, along with a method of obtaining more information than was previously possible from the Karnaugh map.

A dynamic, flexible, and easy to use collection of computer subroutines written in the MAD language to accomplish such manipulations as a subset of an entire logic design system of computer programs is described. A user's manual for the entire system is included, as well as descriptions of the programs used in conjunction with the map manipulation process. Several examples are included.

ACKNOWLEDGEMENTS

The author wishes to express his appreciation to Doctor Alan B. Marcovitz of the Electrical Engineering Department, University of Maryland, for his guidance and suggestions in the preparation of this thesis. Mr. Edward F. Miller's generosity with his extensive library of literature in the area is also appreciated.

This work was supported in part by the National Science Foundation under grant GK-811. Additional support was furnished by the University of Maryland Computer Science Center.

Several of the computer routines described in Chapters I, III, and IV of this thesis were written by James Piersall and Charles Keyserling. Mr. Keyserling prepared a thesis under the direction of Dr. James H. Pugsley of the Electrical Engineering Department, University of Maryland. Mr. Piersall prepared his thesis under the direction of Dr. Marcovitz.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND SYSTEM BACKGROUND CONSTRAINTS

In considering the development of the use of the Karnaugh Map as a visual aid in Computer aided logic design, it is obvious that the concepts involved in setting the design philosophy for an entire package of computer routines provide constraints for the use of the Karnaugh Map. The purpose of this research is to produce a set of Karnaugh Map manipulation routines which will successfully interact with other sections of a logic design package, thus providing a useful tool as a part of the overall system. Therefore, before dealing with the specific problem involved, it is necessary to understand the constraints imposed by the general system.

By way of background, then, this chapter comments on the general constraints and describes some other work being done in the general area of computer aided logic design that is necessary to the specific problem. The second chapter sets down some theory involved in the Karnaugh Map, and defines the rationale behind the various features of the Map package. The third chapter provides a complete User's Manual for the Map package, as well as the simulation, Input/output, and minimization packages. The fourth chapter provides detailed documentation of the subroutines used in the Map package including tables of subroutine nesting, storage requirements, and the use of compile time parameters. This is provided for aid in making any changes in the programs. The fifth chapter offers suggestions for further extension of the package, and the final chapter summarizes the results.

There are several important criteria that must be considered in any approach to the general problem of computer aided logic design. These

policy decisions, some of which have already been made in dealing with earlier phases of the project, as well as those decisions made in conjunction with the development of the map package are outlined below, along with some motivation for the decision. Original policy decisions made in early phases of the project were based to a large extent on the knowledge gained from studying previous approaches. The shortcomings of such approaches led to new ideas, while the highlights of other approaches were incorporated.

The present state of the art dictates that the machine is used only for assistance rather than for a complete design. Furthermore, it is not clear that it would be desirable to eliminate the human from the design process even if it should become feasible. For example, consider the problem of a vote register for the Security Council at the United Nations. Each nation would push a button to vote yes, and another button to vote no, and the yes light would light if the measure were passed. Clearly, the designer must consider several approaches to this problem before he is ready to use a computer. From the above description, the designer must first formulate a mathematical specification of the problem, then he must minimize his circuitry. From the minimization, the designer produces block diagrams. Using the block diagrams, the circuit is simulated to see that it does perform the correct task, and finally the circuit is built. The computer can best be used in the minimization and simulation phases. In light of this, and similar examples, the project proceeded in the vein of computer assisted logic design, rather than logic design by computer.

Present speed limitations on digital computers put a limit on the size of a problem which a computer cann assist in solving. The classic example for

this is, of course, the design of a parallel adder for a 36 bit computer. (Such as the IBM 7094). Such an adder can be considered as a 72 input - 37 output logic design problem. The truth table, alone, for this function has about $5*10^{21}$ entries! If a computer could print one line of this truth table per nanosecond, it would still take $5*10^{12}$ seconds or almost 200,000 years! Clearly, then the computer is not going to solve the "size problem." On the other hand, the computer can provide valuable assistance on most smaller functions, i.e., those involving up to twelve variables. Hand calculations for many such functions are inordinately complex, thus making manual solutions rather tedious. Yet the problem is small enough to enable the computer to assist in the bookkeeping. With this in mind, the decision was made to limit the program to problems involving up to twelve variables. This is not meant to imply that all problems involving twelve variables are small enough to be handled by the computer, and that all problems involving thirteen or more variables are too large. Twelve variables happened to be a convenient number for reasons which will become apparent later. If, however, it becomes desirable, the task of modifying the programs to handle problems involving up to eighteen variables is fairly simple.

There have been many computer routines written for assistance in both circuit and logic design and analysis. Most of them have at least one of the following shortcomings. One specific problem is solved, rather than a whole class of problems. Input formats are generally so restrictive that it usually takes longer to figure out how to set up a problem for computer assistance than to actually solve the problem. Furthermore, the input formats for two related problems are generally quite different. See, for

example, the IBM User's manual for ECAP (M20-0170-1). Also, it is nearly impossible to make minor modifications in the computer routines.

With these problems in mind, the Electrical Engineering Department at the University of Maryland has devoted substantial effort to the design of a package of several computer routines to assist an engineer in logic design. Some of the key concepts influencing the design philosophy have been outlined above. The highlights of this particular package of routines are ease of use and flexibility.

The requirement that the routines be easy to use dictates that the user of the routines need not spend several years, weeks, or days learning either a programming language or a lot of arbitrary formating rules in order to use the routines. The user needs merely several control cards, and his data sets. Chapter three contains a complete description of the use of these routines and can thus serve as a User's manual.

Flexibility dictates that the routines should allow several different uses. Furthermore, if the user has a specific need which is not taken care of by the routines, he should be able to modify the routines to apply to his particular problem. Consequently, the decision was made to use a high-level, procedure-oriented language; since in such an environment, it is much easier to understand what a computer routine does than it would be if the routine had been written in an assembler type language. Also, variable names in the routines have been chosen to parallel the engineer's thoughts. For example, NOTRM for the number of terms, LISTDC for a list of don't care terms, of DEC for a list of decimal equivalents. Furthermore, a high level

of documentation can be kept with each routine written. The particular language chosen was MAD because it has the requisite powers. For example, MAD allows: bitwise operations, compound conditionals, recursive subroutine calls, push-down stacks, iterated statements, simplified I/O, re-entrant subroutines, and subroutines with multiple entry and return points. Flexibility further implies that it should be possible to add new routines to the package without making changes in existing work. To this end, the package of routines consists only of subroutines or external functions so that the user must supply only a simple calling program as specified in Chapter 3.

In keeping with this flexibility, one feature initiated in conjunction with the map package is the idea that the routines should work for a variety of users with different information generated for different levels of use. For example, while a design engineer might merely be interested in a minimum cover for a switching-function, a student would require more information, such as a list of prime implicants, indications as to how the prime implicants were obtained, a list of essential prime implicants, a Karnaugh map of the function showing how the prime implicants and/or essential prime implicants combine to actually cover the function. Therefore, a level switch was built into the routines so that the user could indicate if he wishes to receive additional information. In actuality, then, there are two levels of interaction, namely the quick expert mode which will produce answers for an engineer; and also a slower, more informative, mode which generates information which is not necessary for the design engineer, yet which proves most helpful to the student in logic design.

The package of routines have been designed for eventual use in a real-time, user-interacting, conversational-mode computing environment. However, at the present time, such computing facilities are not generally

available, so testing of the package of routines has been done in a batch processing, hands off environment. The batch processing has led to the further constraint that in case of a fatal or pseudo-fatal error, the computer should go on and do as much as possible with the idea that partial results are better than none.

In summary of the major design concept of the overall package of routines, the following stand out: The user need not waste a lot of time learning a set of rules, but can use the routines almost immediately. The routines can be added to or modified if the user wishes to learn the mechanics of the MAD language. The routines provide for two levels of use, namely that of the design engineer who is merely interested in results, and that of the student who needs much more information.

To date, work has been done in several areas, a simulator exists. Routines to perform function decomposition are under development. Partition analysis is also under development. More specifically related to the specific problem of this dissertation, however, are the I/O package, and the minimization package.

The I/O routines (17) will read in a logic function in essentially free format, and perform the necessary manipulation to store the function in standard internal storage format. The output routines will print the logic function or parts of the logic function as either lists of product terms or as a sum of products.

Keyserling (8) uses the iterated consensus method to find the prime implicants of a function because that algorithm does not require that the function be expanded into canonical form, and is generally faster than the Quine-McClusky algorithm. The cover routines use a tabular method to produce all non-redundant covers, and then lists all minimum covers based on either

the number of prime implicants, the number of literals, or the number of prime implicants plus the number of literals for a sum of products realization. If the cover routines overflow storage, the best cover found so far is listed along with the message that it might not be minimum. Keyserling can also find the mulitiple output prime implicants by the use of pseudo-variables in a tag section as outlined on pages 209-218 of Prather. (18)

The simulator (17) will read in a logic circuit, simulate the operation of the logic circuit, and compare the output of that logic circuit to several switching functions.

The specific result of this research is the design and implementation of the set of algorithms to enable the computer to do all bookkeeping for the manipulation of Karnaugh maps. The routines developed include the ability to display a Karnaugh map of a switching function of up to six variables, and sufficient flexibility to enable both the student of logic design and the design engineer to manipulate switching functions on a Karnaugh map to produce minimal covers of the switching functions. Additional capabilities of result testing are also provided. This enables the user to compare his results with machine generated answers and thus provides, among other things, a homework grader and self instruction device for the student. The dynamic bookkeeping arrangement devised to provide the requisite flexibility needed to manipulate the switching functions on the Karnaugh map led to algorithms for first finding an irredundant cover of a switching function, and then further reducing of the irredundant cover. The idea of using unique identifiers for each prime implicant is introduced to aid in finding redundant terms.

CHAPTER II

THE MAP PACKAGE

Prior to examining the role that the Map Package should play in interaction with the overall system of computer aided logic design programs, it is necessary to delineate some of the underlying theory and develop some notation.

A switching function is any function which can be expressed by a finite number of binary switching variables and constants (0 and 1), combined by use of a finite number of occurrences of the basic operators ',+, and *. In general, the * operation is assumed. That is to say X*Y is represented by XY. A switching variable is denoted in this system by a letter followed by a string of from zero to five digits. The term literal means either the switching variable or its complement.

A product term is defined as a concatonation of literals. For example, A1C and D12E'F are product terms. A product term is said to be a standard product term or minterm if all the switching variables of the particular switching function are present either complemented or uncomplemented. A product number, standard product number, or minterm number is merely a decimal number which represents a particular standard product term. A derivation of the functional relation between minterm numbers and standard product terms can be found in Prather. (18) Two product terms are said to be adjacent if they are identical with the exception that one and only one literal appears in one product term complemented and appears in the other uncomplemented.

A K-grouping is defined as a product term with K switching variables not present. Thus, for example, an 0-grouping is a standard product term,

while a 1-grouping is a combination by use of the basic property $AF + A'F = F$ of switching algebra (19) of two adjacent standard product terms, and in general a K-grouping is a combination by the same property of two (K-1)-groupings. Note that K different sets if (K-1)-groupings yield the same K-grouping. For example, the pair of (K-1)-groupings ABC and A'BC and the pair BCD and BCD' both yield the same K-grouping BC. This notation is somewhat similar to the notation of Prather (18) in his development of the cellular N-cube, C^N, as a model for studying the minimization problem. A K-grouping is similar to one of Prather's K-cells in C^N. In general, the terms group or grouping are used interchangeably to refer to any K-grouping. A further constraint on groupings is that in order for a grouping to be defined it must be contained in the switching function under question. That is to say that if, by the use of the Shannon expansion theorem (4,5,9, 10,11,14,15,16,18,19) both the switching function and the grouping are expanded to canonical form, the set whose elements are the standard product terms generated by the expansion of the K-grouping is a proper subset of the set whose elements are the standard product terms generated by the expansion of the switching function. Thus, there is a finite set of groupings associated with any switching function. By way of illustration, consider the switching function $F = A'BC + ABCD$. Two groupings obviously are A'BC and ABCD. In addition, the groupings BCD, A'BCD and A'BCD' are also contained in this particular function. Notice that the groupings A'BCD and A'BCD' are contained in the grouping A'BC. Groupings which are contained in a larger grouping are called sub-groupings. Consider the set of groupings which are not sub-groupings of any grouping contained in the function. In light of the terminology above, it should be clear that

such elements are prime implicants of the switching function. Because of this fact, groupings which are not sub-groupings are referred to as prime groupings. Further, a grouping is said to be an essential grouping if it is a prime grouping, and if the canonical expansion of that grouping contains a minterm of the switching function which is not contained in any other prime grouping. It should be obvious that when a grouping is said to contain a minterm, what is meant is that the minterm is an element of the set of minterms obtained from the canonical expansion of the grouping.

The set of all K-groupings of a switching function is a partially ordered set. The ordering relation is that one K-grouping is said to be less than or equal to a second K-grouping if and only if both the numerical value of the first K is less than or equal to that of the second K and that all of the literals present in the second grouping are also present in the first grouping. This ordering relation can be extended to the "implies" relation discussed by Marcovitz and Pugsley,(11) where a switching function, F, is implied by G, i.e., $F \leq G$, if all minterms of F are minterms of G.

When talking about a set of groupings, it is convenient to think of the elements of the set being connected together by the "or" operation. That is to say that if a typical set of groupings is $\{A'BC, ABC'D, AB'C\}$, this grouping can be thought of as representing the switching function $G = A'BC + ABC'D + AB'C$. Clearly, then, any switching function, or part of any switching function can be represented by a set of groupings. For example, if $G = \{A'BC, ABC'D\}$ then the function $F = A'BC + ABC'D + AB'C$ can be represented by saying $F = \{G\} + AB'C$. In general, the curly brackets are dropped, and just $F = G + AB'C$ is written and it is understood from the context when G represents a set of groupings and when G represents a switching variable.

While the concept of covering a switching function may be familiar, the term cover is used here in the following senses. A switching function is said to be covered by a set of groupings if, when the switching function is expanded to canonical form, each standard product term of the switching function is contained in at least one of the groupings of the set. This is equivalent to saying that the set of groupings is a correct expression for the function.

A don't care term is defined as a minterm that may be contained in a switching function, yet which does not necessarily have to be contained in any grouping of a set in order for the set of groupings to cover the function. Define a function F to be made up of a set of Care terms, C, and a set of Don't care terms, D. One representation for this function would be $F = C + D$. Consider a function, F, (with care terms, C, and Don't care terms, D) and the set of groupings, G. Then define P as that set of minterms of C not included in G. Thus $P\,G = 0$ and $C \leq P + G \leq C + D$. If a set of groupings covers a function, then P is the empty set of minterms.

In general, the object of minimization theory is to generate a minimum cost cover for a given switching function. There are several criterion that can be used to determine relative costs of realizations of switching functions. In terms of logic blocks, the criteria are related to the number of gates, or the number of inputs, or a combination of the two. The details of conversion between product terms and gates can be found in Prather. (18)

In the terms of switching functions, the cost criterion which this paper will consider are the number of gates, which corresponds to the

number of product terms (plus one for the OR gate, but that is neglected;) the number of inputs to the AND gates, which corresponds to the number of literals; the total number of inputs, which corresponds to the number of literals plus the number of product terms; and finally a hybrid notation which uses number of gates, and if two functions have the same number of gates, then the number of literals is used. In terms of the grouping notation, the first cost function is that cost is directly proportional to the number of groupings in the set. The minimum number of inputs to the AND gates translates to cost being proportional to the product of the number of variables in the function and the number of groupings in the set decreased by the K of each K-grouping in the set. The cost function based on the total number of inputs is merely the above cost plus the number of elements in the set of groupings. Under the final cost criterion, the cost is proportional to the number of groupings in the set. If, however, two sets of groupings have the same number of elements, then the set with the higher total K has lower cost.

A set of groupings which covers a switching function is said to be a minimum set if there exists no other covering set with lower cost. Obviously, then, a set of groupings which contains subgroupings is not minimum, except in the case where just the number of groupings in the set determines the cost. If a grouping is a subgrouping, then it is contained in some larger grouping. The larger grouping has a higher K, and thus using the larger grouping will reduce the cost in the final three cases stated above. In the first case, it is obvious that of all minimum cost sets of groupings, there is at least one such set that contains only prime groupings. Thus, restricting minimum cost sets to sets of prime groupings, there exists no other set of groupings, prime or not, that is less expensive.

A set of groupings which covers a switching function is said to be a redundant set if any proper subset also covers the switching function. Under the first, third, and fourth cost criteria, it is trivially obvious that a redundant set cannot be a minimum cover. This is true also under the second cost criterion because the K of the grouping is always less than or equal to the number of variables, so by adding a grouping to a set, the cost must go up. On the other hand, a non-redundant set is not necessarily a minimum set.

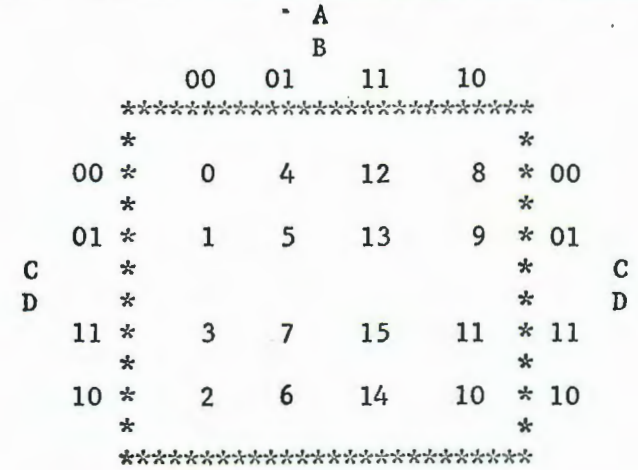
With this notation, certain aspects of the covering problem simplify to that of set system manipulation. The determination of whether or not a set of groupings is a redundant cover is quite simple, while the determination of minimality is not quite as trivial. The algorithmic approach to the first problem is to go through the groupings in the set one by one, and for each grouping determine if there is at least one element of that grouping contained in no other grouping. If this holds true for all groupings of the set, and the set does cover the function, then the set is a non-redundant set. The following approach can be used in the cost reduction problem. Given a set of groupings which covers a switching function, delete groupings from the set until the set in non-redundant. Call this set G. For all possible pairs of groupings P contained in G, determine if there exists a grouping "g" such that (G-P)+g is a covering set. Here, the notation G-P denotes a set of groupings G* such that G*+P is the set G and the sets G* and P are disjoint. If such a g exists, then the set (G-P)+g is a lower cost cover than G. If no such g exists for any pair P, the next step is to consider all possible trios of groupings T contained in G, and determine if there exists a pair of groupings P such that (G-T)+P is a covering set. This

procedure should continue until the n groupings removed from G leave just a single element.

The Karnaugh map can be a great help in this procedure. There is a one to one relation between the squares of the Karnaugh map and the set of all possible minterms of any switching function of N variables. (19) One feature of the map is that two minterms which can be combined to give a 1-grouping are represented by physically adjacent squares on the Karnaugh map taking into account that two squares on opposite edges are adjacent around the corner of the map so to speak. A 2-grouping is represented by four adjacent squares either in a two by two or one by four array. A 3-grouping is represented as eight adjacent squares, either in a four by two or eight by one array. In general, a K-grouping is represented by two (K-1)-groupings of the same shape adjacent to each other. The general practice is to represent a switching function on a Karnaugh map by placing an "1" in the squares representing the care terms, by placing an "X" or sometimes a "-" in the squares representing don't care terms, and leaving the remaining squares blank. This shows the user of the map which squares are covered by groupings, and which squares are not. Experience in working with the Karnaugh map makes it easier to choose additional groupings to cover a particular switching function.

In lieu of this experience, and perhaps even in spite of it, and in light of the remarks above, it is convenient, for visualization purposes, to assign a unique character to represent a grouping, and to display that particular character is each square of the map which the grouping covers. For example, in Figure 2.1, the character "Q" has been assigned to the grouping A'B'C', the character "R" has been assigned to the grouping B'D', and the character "S" to the grouping A'C'D. Notice that the square

representing the standard product term A'B'C'D' contains both a "Q" and an "R" to indicate that the square is covered by both grouping "Q" and grouping "R" rather than just an "X" to indicate that the square is in a don't care status because it is covered by some grouping. This assignment enables the Karnaugh map to display more information than was previously possible. For example, in Figure 2.1, it is immediately obvious that grouping "Q" is redundant insofar as every square of the map which contains a "Q" contains at least one other character.



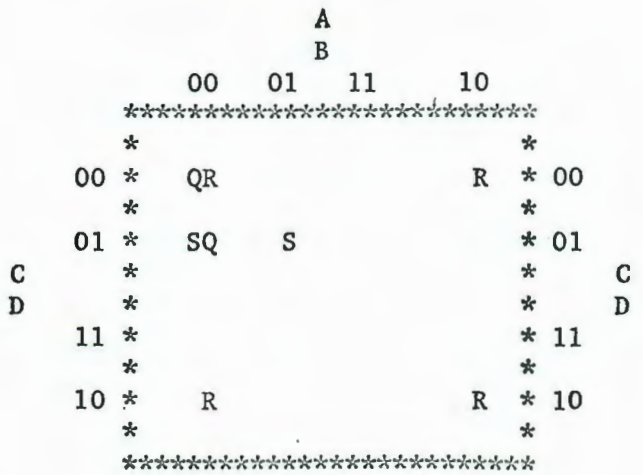


Figure 2-1. Illustration of Assignment of Character to Grouping.

Using this scheme does require a good deal of bookkeeping insofar as there are several details to be accounted for. First, which squares on the map are contained in the switching function. Secondly, which groupings are present in the set of groupings, and which characters represent each of the groupings. Finally, track must be kept of which characters go with which squares on the map. Based on the philosophies set down in the initial chapter, then, it appears that the digital computer will lend itself quite well to this particular approach to the covering problem as there is a lot of accounting to be done in the process of manipulating a set of groupings to build a minimum cover of a switching function.

The origin of the Map package was the undertaking of the following two tasks. First, to print a Karnaugh map of the switching function, and secondly to manipulate switching functions on the map. The purpose of the second task was to approximate the actual actions that the user of a Karnaugh map makes in selecting a cover for a switching function. The first approach to this manipulation was to respecify the input function with the selected terms in a don't care status, much in the way discussed in the earlier theory. Thus, the original function was internally changed to indicate both the original don't care terms and the covered terms as don't care terms, Unfortunately, this created problems insofar as there was no way to back up a step. Once this change was made,the original information was lost. The initial approach to resolving this problem was to write the switching function on tape prior to each grouping. The tape approach was discarded in favor of the dynamic bookkeeping scheme of assigning a character to represent a grouping.

In order to provide for eventual use in a real time user interacting conversational mode computing environment, and to increase the flexibility of the program, an option to read in information about groupings from the data file was added. This addition enabled the use of the program for several different data sets without modification of the simple calling program.

Rather than just print the map, it seemed that the internal information contained in the dynamic bookkeeping routine would be most useful to the student in logic design, so an option was added to enable the map to display the characters assigned to the groupings. Since this implied two levels of usage, a level switch was built into the programs to enable the student to obtain more information than the design engineer.

For the experienced design engineer, however, the information as to which groupings cover which squares of the map is not really important. Rather, he is concerned merely with whether the function is covered. Therefore, a computer program for the experienced engineer should not bother with printing special characters, but should merely print the Karnaugh map with all covered terms represented by an "X" and all uncovered terms represented by an "1". The student, on the other hand, should have several controls and checks made to determine if he has made a logical error in selecting a grouping. For example, if a student selects a grouping which does not cover any new terms of the switching function, he should be warned that he has made a poor choice. Also, if the student selects a grouping which is not a prime grouping, he should be warned. Furthermore, since any minimum set of groupings must contain all of the essential groupings, the student should be warned if he selects a non-essential grouping before he has selected all of the essential groupings.

These ideas lead to the concept that the map manipulation package should appear to be two separate packages, one designed for the student, and the other designed for the engineer. The package designed for the engineer would merely keep track of groupings and print simple Karnaugh maps, while the package designed for the student would perform several additional checks, and generate further information useful to the student which is superfluous to the design engineer. This duality of purpose has been achieved to some extent by the use of a mode switch which allows the user to indicate whether or not he desires the additional information useful to the student in logic design. Once this switch has been set, the value of the switch determines which level of output is used to perform the operation requested.

Insofar as Piersall's Input/output routines were equipped to handle inputs in a literal form, the option of specifying a grouping as a product term rather than a list of minterms was made available. At this point, it seemed as if the user had many different commands available to him, and there was a good deal of duplication of effort, so an attempt was made to systemize. the package of routines. For example, the prime implicants had to be generated on each pass through the grouping process to determine if the grouping was a prime one. Rather than do this, a prime implicant monitor was set up to generate the prime implicants one time and store the information. There was also several commands set up to enable the user to use just one command instead of six commands to produce all of the minimum covers.

Motivated by the desire of checking a student's work, a result testing feature was added. Once set up, it seemed advisable to add the capabilities for other types of result testing, such as checking a list of prime implicants or essential prime implicants for correctness.

The next observation made was that there were several approaches to finding a covering set of groupings.

One procedure is to start with the empty set of groupings, and to add groupings to the set to build up a covering set. A second procedure would be to start with the groupings implied by the input form of the function and to remove any redundant groupings. For example, the groupings implied by the function F= A'C'+ B'C'D'+ A'B'D + AB'C+AB'D'+ B'CD are A'C', B'C'D', A'B'D, AB'C, AB'D', and B'DC while a more minimal covering set is A'C', AB'D', and B'CD. A third procedure would be to start with the set of groupings being the set of all prime groupings, and to remove any redundant groupings from the set. The fourth procedure would be to start with all the essential prime groupings and add groupings to the set until a covering set is obtained.

Finally, due to the fact that on some functions, particularly those with cyclic prime implicant tables, the number of irredundant covers was so large as to exceed the machine capacity, an algorithm for further reducing the irredundant cover was developed.

Obviously, the entire map manipulation package would be most useful in a real-time, user-interacting, conversational mode computing environment. Presently, however, such an environment is not readily available, so the routines must be developed in a batch-processing, hands off environment. This mode of operation renders the routines a lot less useful in producing the dynamic results described insofar as the time delay between submitting information and receiving intermediate results makes the process a static one rather than a dynamic one.

The map manipulation routines presently work for functions involving from two to six variables. The command to produce a Karnaugh map of a switching function is MAP. With each switching function, the set of groupings is initialized as the empty set.

Adding a grouping to the set is accomplished by using the command GROUP. This command, when used with a list of parameters, treats the parameters as standard product numbers, and automatically finds the smallest grouping which contains all of the standard product numbers in the list. Alternatively, if the parameter field is empty, the command GROUP. causes a scan of the data file for instructions and continues executing those instructions as long as they are recognizable. The data file instructions are the control work GROUP followed either by a list of standard product numbers, or else a product term which is taken to be the grouping desired. If the data card contains a list of standard product numbers, the grouping is found in the same manner as if the numbers had been a list of calling parameters after the GROUP. command.

There are several alternative methods for initializing the set of groupings, plus a command to empty the set. RESETS. is the command to empty the set of groupings. ESSGRP. is the command used to initialize the set of groupings as the essential groupings. The command PRIGRP. is used to initialize the set of groupings as the prime groupings. Finally, the command SETGRP. puts the groupings implied by the input form of the function into the set of groupings.

There are several methods for removing a grouping from the set. If the command GROUP. without a parameter list has caused the data file to be scanned for instructions, the instruction UNDO causes the last grouping

added to the set to be deleted. Also, the instruction card REMOVE will cause the grouping represented by the character appearing on the data card to be removed from the set of groupings. On the other hand, the program command UNDO.(N) causes the last N groupings added to the set to be deleted. If the parameter N is omitted, it is assumed to be a 1.

Having manipulated a set of groupings, the user, if he is so inclined, can compare his result with a computer generated list of minimum sets. This is done by either the instruction CONFIRM or CONFRM in the data file if an earlier command of GROUP. has caused the scanning of the data file, or alternatively by use of the result testing feature of the map manipulation package.

Use of the result testing feature is obtained with the command CONFRM.(N). There are four permissable values of N, namely 1, 2, 3, and 4. A value of 1 for N directs the result tester to compare the set of groupings with the machine generated minimum sets to determine if the set of groupings is indeed a minimum cover of the switching function. A value of 4 for N directs the result tester to read in a second switching function expressed in a sum of products form, and compare this second function with the minimum covers of the original switching function. A value of 3 for N directs the result tester to read in a list of product terms and compare that list with a machine generated list of essential prime implicants. Finally, a value of 2 for N directs the result tester to read in a list of product terms, and compare that list with the machine generated list of prime implicants. In any case, the result tester indicates if the user's list matches the machine generated lists, and in the event that comparison was unsuccessful, both the user's list and the machine generates lists are printed to show where the error exists.

A detailed description of the uses of these commands, and other commands available, as well as a description of the restrictions on the input formats are given in the following chapter.

With the aid of a digital computer as outlined above, the matter of determining if a set of groupings is redundant is trivial at best. Merely scan the map to ensure that each character assigned to a grouping appears alone in at least one square of the map. If, for example, the character "A" has some other character in the same square in every square that has an "A" in it, then the grouping represented by the character "A" is clearly redundant insofar as it can be removed without uncovering the switching function.

The procedure for improving on minimality is not quite so simple. A cut and try approach as a first approximation which seems somewhat successful is to remove a grouping which is essential for only one square on the map. This is equivalent to looking for a character which appears alone in only one place. Then determine if there is a grouping which contains the now uncovered square which will also cover the only essential square of another grouping. This gives removal of two groupings for the addition of only one, thus removing an element from the set of groupings.

Obviously, this procedure does not apply to essential groupings insofar as no essential grouping can be removed from the set. However, in functions with many non-essential groupings, this procedure is quite useful. It is generally most useful when the groupings are of a cyclic nature, such as in the second example in Appendix B.

CHAPTER III

COMPLETE USER'S MANUAL

The typical form of a program to use the logic design package is

```
1     7     12        16
$EXECUTE              MAMOS
$ID   NAME                  *XXX/YY/ZZZ
$COMPILE MAD, EXECUTE                 -
          NORMAL MODE IS INTEGER
          PROGRAM COMMON DUMMY (12188)
START     CONTINUE
               {<commands>}
          TRANSFER TO START
          END OF PROGRAM
$BINARY
               {<binary decks>}
$DATA
               {<data sets>}
```

If the simulation routines are to be used, the number in parenthesis after DUMMY must be increased from 12188 to 13600. The computer size limits prohibit use of both the simulator and the map manipulation packages at the same time, so there are two different binary decks available from the Electrical Engineering program library. In addition, under the University of Maryland operating system, the subroutines are available on tape. The tape for the Map package is number 2030, while the tape for the Simulator is number 406. When using the tapes instead of the binary decks, the job

submittal card should contain a note in the remarks portion that the desired tape is needed on unit A-5 with no ring. In addition, the following program format should be used:

```
1     7     12     16
$IBSYS
$*    TAPE 2030          ON    A - 5     RING OUT
$PAUSE
$EXECUTE              MAMOS
$ID   NAME
$COMPILE MAD, EXECUTE
            NORMAL MODE IN INTEGER
            PROGRAM COMMON DUMMY (12188)
START       CONTINUE
               {<commands>}
            TRANSFER TO START
            END OF PROGRAM
$BINARY (11)
               {<data sets>}
```

The listing of program commands is broken down into three sections. The first section describes the commands to the input and minimization programs which are common to both the Map package and the simulator. The second section delineates the commands available in the Map package. The final section gives the commands to the simulator. All input commands must be punched in columns 12-72 of the command card.

INPUT COMMANDS

READS. Reads in and prints out a switching function. See the section on data sets for acceptable formats.

PRIMES. Lists the Prime Implicants of the currently stored function.

ESSENS. Lists the Essential Prime Implicants of the currently stored function.

COVERS. Lists all the minimum covers of the currently stored function using the number of prime implicants as the criterion for minimality.

In addition, minimum covers can be generated by the following sequence:

ITER. Gets a list of prime implicants in storage.

EXPAN. Places a canonical expansion in storage.

TAEX. Expands the basic cell matrix (generates prime implicant table)

ENUMER. Enumerates all non-redundant covers within the machine.

PRALL. Prints all non-redundant covers. (optional call)

WGT. Prints all minimum covers using number of prime implicants

or as criterion for minimality.

WGTA. Prints all minimum covers using number of prime implicants

or plus number of literals as criterion for minimality

WGTB. Prints all minimum covers using number of literals as criterion for minimality.

MAP PACKAGE COMMANDS

MAP. Prints a Karnaugh map of the currently stored function.

EXPERT. Causes the program to delete additional printing and checking which is mainly valuable to the student.

TEACH. Returns program to mode of teaching aid. (This is set as the normal mode of operation).

GROUP.(list)Where the list is a list of standard product numbers. Finds the smallest grouping "G" that contains the minterms on the list. If the currently stored function has already been covered by previous calls to GROUP., a message is printed. If "G" is not contained in the currently stored function, a message is printed. If "G" is not an essential prime implicant of the currently stored function, and there are still essential prime implicants not found, a warning message is printed. If "G" is not a prime implicant of the currently stored function, a warning message is printed. "G" is printed out in literal form and assigned a character. If one of the standard product terms in the currently stored function has been included in six or more prior groupings, a message is printed saying that the particular minterm will not be mapped with the character assigned to "G". A list of all the groupings made so far on the currently stored function is printed out, and then a special Karnaugh map is printed. This map represents all "care" terms that have not been included in a grouping as a "1". All don't cares that have not been included in a grouping print as an "X". All other terms print out as the character or characters assigned by the different groupings.

GROUP. Scans the data cards for recognizable instructions and executes the instructions until there are no more recognizable instructions.

UNDO. Undoes the effect of the last GROUP. command.

UNDO. (N) Undoes the effect of the last N GROUP. Commands.

CONFRM. (N) If N = 1, the set of groupings made by the use of the command GROUP. is compared to a machine generated list of minimum covers.

If N = 2, result comparison cards (see data section) are read from the data cards and comparison is made with the machine generated list of Prime Implicants.

If N = 3, result comparison cards are read from the data cards and comparison is made with the machine generated list of essential prime implicants.

If N = 4, result comparison cards are read from the data cards and comparison is made with the machine generated list of minimum covers.

If N = anything else, an error message is printed. In all cases, a message is printed as to whether comparison was successful. In addition, both the user's list and the correct list are printed if the comparison was unsuccessful.

SETGRP. Displays a map with groupings as implied by the input function.

ESSGRP. Displays a map with groupings as implied by the essential prime implicants of the currently stored function.

PRIGRP. Displays a map with groupings as implied by all the prime implicants of the currently stored function.

RESETS. Negates all previous GROUP. commands.

SIMULATOR COMMANDS

SIMLAT. Reads in and simulates a logic circuit. See the section on data sets for acceptable formats.

CHECK. Reads in one or more switching functions and determines if the switching functions are realized by the logic circuit.

DATA CARDS

There are four types of data cards that can be used. They are function specification cards, GROUP. instruction cards, results comparison cards, and simulator logic circuit description cards. A brief description and severl example are given below, along with the formal meta-language description.

1) Function Specification card is:

{<null> or { <fname> = } } <funct1> or <funct2>

where <funct1> is a list of standard product numbers

and <funct2> is a sum of products literal input form

examples: FUNCTION = 1,2,7,9,12

F = ABC + A'B'C'

ONE WITH DON'T CARES = 1,2,5,6(8,10)

ANOTHER WITH DON'T CARES (A,C,BO) = ACBO + A'C'BO, ACBO'

2) There are several recognizable GROUP. instruction cards which cause different actions. The form of an instruction card to make a grouping is:

GROUP <break> {<product term> or L {<product number>} } <rp>

examples: GROUP A'BC'
GROUP.(4,7,8)

The form of an instruction card to remove the last grouping made is:

UNDO

The form of the instruction card to remove the grouping represented by the character <char> is: REMOVE <char>

The form of the instruction card to simulate a program command of CONFRM.(1) is:

CONFRM or CONFIRM

3) Result comparison cards are identified by two asterisks(*) in the first two non-blank positions on the data card. The form is:

$$** \left\{ L \left\{ \langle \text{product term} \rangle \right\} \quad \underline{\text{or}} \quad L + \left\{ \langle \text{product term} \rangle \right\} \right\}$$

examples: **ABC A'B'C'

**ABC, A'B'C'

In all three cases, blanks on data cards are ignored, and the character "/" indicates that the data set is continued on the next data card. In the input specification, any don't care terms are put in parenthesis if the input is a list of standard product numbers, or separated from the care terms by a comma if the input is in literal form. At present, the literal form must be as a sum of products without parentheses.

4) Simulator logic circuit description cards.

A) An identifier represents a gate or logic block. The same i identifier represents the output of that logic block. It must be punched in columns 1-10 of the data card.

$$\langle \text{identifier} \rangle \quad \underline{\text{is}} \quad \left\{ \langle \text{letter} \rangle \quad S_0^5 \left\{ \langle \text{digit} \rangle \right\} \right\}$$

B) Type represents the type of logic block. Presently available types are AND, NAND, OR, NOR, NOT, EXOR (Mod 2 addition). Piersall details methods for adding other types of gates. The type must also be punched in columns 1-10 of the data card.

$$\langle \text{type} \rangle \quad \underline{\text{is}} \quad \left\{ S_2^4 \quad \langle \text{letter} \rangle \right\} \quad \underline{\text{or}} \quad \left\{ \langle \text{letter} \rangle \langle \text{letter} \rangle \left\{ S_1^2 \langle \text{digit} \rangle \right\} \right.$$

C) Inputs represent the inputs to a gate or logic block.

$$\langle \text{input} \rangle \quad \underline{\text{is}} \quad \left\{ \langle \text{identifier} \rangle \quad \underline{\text{or}} \quad \langle \text{variable name} \rangle \quad \underline{\text{or}} \quad 0 \quad \underline{\text{or}} \quad 1 \right\}$$

Inputs are punched in columns 12-80 of the data card.

D) The element of the logic circuit is described below:

<element> is {<type> or <null>} <identifier> L{<input>}

E) The normal gate type specification. This card must have columns 1-11 blank. Any gate of logic block with no type specification is assumed to be of the normal type. If no normal type is specified, the AND gate is assumed to be the normal type.

NORMAL GATE TYPE IS <type> or N'S <type>

F) Indicator of the end of a block of input data. This card must have columns 1-11 blank. The form is

END OF CIRCUIT or E'T

G) Input specification card allows an input to be specified as a constant (0,1). Punch INPUTS in columns 1-10, the list of inputs to be defined in columns 12-80 followed by 0 or 1

INPUTS L{<input>} = {0 or 1}

H) Output specifications are made by punching the control word OUTPUT in columns 1-10 and a list of elements in columns 12-80. The normal situation is that all outputs are printed.

OUTPUTS {ALL or NONE or L{<identifier>}}

EXAMPLE

```
1          12
           N'S NOT
AO         A
DO         D
AND G1     A,B,C
AND G2     AO,B,DO
```

```
AND G3      B,C,DO
OR  F       G1,G2,G3
OUTPUTS     G1,G2,G3,F
            E'T
```

Specifies the function $F(A,B,C,D) = ABC + A'BD' + BCD'$

CHAPTER IV

PROGRAM DOCUMENTATION

This detailed description of the workings and interconnections of the routines is provided mainly for the information of users of the package who have some knowledge of the MAD language and wish to make modifications in the package. For these purposes, it is helpful to define what is meant by several terms. In general, the words "routine", "subroutine", and "program" are used interchangeably and refer to one specific MAD external function. The correspondence is also made between the title of the routine, and a typical call. For example, the routine which has the entry GROUP., can be referred to either as GROUP., or the grouping routine. Variable names are given in capital letters, examples are PROPRI or TEACHR.

It is also important to be aware of the different ways of storing information about the function in the computer. There are two different types of function storage used. One is as the binary equivalent of the decimal standard product number. For example, the standard product number 37 is stored in the computer as 100101 (2) right justified in the storage location or 000000000045 (8). This form of storage is generated only when the function is expanded to canonical form, or read in as a list of standard product numbers. This information is kept in the program common storage array DEC. The other storage form is what is referred to as the standard internal storage format. This format enables one product term to be stored in each machine word. The machine word is divided up into twelve equal parts. In the 7094, this means that each part is composed of three bits. Thus one machine word is composed of 12 bytes, and each byte has

three bits in it. Each byte represents one variable of the function. If a variable is not present in the particular product term, the corresponding byte has the value of 0. If a variable is present in unprimed form, the byte has a value of 1, and if the variable is present in primed form, the byte has a value of 2. The bit configurations corresponding to the byte values of 0, 1, and 2 are 000,-001, and 010 respectively. Though it appears at this point that there is a superfluous bit in each byte, it turns out that the extra bit is most convenient in performing the consensus operation. The storage assignment is such that the least significant byte corresponds to the first variable, the second least significant byte corresponds to the second variable and so forth.

In the detailed documentation, several routines are described as part of either the minimization package, the simulator, or the input output package. These packages refer to sets of programs written by Charles Keyserling and James Piersall respectively. Needless to say, the documentation on these routines is not nearly as detailed as the detail on the map package.

In addition to the descriptions of the programs, several tables are provided. Table 4.1 lists a name for the various program common variables, and their general uses. Table 4.2 describes the meaning of the value stored in the variable PROPRI. Table 4.3 describes the meaning of the value stored in the variable LOC. Table 4.4 delineates the compile time parameters which can be changed to increase or decrease the size of various working storage tables. Note that changes in the parameters ADIM or INDIM must also be compensated for by making changes in the minimization package which is written in machine language for speed in execution. Table 4.5 provides a list of programs which use system input/output routines with a view to show

TABLE 4.1

PROGRAM COMMON STORAGE

VARIABLE NAME	USE
NOTRM	number of terms in the currently stored function
NOCAR	number of care terms in the currently stored function
NOVAR	number of variables in the currently stored function
EN	used by Keyserling in minimization package
ESS	used by Keyserling in minimization package
BL7	on line flag, normally 1. Set to zero for on line operation
TEACHR	mode switch, normally 1. Set to zero for expert mode operation
COVPRT	normally 1. Set to zero if generated covers are to be saved on tape for result testing routine.
PROPRI	product of primes indicating which routines have been used.
UNVDAT	normally 0. Set non-zero if error is detected.
LOC	Output flag. See table 4.3
PS	Location of first term to be printed
PN	Number of terms to be printed
A(10000)	Dynamic storage used by Keyserling
IN(1023)	Internal storage of the function
DEC(1023)	Storage for decimal equivalents
NAME(12)	Variable names
NAMEL(12)	Length of variable names
FNAM(100)	Title of the function

TABLE 4.2

USE OF PROPRI

If the value in PROPRI is a multiple of this number	this routine has been called
2	MAP.
3	GROUP.
5	EXPND.
7	QMC.
11	MYESS.
13	COVER.
17	ESSGRP.
19	PRIGRP.

TABLE 4.3

USE OF LOC

If the value in LOC is	you are printing out
0	the input function
1	the prime implicants
2	the canonical expansion of the input function in dynamic storage
3	the essential prime implicants
4	the one minimum cover
5	the first of several minimum covers
6	additional minimum covers
7	canonical expansion in input vector with care terms and don't care terms separated

TABLE 4.4

COMPILE TIME PARAMETERS

Subroutine	Parameter	value	use
CONFRM.	FDIM	500	maximum number of prime implicants to be compared
CSCOV.	BADIM	200	maximum number of terms in cover
ERRORS.	LIMIT	13	number of different errors
DATA.	TAPENO	2	logical tape to save contents of PROGRAM COMMON on
GETEM.	TDIM	1000	erasable storage
EXPND.	LSTDIM	1023	erasable storage
TAPE.	TAPENO	3	tape to write minimum covers on
SETGRP.	GP1TIM	37	limit to number of times GROUP. commands can be made
	NRGRP	100	size of storage array
	ADIM	10000	size of dynamic storage
	BLDIM	9	number of PROGRAM COMMON control parameters
	NRBITS	3	number of bits per variable in machine word storage
	INDIM	1023	size of input vector and decimal equivalents
	UPPER	63	number of squares on largest map printed
	NAMDIM	100	maximum size of name of function

TABLE 4.4A

ROUTINES USING PARAMETERS

BLDIM	ADIM	INDIM	NVDIM	NRBITS	NAMDIM	UPPER
CONFRM.	CONFRM.	ERRORS.	EXPND.	DECIML.	LETTER.	GROUP.
CSCOV.	CSCOV.	EXPND.	LETTER.	EXPND.	NUMBS.	MAP.
COVER.	ERRORS.	GROUP.	MAP.	GROUP.	PNT.	SETMAP.
ERRORS.	EXPND.	LETTER.	NUMBS.	LETTER.	READ.	
EXPND.	GROUP.	MAP.	PNT.	STORAJ.	SORT.	
GETEM.	LETTER.	NUMBS.	READ.			
LETTER.	MYESS.	PNT.	SORT.			
MAP.	NUMBS.	READ.				
MYESS.	OUTPUT.	SETGRP.				
NUMBS.	PNT.	SORT.				
OUTPUT.	READ.					
PNT.	SETGRP.					
QMC.	SORT.					
READ.	TAPE.					
SETMAP.						
SETGRP.						
SORT.						
TAPE.						
USER.						

TABLE 4.5

ROUTINES USING SYSTEM I/O

CARD INPUT	PRINTER OUTPUT	TAPE I/O
ERRORS.	CONFRM.	DATA.
GETEM.	COVER.	TAPE.
READ.	ERRORS.	
	ESSPRI.	
	EXPAN.	
	GETEM.	
	GROUP.	
	ITER.	
	LETTER.	
	MAP.	
	NUMBS.	
	OUTPUT.	
	PNT.	
	READ.	
	SETGRP.	
	TAEX.	

where the reprogramming effort must go to convert to real time operation. This table also lists the programs which use auxilliary storage which is presently tape storage. Table 4.6 lists the storage required, the entry points, and other routines required for each routine in the logic package. Table 4.7 gives the library programs required from the operating system library. Table 4.8 delineates the uses of some of the variable names in several of the routines. This is for referral during study of the actual documentation and helps to add insight to what the routines are doing.

Several figures provide basic flow diagrams for a few of the routines. These diagrams are referred to in the text of the documentation.

Copies of all source programs are available at the Electrical Engineering Program Library, or can be obtained by force punching University of Maryland Computer Science Center tape number 204. A complete listing is available on Computer Science Center tape number 985, as well as in the Electrical Engineering program library.

CONFRM. (see Figure 4.1) is the nucleus of the result testing feature. It receives as inputs the user's list and the machine generated lists, and makes the comparison. The form of the call is CONFRM.(N). An alternate entry is to CFRM.(N) which is used internally by the grouping routine when a CONFIRM instruction is read from a data card. If N is equal to 2 or 3, QMC. or MYESS. is called to get the storage locations of the prime implicants or essential prime implicants which are then moved into the FINK array. Calls to GETLST. and ADLST. place the users list of prime implicants in the ARRAY vector. ARRAY is then compared term by term with FINK, and then if comparison was successful, a message is printed stating so. If the comparison was unsuccessful, in addition to a message, both the user's list and the machine generated list are printed. Control is then returned to the calling program. If N is equal to 1, a call to GROUP7. stores the groupings in

TABLE 4.6

STORAGE REQUIRED AND ROUTINES CALLED

NAME	OCTAL STORAGE	ROUTINES CALLED DIRECTLY
CONFRM.	2521	GETEM. COLAPS. COVER. CSCOV. GROUP. MYESS. ORDER. PNT. QMC.
COLAPS	70	none
COVER.	106	DATA. ESSPRI. EXPAN. QMC. TAEX. TAPE.
CSCOV1. CSCOV3. CSCOV4.	762	ORDER. COLAPS.
DATAS. DATAR.	44	none
DECIML.	77	none
ERRORS.	561	none
ESSPRI.	164	OUTPUT. TAPE.
EXPAN. EXPAND. EXPALL.	207	none
EXPND. EXPND2.	451	COLAPS. ERRORS. ORDER.
GETEM. GETLST. ADLST.	1120	ERRORS. LETER2.
GROUP. GROUP3. GROUP4. GROUP7. UNDO.	2475	CONFRM. DECIML. ERRORS. EXPND. GETEM. MAP. MYESS. ORDER. PNT. QMC. RUTH. STORAJ.
INTDEC.	242	none
ITER.	275	OUTPUT.
LETTER. LETER2.	1244	ERRORS. SORT.

TABLE 4.6 Continued

MAP. MAP2.	1702	DCIML. EXPND. OUTPUT. SETMAP.
MYESS.	204	EXPND. QMC.
NUMBS.	577	ERRORS. INTDEC. STORAJ.
ORDER.	116	none
OUTPUT. LIST. PRINT.	473	PNT. ROTH.
PNT. PNT2. LST PRT.	1015	none
QMC.	55	ITER.
READ.	551	ERRORS. LETTER. NUMBS.
RUTH.	21	none
SORT.	160	none
SETGRP. PRIGRP. ESSGRP. RESETS.	636	GROUP. MAP. MYESS. QMC.
SETMAP. SETMP2.	174	none
STORAJ.	103	none
TAEX. ENUMER. PRALL. WGT. WGTA. WGTB.	1342	OUTPUT. TAPE.
(TAPE) PRTOUT. FINISA. TAPOUT. INIT.	315	CSCOV. PNT.

TABLE 4.6 Continued

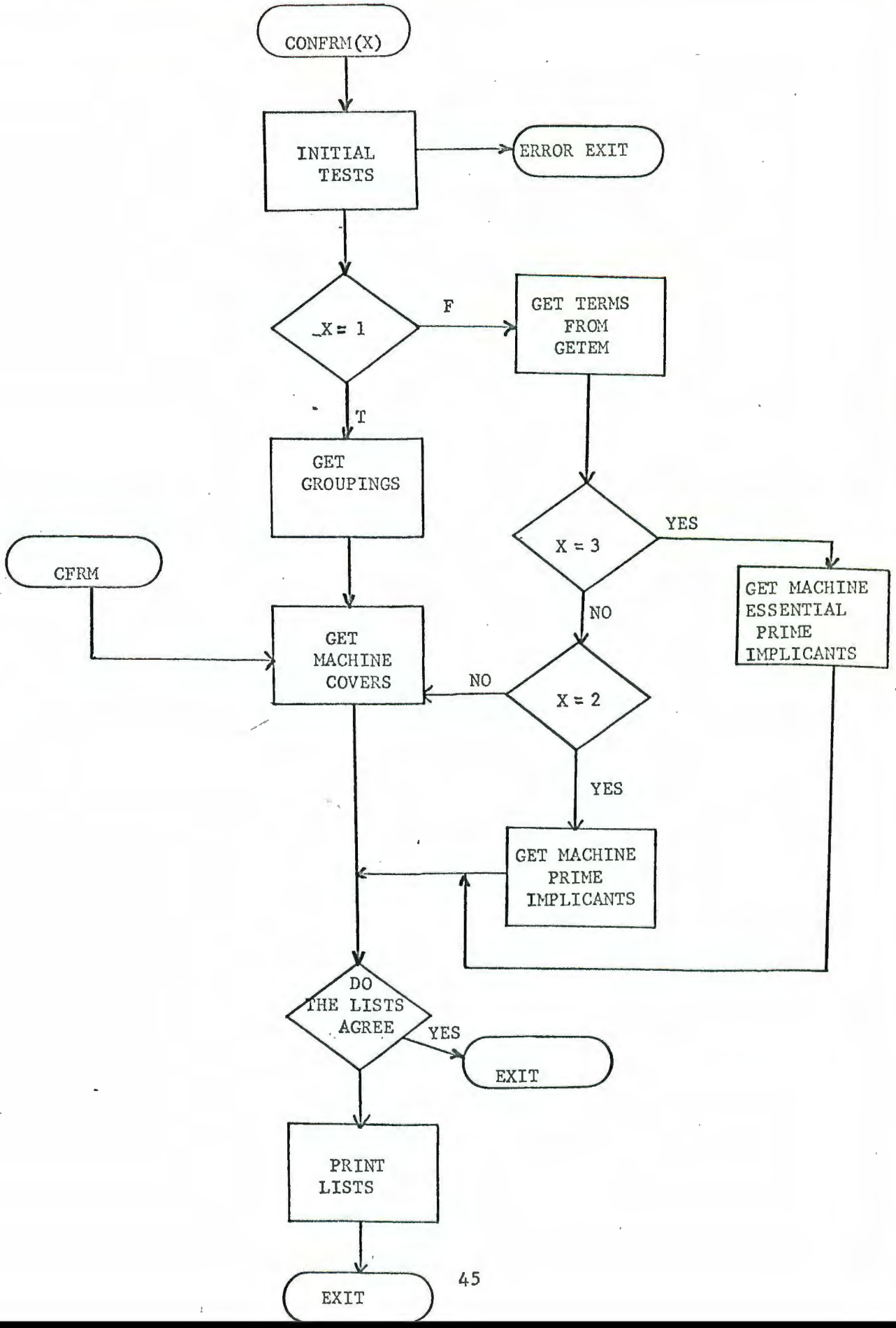
(USER)	54	COVER. MYESS. OUTPUT. QMC. READ.
COVERS.		
ESSENS.		
PRIMES.		
READS.		
EXPERT.		
TEACH.		
SPRINT	266	none
DFMP		
DFDP		
.PROCT		
.PRBCD		
CLOSIT		
SYSTEM		
ERROR		

TABLE 4.7

LIBRARY SUBROUTINES USED

SPUNCH	WRSBIN	.CHEKIO	RDSBIN	REWTAP
RUNTAP	SPEEK	SCARDS	SKIP6	SPRINT
SYSTEM	ERROR	.IOH	.IR4	.IR2
.IR1	.03311	.RTN	SETERR	.ERPRT
.ERR	.PRINT	.WR.	.RD.	.RDSW
.READ	.LOOK	.PRBCD	.EXIT	ATLOC
GTCTLW	.01311	.RST4	.IOFMT	.IOSET
.LBUFF	SETEOF	.RWT	.EOT	.EOF
.EOFLG	.EOF7	.IOB	.RBIN	.WBIN
SPRAY	ZERO	.RSTOR	.SAVE	.SET
(SUBT)	COMPZ			

FIGURE 4.1 -- Flow Diagram for CONFRM. Routine.



the ARRAY vector. If N is equal to 4, a call to GETLST. places the user's cover in the ARRAY vector. COVPRT is set to zero. The ARRAY vector is sent to be checked by CSCOV1. COVER. is called to generate the covers and to make the comparison. COSCOV3 is called to determine if comparison was successful. A message is printed out. If comparison was not successful, the user's cover stored in ARRAY is printed, and after setting COVPRT to 1, the machine covers are printed by a call to COVER.

The CFRM entry receives the elements of the user cover from the calling routine, and stores them in the array vector and proceeds as if N is equal to 1 or 4.

The following error conditions exist, and anyone will terminate the routine: If UNVDAT is non-zero, if TEACHR has been set to zero, if the calling parameter N is other than 1 thru 4, when the number of prime implicants or essential prime implicants or terms of the cover is greater than FDIM (500).

COLAPS. is used to edit duplications from a list. A typical call is: COLAPS.(L,LL) which causes L(1)...L(L) to be edited into LL(1)...LL(LL) where LL contains the same elements as L except that adjacent equal terms are condensed to one equal term. For example, if L = 5,7,8,8,8,9, then LL= 3,7,8,9. The call COLAPS.(L,L) is legal.

COVER. is used as a control program to generate the machine covers, store them on tape, and either print them or use them for comparison by the CONFRM. routine. If the covers are to be printed, a message saying so is printed. If PROPRI indicates that the covers have been generated and are stored on tape, the following generation sequence is skipped. The generation process begins with a call to QMC. to ensure that the prime implicants are stored

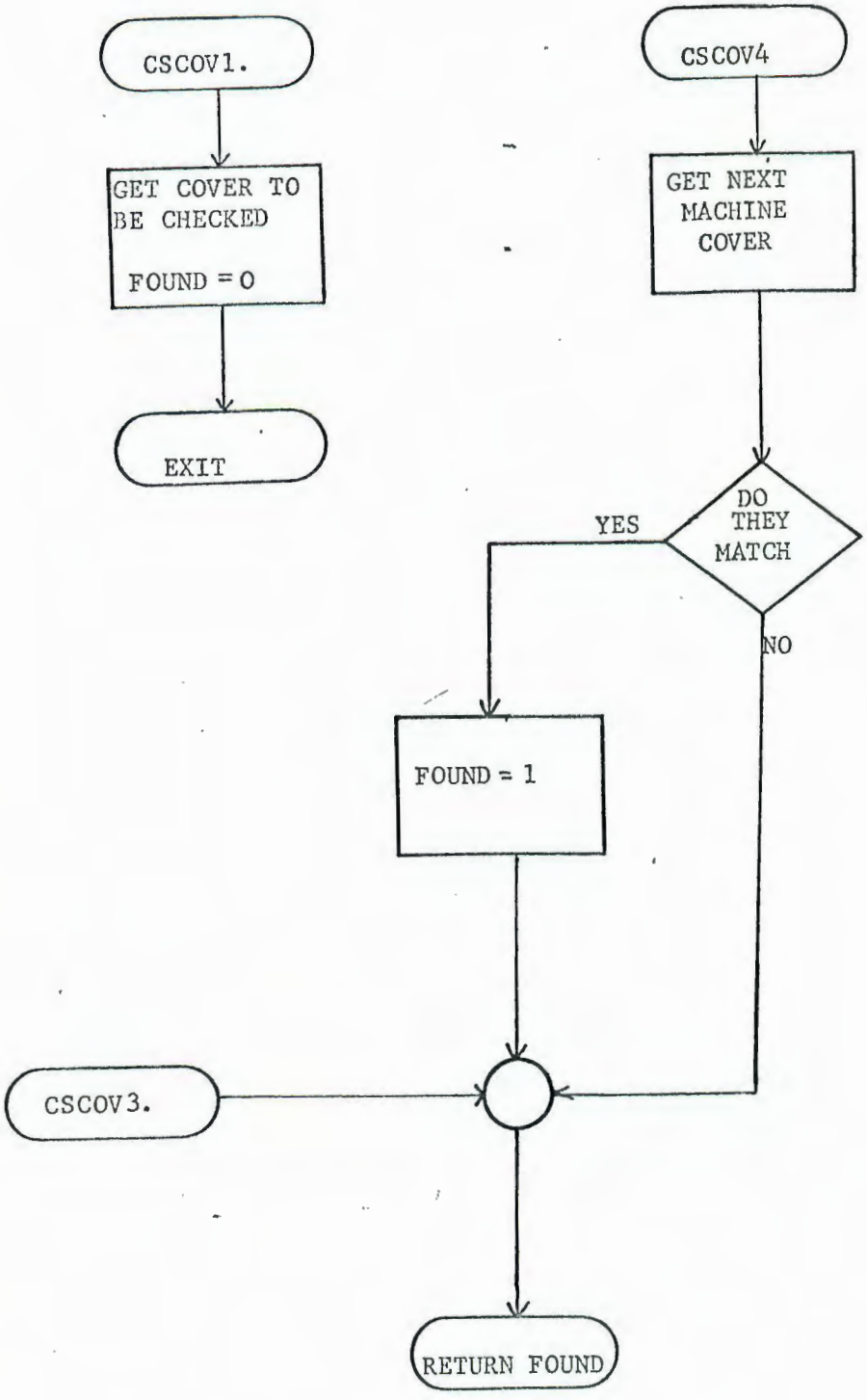
in dynamic storage. DATAS. saves the contents of PROGRAM COMMON storage. COVPRT is set to zero so that found in calls to Keyserling's routines will be sent to tape storage instead of being printed. INIT. initializes the storage tape. The sequence, EXPAN. ESSPRI. TAEX. ENUMER. WGT. uses Keyserlings routines to generate the covers and write them on tape. FINISA. puts an end block on tape, and DATAR. restores the contents of PROGRAM COMMON storage thus finishing the generation process. Then a call is made to PRTOUT. to either print the covers or send them to the result testing routine.

CSCOV. (see Figure 4.2) is used in the result testing package to make comparison between user generated covers and machine generated covers. It has three entry points. CSCOV1. is called from CONFIRM, and stores the user cover in the AARRAY vector. The user cover is ORDERED, and duplication are removed by calling COLAPS. the variable FOUND is initialized to 0B. CSCOV4. is called from the output section of the tape storage routine TAPE which sends a machine generated cover for comparison. The machine cover is stored in the BARRAY vector and compared with the user cover in AARRAY. If they match, FOUND is set to 1B and control is returned to the calling program. CSCOV3. returns the value of FOUND to the CONFIRM. routine.

DATA. has entries DATAS. and DATAR. DATAS. writes the contents of PROGRAM COMMON onto a scratch storage tape, and DATAR. reads the information back into PROGRAM COMMON STORAGE. The data tape is always rewound both before and after reading or writing.

DECIML. receives a standard product term in internal storage format and returns the decimal number equivalent.

FIGURE 4.2 -- Flow Diagram for CSCOV. Routine.



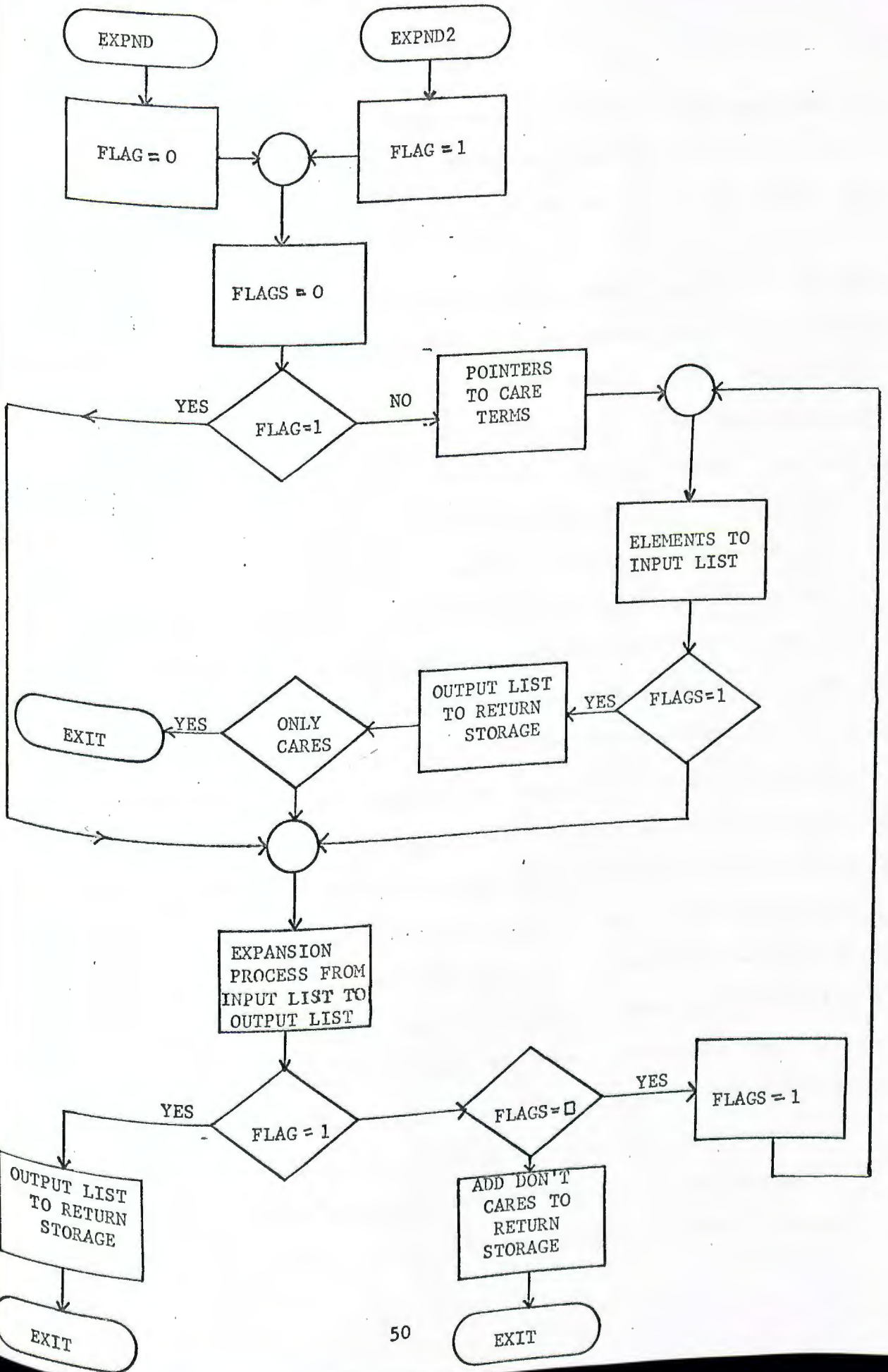
ERRORS. is used to print several error and pseudo error messages.

ESSPRI. is Keyserling's routine which generates the essential prime implicants and determines if the essential prime implicants cover the function. If they do, then COVPRT is tested to determine whether to send the information to the Tape cover storage or to print the cover.

EXPAN. has entries EXPAN. EXPAND, and EXPALL to expand the input function to canonical form. See Keyserling for details.

EXPND. (see Figure 4.3) has entries EXPND. and EXPND2. EXPND. is used to expand the input function to canonical form, expanding care terms and don't care terms separately. EXPND2 takes a single product term and expands it to canonical form. Initially, either the input care terms or the single product term is put into L, a push down stack. The expansion process is as follows. The top element in the stack is moved to the variable Y. If Y has a literal missing, Y with the literal added in primed form is put into the pushdown stack, and the same literal is added in unprimed form to Y. When Y has no more missing literals, it is stored in another stack represented by LL. If either stack gets too big, the stack is edited to condense out duplications. The expansion process is complete when the L stack is empty. If the call was to EXPND2., the output stack is condensed into the ARRAY vector and control is returned to the calling program. Otherwise the don't care terms are put into the input stack L, and then the output stack, LL, is copied into the input storage vector IN. Next the don't care terms are expanded into canonical form in LL. Finally, the don't care terms are added to the end of the input vector IN if they are not duplicates of a care term. NOCARE is set to the number of care terms generated, and NOTRM is set to the total number of care and don't care

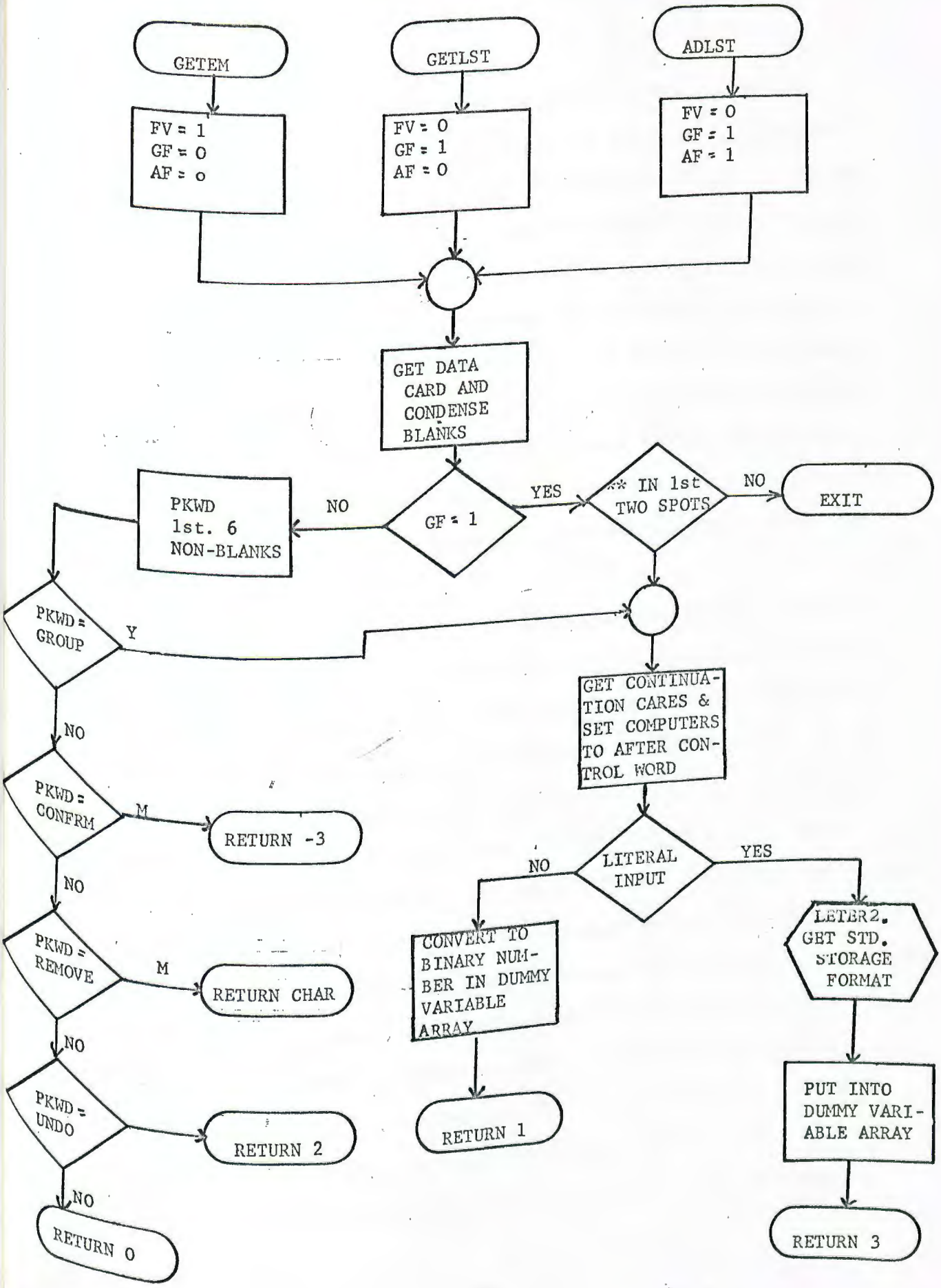
FIGURE 4.3 -- Flow Diagram for EXPND. Routine.



terms generated. The only fatal error is if either of the pushdown stacks are still full after condensing duplications out of them. If this happens, UNVDAT is set to non-zero, and control is returned to the calling program.

GETEM. (see Figure 4.4) has entries GETEM., GETLST., and ADLST. and serves as the input routine for both GROUP. routine instruction cards and result comparison cards. A typical call is GETEM.(ARRAY). Upon entry, GETFLG and ADFLAG are set to indicate which call was made. An input data card is peeked at and stored in the TEMP array. The input image is printed, and the blank characters are condensed from the TEMP array. If the entire card is blank, another card image is read in. If GETFLG indicates that a result comparison is requested, the first two non-blank characters in TEMP are checked to affirm that they are both asterisks (*). If not, control is returned to the calling program. If, on the other hand, the call was to GETEM., the first six characters in TEMP are condensed into PKWD to check which instruction has been made on the data card. If the control word is REMOVE, GETEM. returns the character to be removed after disposing of the input card image. If the control word is UNDO, GETEM. returns a value of 2 after disposing of the input card image. If the control word is GROUP, the rest of the input string is processed in the following manner. The TEMP array is scanned for a slash (/) which indicates that information is continued on the next data card. If there is additional data, it is read in and appended to the TEMP array. If the input information contains more than 1000 non-blank characters, the rest of the input information is read in, and an error message is produced. Having gotten the input information into TEMP, it must now be converted into internal storage

FIGURE 4.4 -- Flow Diagram for GETEM. Routine.



format. The first non-control character in TEMP (this excludes the control word GROUP, and any punctuation or the double asterisk if the call is to GETLST. or ADLST) is checked to determine if the information is in the form of standard product numbers or product terms. If working with standard product numbers, the input format is converted into decimal numbers using code generated by Piersall. These decimal numbers are stored in the ARRAY vector, and GETEM. returns a value of 1. If working with product terms (literal input) a call is made to LETER2. to convert the input character format into standard internal storage form in the TEMP array. If the call was to ADLST., the new terms are added to the ARRAY vector. Otherwise, the terms are moved into the ARRAY vector and then GETEM. returns a value of 3.

GROUP. (see Figures 4.5 and 4.6) has entries GROUP., GROUP3., GROUP4., GROUP7., and UNDO. and is the nucleus of the map manipulation routines. It stores the special print information, and manipulates the SAVER array which contains the information to be printed on the Karnaugh map. If the call is to GROUP. or GROUP3., RUTH. is called and the number of arguments in the calling list is stored in NRPAR. If UNVADT indicates an error condition, control is restored to the calling program. If PROPRI indicates that GROUP. has not been called, TERMS storage is set to zero, as are flags DUNYET and TEST. The storage of print characters, SAVER is set to all blanks, then the don't care terms are filled with "X" and the care terms with "1". If the call was to GROUP., the information about what standard product terms are to be grouped must be obtained. On the other hand, a call to GROUP3. provides this information, and the information is stored in ARRAY1, and the obtaining process is omitted. The obtaining process is dual insofar

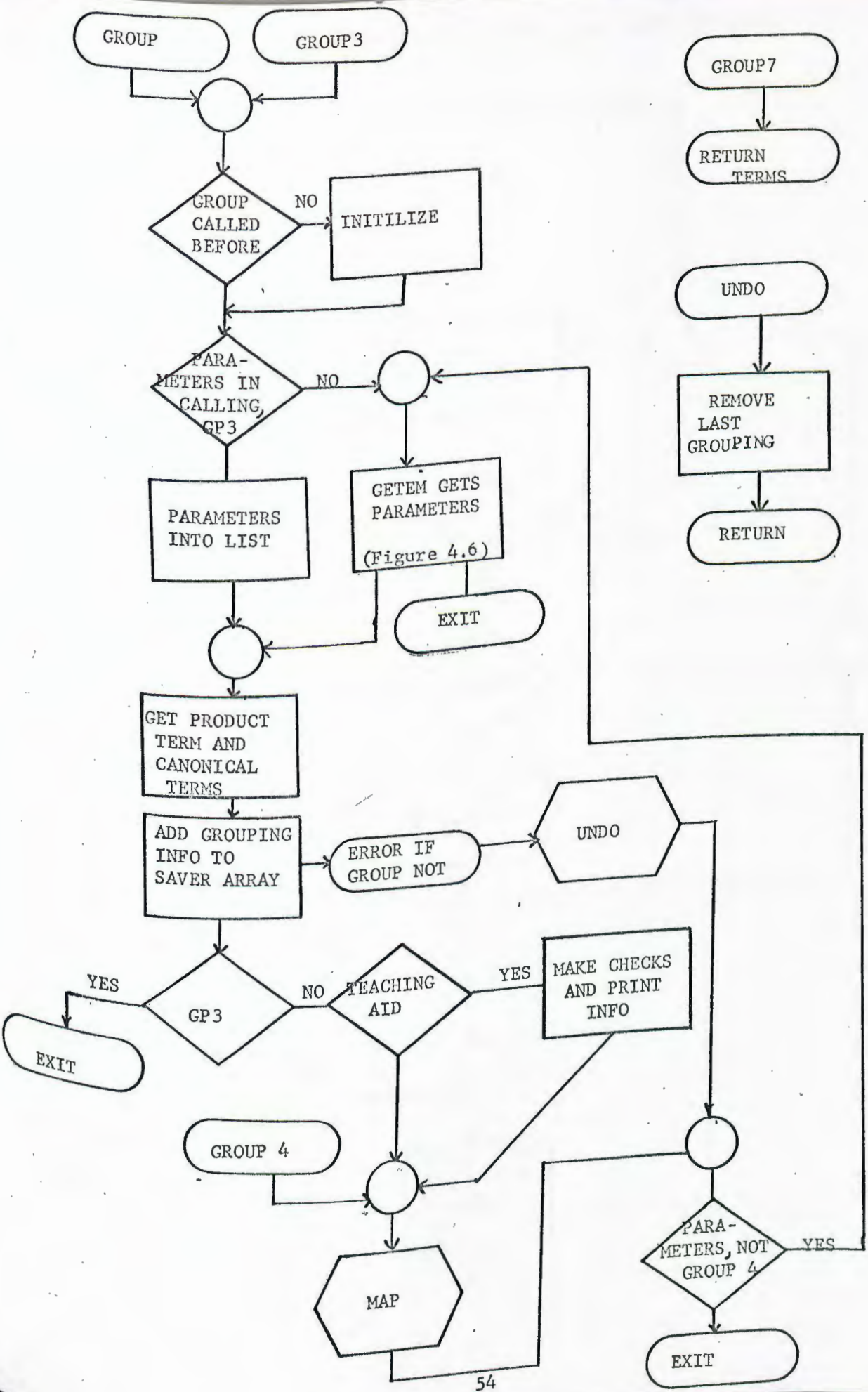
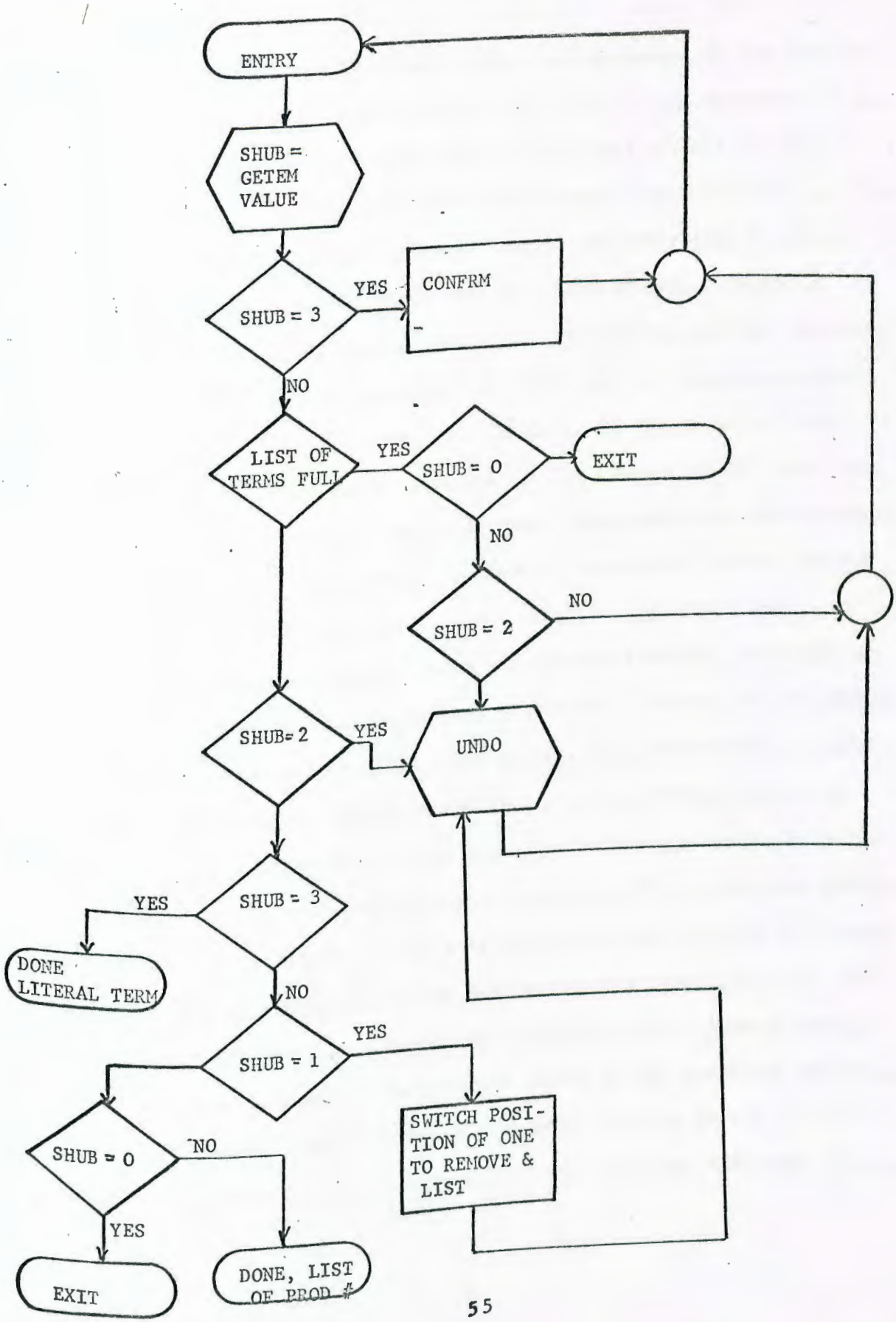

GROUP
GROUP3
GROUP7
RETURN TERMS
GROUP CALLED BEFORE
NO
INITILIZE
UNDO
PARA-METERS IN CALLING GP3
NO
REMOVE LAST GROUPING
RETURN
PARAMETERS INTO LIST
GETEM GETS PARAMETERS (Figure 4.6)
EXIT
GET PRODUCT TERM AND CANONICAL TERMS
ADD GROUPING INFO TO SAVER ARRAY
ERROR IF GROUP NOT
UNDO
YES
GP3
NO
TEACHING AID
YES
MAKE CHECKS AND PRINT INFO
EXIT
GROUP 4
MAP
PARA-METERS, NOT GROUP 4
YES
EXIT

FIGURE 4.6 -- Flow diagram for Input Section of GROUP. Routine.



as the information can be contained either as arguments to the function call, or as data cards. If the first case is true, the arguments to the call are put into LIST(1)...LIST(LIST). Otherwise, a call is made to GETEM. sending the LIST vector for the information to be stored in. The value of GETEM. indicates if the instruction was other than to make a grouping. If the value of GETEM. is -3, a call of CFRM.(TERMS) is generated to simulate a program command of CONFRM.(1), and the obtaining process is restarted. If the value of GETEM. is 3, a product term is stored in LIST(1), so it is stored in ARRAY1. If the value of GETEM. is 2, this means that the instruction was to UNDO, so a call of UNDO. is simulated. If the value of GETEM. is zero, that means that GETEM. did not find a recognizable instruction, so control is returned to the calling program. If the value of GETEM. is not equal to 1, at this point, it means that the command was to remove a character from the groupings, so control is transferred to the removal processer. Finally, the obtaining process has gotten a list of standard product numbers in LIST(1)...LIST (LIST). These numbers are converted into standard storage format in ARRAY(1)...ARRAY(LIST) these terms are then "or"ed together to eliminate variables that appear both primed and unprimed, and the resultant product term is placed in ARRAY1. At this point, the product term to be grouped is in ARRAY1, no matter how it was inputted to the GROUP. routine. If DUNYET indicates that the function is already covered, an error message is printed out, and control is returned either to the obtaining procedure if the call was to GROUP. without a parameter list, or to the calling program in any other case. If more groupings have been made than there are

special characters to represent groupings, an error message is printed, and if the call was to GROUP. with a parameter list control is returned to the calling program. Otherwise, control transfers to the obtaining procedure which scans instruction cards until it finds an UNDO instruction. This is built into the obtaining process and hinges on the value of TEST. The product term to be grouped is then stored in the TERMS stack, and then the product term is expanded to canonical form in ARRAY(1)...ARRAY(ARRAY). Then the actual map information is manipulated. The variable ADDM is set to 1B to be reset to 0B if this grouping covers any previously uncovered term. LIST is set equal to ARRAY to get the number of terms to be worked with. The map information manipulation is then carried out for each term until the list of terms is exhausted, or a term which is not in the function is found in which case a message is printed saying that the grouping is not in the function, and control is transferred to the undoing routine to undo the manipulation made so far for grouping being worked with. A pass for the Ith term is described. The decimal equivalent is put in LIST(I) by a call to DECIML. Whenever SAVER(LIST(I)) is a blank, it means that an attempt has been made to cover a term that is not in the function. X is set equal to SAVER(LIST(I)). If X is a "1" ADDM is set to 0B, and the special character from the TABLE of special characters is put in X. If X is already in a don't care status the special character from TABLE is added to X if there is room. If there is no room, and TEACHR indicates the program is being used as a teaching aid, the information of the standard product number and the special character are printed. Then SAVER(LIST(I)) is set to the new value of X. This completes the map information manipulation.

If ADDM indicates no new terms were covered, a warning message is printed. If TEACHR indicates that the program is not being used as a teaching aid, the following tests and printed information are bypassed. The standard product numbers in LIST(1)...LIST(LIST) are ordered and printed out. The term grouped is printed in literal form with the special character assigned to it. At this point, if the original call was to GROUP3., control is returned to the calling program. MYESS. is called to obtain the number and location of essential prime implicants. If the number of groupings is less than the number of essential prime implicants, a warning is printed out if the grouping is not an essential implicant. QMC. is called to get the location of the prime implicants, and if the grouping is not a prime implicant, a warning message is printed. That ends the special checks and information given if TEACHR indicates the program is being used as a teaching aid. The groupings made so far are printed, and a call is made to MAP2. to actually print the special map. Then, if the call was to GROUP. without a parameter list, control reverts to the obtaining section. Otherwise control is returned to the calling program.

The UNDO. entry determines by a call to RUTH. if there is a parameter indicating that the undoing process is to be done more than 1 time, and sets III equal to the number of passes through the undoing routine. TERMS, which indicates the number of separate terms grouped (the number of times the grouping process has been applied) is decremented by 1 for each pass, and the typical pass does the following: X is set equal to each member of the SAVER array, and then if X contains the special character in TABLE(TERMS) that character is replaced by a blank, and if the result of this replacement

is completely blank, X is set to a "1". Then the element of the SAVER array is reset to the new value of X. The removal section saves the number of TERMS in STORE, searches through the TABLE until the entry in TABLE is equal to the value returned by GETEM, makes an undoing pass using the correct value of TERMS to eliminate that particular grouping from the SAVER array, sets that particular TERMS(TERM) to zero, restores TERMS to the value saved in STORE, prints the function so far, prints the special Karnaugh map, and returns to the obtaining section of the program.

The entry to GROUP7. merely stores the TERMS array in the dummy variable array to the groupings made to the CONFRM. routine.

The entry to GROUP4. generates a call to MAP2. to print the Karnaugh map.

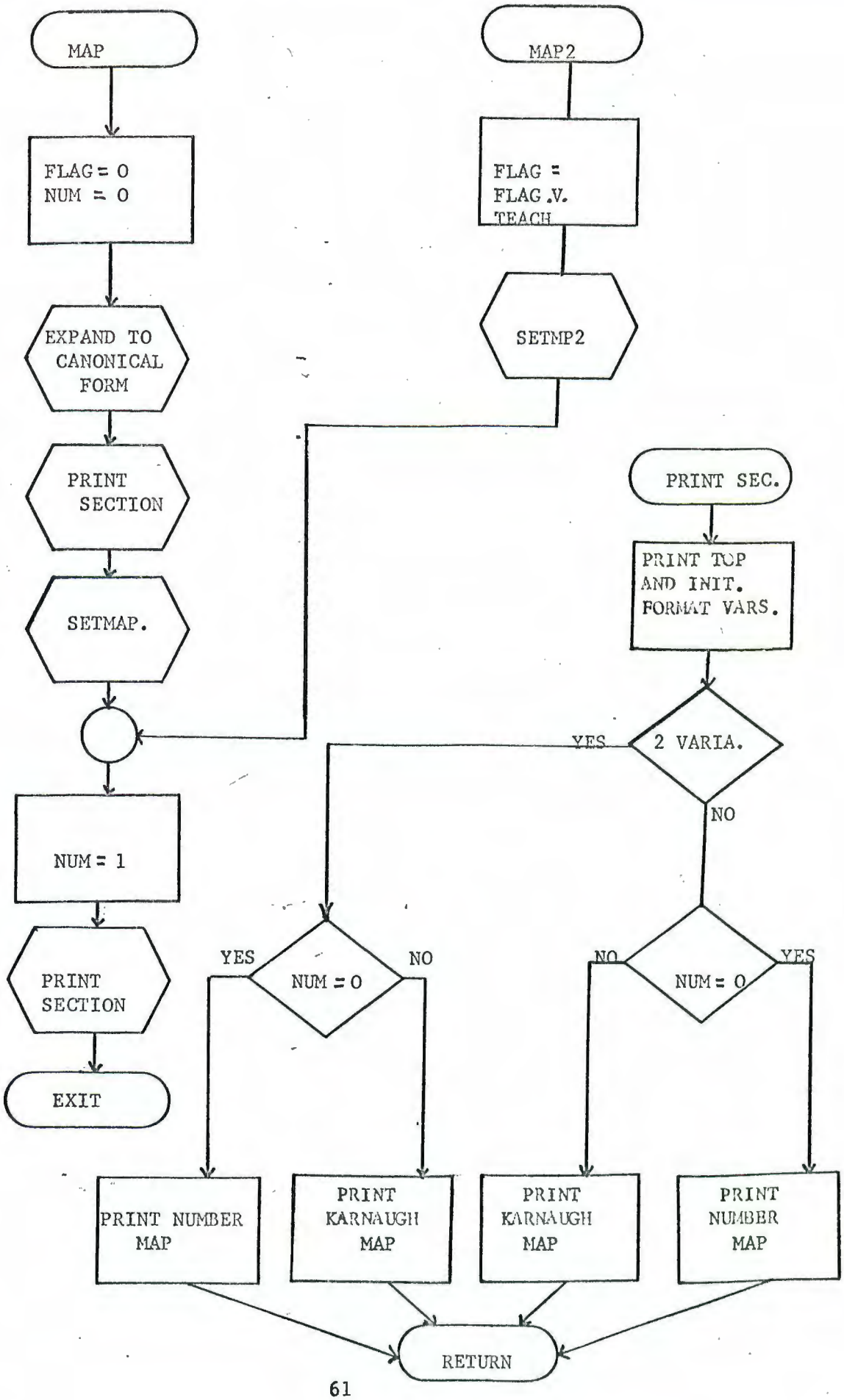
INTDEC. is part of Piersall's input output package. It has been modified by adding a list of dummy variables so that it can be used with the GETEM. routine as well. A typical call is: INTDEC.(L,NOCARE, IN, DEC) IN contains the input vector in a C1 format in IN(1)...IN(IN), which is converted into decimal equivalents in DEC(1)...DEC(DEC). L at return time contains the number of terms, numerically equal to DEC, and NOCARE contains the number of care terms.

ITER. is part of Keyserling's Minimization package written in UMAP, and performs iterated consensus. In case of storage overflow, both an error message, and work done so far are printed. The only modification made was to move the variable DUMMY out of PROGRAM COMMON storage to make room for other variables.

LETTER. is part of Piersall's input output package. It has been modified by adding a list of dummy variables and an additional entry, LETER2. so that

no new variable names are added to the list of variables. The routine takes IN(1)...IN(IN) as C1 format and converts it to IN(1)...IN(IN) in internal storage format with NOTRM indicating the total number of terms present, and NOCARE indicating the total number of care terms present. MAP. (see Figure 4.7) has entries MAP. and MAP2. These routines are used to do the actual printing of a Karnaugh Map. The form of the calls are MAP. or MAP2.(SAVER) where the dummy variable array, SAVER, has information about the special characters required for printing a map displaying the groupings. The program divides into two sections, namely the control section, and the actual print section. The print section is similar to an internal subroutine, only instead of using standard subroutine linkage, variable statement labels are used to preserve the return point. The control section operates as follows. If the call is to MAP., PROPRI is checked to determine if a map has been printed. If so, control is returned to the calling program. Then, if the function has not been expanded to canonical form, it is expanded to canonical form, and if TEACHR indicates that the program is being used as a teaching aid, the expanded function is printed out. Next a map showing numbering is printed out, by use of the print section. Then SETMAP. is called to store the proper characters in the M array for the printing, and the map is printed. If, on the other hand, the call is to MAP2., SETMP2. is called to provide the special characters in the M array, and the map is printed. The operation of the print section divides into two phases. The first step is to check NV for thenumber of variables in the function and set the format variables L, L2, L3, L4, B, N, and LL accordingly. While doing this, the proper top border of the map is printed.

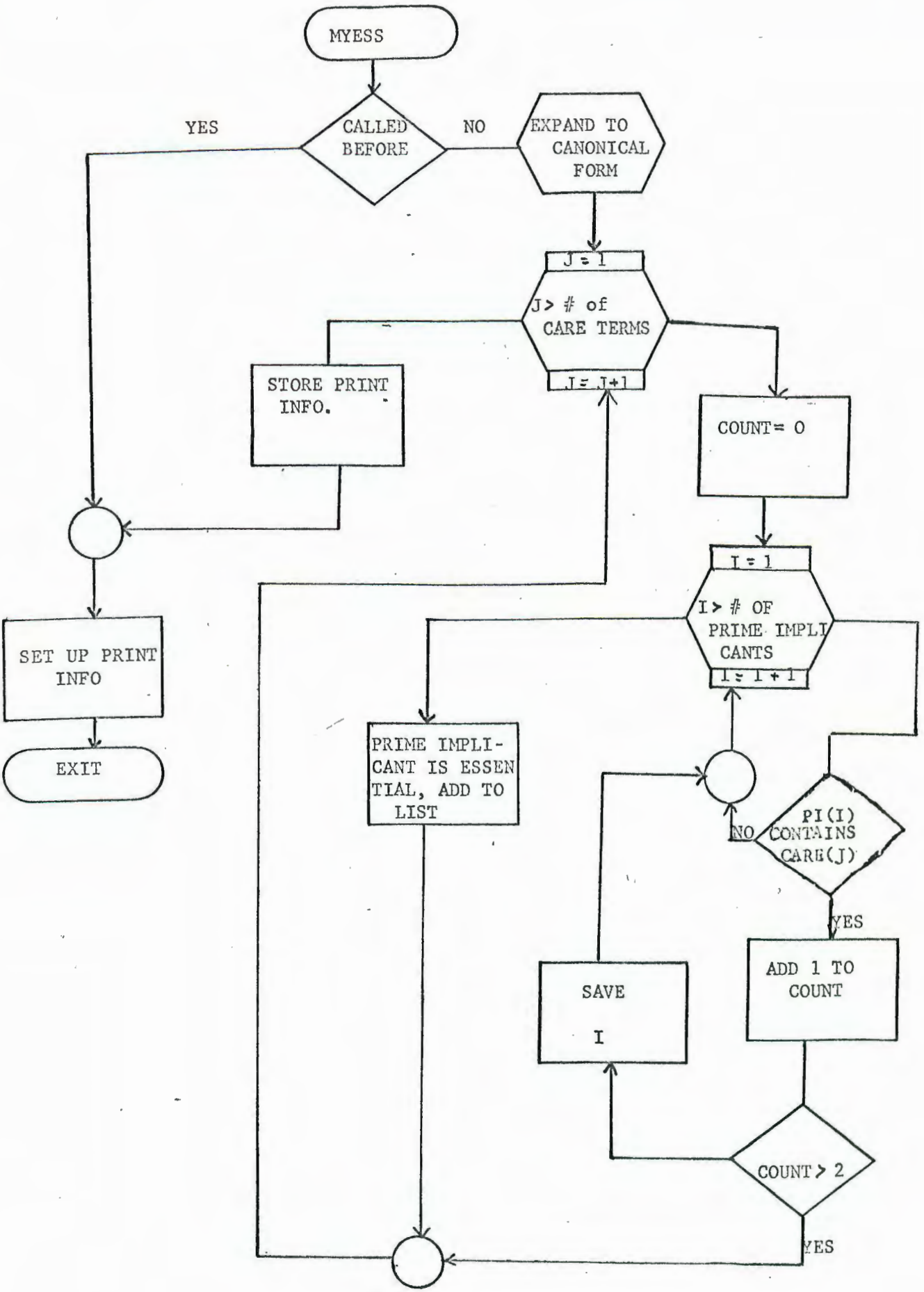
FIGURE 4.7 -- Flow Diagram for MAP. Routine.



Then TOPAS, a row of asterisks is printed. Next, the four lines of the map itself are printed, including the variable names along the side of the map. Finally, a bottom row of asterisks is printed, and then if there are 5 or 6 variables, the bottom row of variable identification is printed. The only error condition is when the number of variables is greater than six or less than two, and in these cases, a message is printed and control is returned to the calling program.

MYESS. (see Figure 4.8) is used to find the essential prime implicants of a function without destroying the list of prime implicants. If PROPRI indicates that MYESS. has not been called for the current function, the essential prime implicants are generated and their location is stored in the following manner. EXPND. is called to get a canonical expansion of the input function. QMC. is called to ensure that the prime implicants have been generated and to get the location in the A vector that the prime implicants are stored in. For each care term in the function, the following procedure is carried out: Each prime implicant is checked to see if that particular prime implicant contains the care term. If the care term is contained in only one prime implicant, then the prime implicant is essential. Then the list of prime implicants is rearranged so that the essential prime implicants are at the beginning of the list of prime implicants. The number of essential prime implicants and the starting location for their storage are saved in SAVENR and SAVEST respectively. Finally PROPRI is multiplied by 11 to indicate that the essential prime implicants have been generated. Then, (and this is what happens if the essential prime implicants had been generated) the print flags are set for the output routines by storing a 3 in LOC, and by storing SAVEST and SAVENR in PS and PN respectively. A value of one less than the starting location is returned for use in the GROUP. and CONFRM. routines.

FIGURE 4.8 -- Flow Diagram for MYESS. Routine.



NUMBS. is part of Piersall's input output package. It handles a sum of products function when the input is in the form of decimal numbers. Specified variable names are found or dummy variable names are generated.

ORDER. is used to order a list in increasing order. It works most efficiently for new items added to the end of an already ordered list. The typical call is: ORDER.(N;L) and this puts L(1)...L(N) into order.

OUTPUT. is part of Piersall's input output routines. It is used to for basic output. Several minor modifications were made. The routine new prints care terms and don't care terms separately for the input function and the expanded function generated by expnd. There are additional entries LIST. and PRINT.

PNT. is part of Piersall's input output package. It does the actual printing. It has the additional entry LST. A typical call is PNT.(A,ST,AA) This causes the name of the function to be printed, and it is assumed that the function is stored in A(ST)...A(AA) LST. causes the terms to be listed on separate lines and is used for printing lists of prime implicants or essential prime implicants. The entry PNT2. was added to use in printing a function without printing the name. This is most useful for printing don't care terms separately. The entry PRT. was added to print a single term without skipping a line. This is used for printing the term represented by a character on a Karnaugh map. In this case, the term to be printed is A, and ST gives how far over on the print line the term is to be printed.

QMC. is used to generate and store location of the prime implicants of a function. If PROPRI indicates that prime implicants haven't been generated then they are generated by a call to ITER. Then the output information in PS and PN is stored in SAVEST and SAVENR. Finally, PROPRI is multiplied

by 7 to indicate that the prime implicants have been generated this concluding the generation procedure. Print information is then stored in PS, PN and LOC from SAVEST, and SAVENR and putting a 1 in LOC and a value of one less than the starting location is returned for use in the GROUP., CONFRM., and MYESS routines.

READ. is part of Piersall's input output routine package. It is the basic input routine. The only modification was to add a list of parameters to the call to LETTER. to maintain compatibility with the modifications in the routine LETTER.

RUTH. is written in machine language. A typical call is RUTH.(N). RUTH. returns a 1 if the program which calls RUTH. was called with at least N actual parameters. Note that funny things can happen with this routine if the contents of index register 2 are changed between entering the calling routine and the actual calling of RUTH. This is gotten around by always calling RUTH. immediately upon entry if the information is necessary.

SORT. is part of Piersall's input output package. It sorts the variable names into alphabetical order.

SPRINT. is written in machine language by George W. Baltz. It is a buffered system input output routine compatible with the University of Maryland system utility programs. Several minor additions have been made to the routine. Dummy entry points to system library routines DFDP, DFMP, .PROCT, and .PRBCD have been added so that these routines which are required by the floating point conversion of the format scanner are not loaded since no floating point conversion is used in formatted input or output. The real low core error location was also changed to machine location 137 (octal) to provide for an automatic dump in case of an error return to system.

SETGRP. has entries SETGRP., PRIGRP., ESSGRP., AND RESETS. SETGRP. displays on a Karnaugh Map, the groupings implied by the form of the input function. Error conditions are when the function is read in as a list of standard product numbers, or when the number of product terms exceeds the number of special characters representing groupings on the Karnaugh map. The product terms are moved into the ARRAY vector. GROUP3. is called for each product term, and then GROUP4. is called to print the map of the groupings. The error conditons merely cause the function to return to the calling program. PRIGRP. checks PROPRI and returns to the calling program if PRIGRP. has been called already. QMC. is called to obtain the location of the prime implicants. The prime implicants are then moved into the ARRAY vector. If there are too many prime implicants, only the first NRGRP prime implicants are used. If ESSGRP. has not been called, PROPRI is adjusted to show a call to ESSGRP. has been made. If ESSGRP. has been called, then by setting PROPRI to indicate that GROUP. has not been called, negates the groupings made by ESSGRP. Finally, GROUP3. and GROUP4. are called as in SETGRP. ESSGRP. returns control if PROPRI indicates that ESSGRP. has been called. Otherwise, MYESS. is called to get the location of the essential prime implicants, which are then placed in the ARRAY vector. Then GROUP3. and GROUP4. are called as in SETGRP. RESETS. adjusts PROPRI to indicate that GROUP., PRIGRP., and ESSGRP. have all not been called.

SETMAP. has entries SETMAP. and SETMP2. A typical call is SETMAP.(F,M) in SETMAP. the M array is set to be a blank for terms not in the function, an X for don't care terms of the function, and a 1 for care terms of the function for printing the map. SETMP2. does the same thing if the mode is expert. In normal mode (i.e., that of a teaching aid) however, the F array is used to put the special characters in the M array for printing. The routine is called

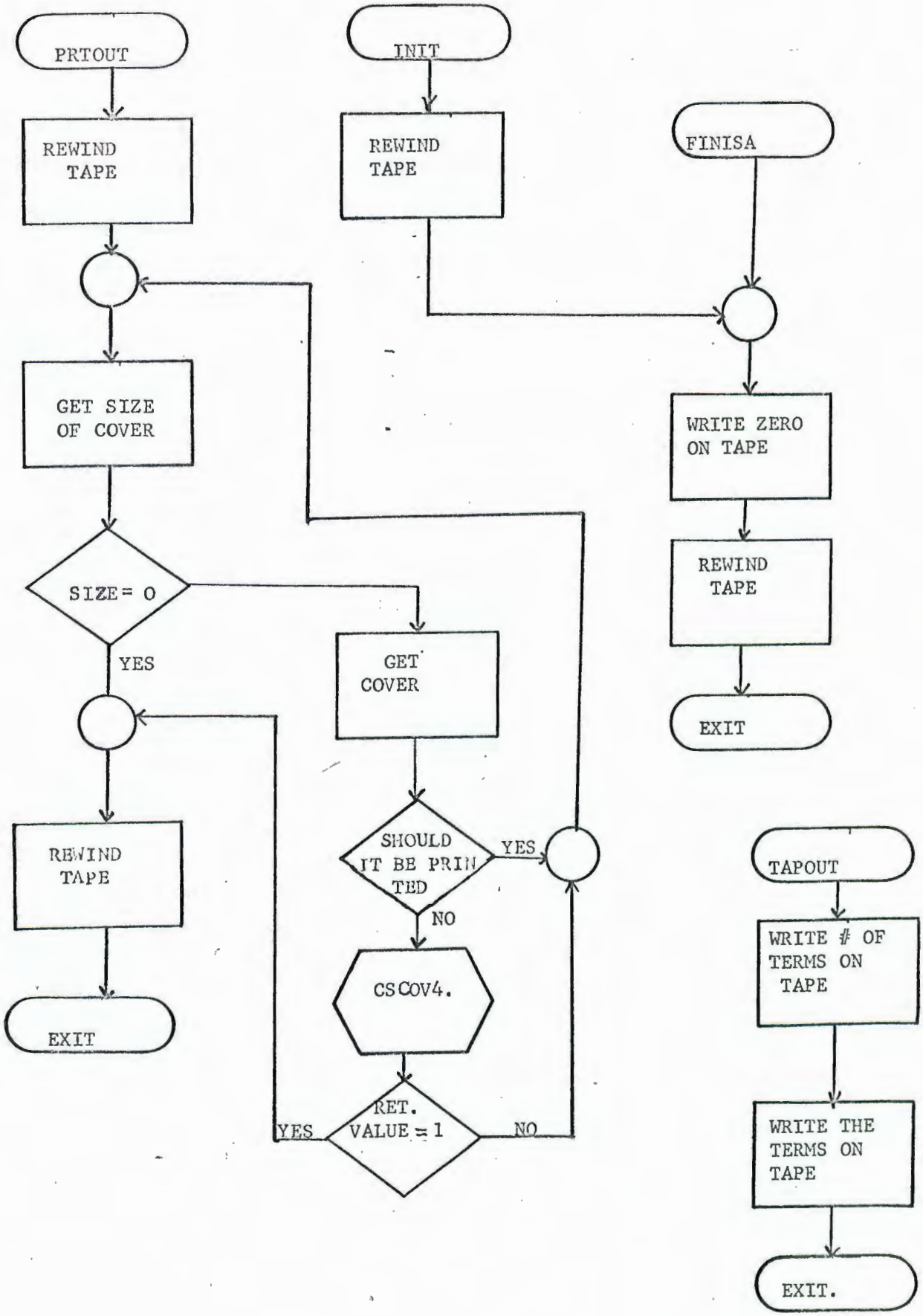
only by MAP and is used to set up the print characters for the actual printing of the map.

STORAJ. is part of Piersall's input output package. A typical call is: STORAJ.(A,AS) the routine takes A as a decimal standard product number and converts it to standard storage form in AS.

TAEX. is part of Keyserlings minimization routine package written in machine language. It has entries TAEX., ENUMR, PRALL., WGT., WGTA., and WGTB. which are used to generate minimum covers. In case of storage overflow, both an error message and the best cover generated so far are given. The following minor modifications were made. 4 variables used in no other routines were moved out of PROGRAM COMMON STORAGE. Instructions to skip printing messages if COVPRT is set to zero were added, along with a patch to call TAPOUT. instead of PRINT. if COVPRT is set to zero, thus sending the cover to tape storage rather than printing it.

TAPE. (see Figure 4.9) is used to store Keyserling's minimum covers on tape, so that they do not have to be generated several times for one function. The entry points are: INIT., TAPOUT., FINISA., and PRTOUT. The INIT. entry rewinds the data storage tape, and writes a zero on it to indicate that no covers are stored on the tape. Then the tape is rewound again and control is returned. FINISA. writes a zero on the storage tape to indicate that no more covers are on the tape, rewinds the tape and returns to the calling program. TAPOUT. writes two records on the data tape. The first record is a one word record indicating the number of terms in the cover. The second record contains the terms of the cover. PRTOUT. rewinds the tape. Then a record of tape is read to determine the size of the next cover on tape. If the size is zero, that indicates no more covers are on tape, so the tape is rewound and control is returned to the calling program. If

FIGURE 4. 9 -- Flow Diagram for TAPE. Routine.



the size is non-zero, the cover is read into the XX array. If the cover is to be printed, PNT. is called. If COVPRT indicates that the cover is to be compared with a user generated cover, CSCOV4. is called. If the return from CSCOV4. indicates a match has been found, control is returned to the calling program. Then, tape is read again looking for another cover and the same process is followed until either all covers are used or a match is found. USER. is provided for the user's convenience. It lets one call do the work of several. The entries are TEACH., EXPERT., READS., PRIMES., ESSENS., and COVERS. TEACH. sets the mode switch to the normal mode. EXPERT. sets the mode switch to the expert mode. READS. reads in and prints out a function by calling READ. and OUTPUT. PRIMES. generates a list of prime implicants by calling QMC. to make sure the prime implicants are generated, and then OUTPUT. to print the prime implicants. ESSENS. calls MYESS. to ensure that the essential prime implicants are generated, and then OUTPUT. to print them. COVERS. calls COVER. to print the minimum covers. The purpose of this routine is that when the programs are converted to real time use, this interface will provide for a lot less in the way of re-programming.

TABLE 4.8

VARIABLE USES BY ROUTINE

In CONFRM.

A	Keyserling's dynamic storage
AA	One less than the subscript of the location of the first prime implicant or essential prime implicant
ARRAY	Storage of User generated array of prime implicants, essentials, or elements of a cover.
TEST	Number of elements in machine generated array of prime Implicants essentials, or elements of a cover

In COVER.

FLAG	Stores information about value of COVPRT at entry.

In CSCOV.

AARRAY	Storage of user generated cover
BARRAY	Storage of machine generated covers
FOUND	Indicator if AARRAY has matched BARRAY

In EXPND.

ERRET	Where to go if lists overflow
FLAG	0 if in EXPND. 1 if in EXPND2.
FLAGS	0 if on first pass, 1 if on second pass.
L	Temporary storage for expanding terms.
LL	Temporary storage for expanding terms.
ARRAY	Dummy variable for EXPND2. to put expanded terms into.

In GETEM.

ADFLAG	1 if in ADLIST. 0 otherwise
GETFLG	0 if in GETEM. 1 otherwise
PKWD	first six non-blank characters on input card image.
TEMP	Temporary storage of input card image.
TEMP2	Temporary storage of input card image.
ARRAY	Dummy variable for standard product numbers to be returned to calling routine

In GROUP.

AA	One less than the subscript of the location of the first prime implicant or essential prime implicant.
ADDM	1 if grouping covers no new terms, 0 otherwise.
ARRAY	Storage in internal storage format of standard product terms which comprise grouping.
ARRAY1	Equivalent to ARRAY(1)
BEGIN	Location of start of product term obtaining procedure.

DUNYET	1 if function is covered, 0 otherwise.
FLAG	1 if GROUP. called with parameter list, 0 otherwise.
FLAG3	1 if call to GROUP3., 0 otherwise.
III	Number of time undoing process is to be done.
LIST	Storage in binary of standard product numbers which comprise grouping.
MASK	Vector to use as mask to separate out BCD characters.
MAX	entry to removal process
NRPAR	number of parameters in call to GROUP.
OUT	dummy statement label for exit from actual undoing routine
OUT1	entry to actual undoing routine
OUT2	exit from actual undoing routine
OUT3	exit from actual undoing routine
OUT7	exit from actual undoing routine
OVFO	list overflow error exit
SAVER	storage of character information to be printed in Karnaugh map.
SHUB	return value of GETEM.
STORE	Temporary storage for value of TERMS while in removal procedure.
TABLE	Table of special characters to be printed in Karnaugh Map.
TEMP	Temporary storage for value of TEACHR when TERMS array is full
TERMS	storage of the groupings made so far
TEST	1 if terms is full, 0 otherwise.
V1	First dummy variable used in GROUP7.

In MAP.

FLAG	0 if in MAP., 1 if in MAP2.
M	Storage of print characters
N	Storage of numbering for squares on map.
NUMPRT	0 if map with numbers being printed, 1 otherwise.

In MYESS.

FLAG	Number of prime implicants containing given standard product.
KTR	Subscript of first prime implicant containing given standard product.
NOESS	Number of essential prime implicants found so far.
SAVENR	Storage of program common variable PN which is number of essentials.
SAVEST	Storage of program common variable PS which is location of first one.
SI	working prime implicant in internal storage format.
AA	subscript of location of last prime implicant.

In QMC.

SAVENR	Storage of program common variable PN which is number of prime implicants.
SAVEST	Storage of program common variable PS which is location of first one.

In SETGRP.

ARRAY	Storage of terms to be grouped in internal storage format.

In TAPE.

XX	Machine generated cover read here from tape.
Y	1 if match has been found, 0 otherwise.

CHAPTER V

MODIFICATIONS AND EXTENSIONS

Needless to say, there are many modifications and extensions which can be made to this package of computer routines to enhance its operation. These modifications and extensions fall into several general categories. Initially, there are some remarks on problems encountered, along with some suggestions for alleviating these problems in future endeavors. Next, some suggestions for minor modifications of the package will be made with a view to indicating a particular approach to be taken. Thirdly, some particularly valuable extensions are discussed. Finally, the package is viewed in a rather abstract sense. That is to say, the function of the package is not the main concern, but rather the package as an entity is considered, and several remarks are made concerning machine independence and internal representation problems.

One of the major problems in the programming end of the endeavor was the lack of documentation of existing work. This could be alleviated to some extent in the future by rather than having several people work on several topics simultaneously, and having to depend on unproven work; to have each person given a finalized package to work with independently from other work that goes on with the constraint that any change in existing work be made in such a manner as to not change the function of the existing work. For example, the routine LETTER. had the additional entry LETER2. added. However, this additional entry point in no way changed the operation of the original LETTER. routine. Further help in this problem could be afforded by creating a greater awareness on the part of the individuals

working in specialized areas of those problems being worked on by other individuals which could in some way use the results of the first individual's work. For example, the generation of minimum covers requires a list of essential prime implicants, and once the essential prime implicants are found, it is no longer necessary for the covering problem to keep those terms in the list of prime implicants, yet in the map manipulation process it is necessary for both the lists of prime implicants and essential prime implicants to be full, and this produces a constraint on the covering routines which is not at all obvious to the individual working on the covering routines. Therefore, in addition to the requirement of additional documentation of existing work, there should also be a more rigorous definition of the specific problems being attacked with regard to an awareness of how the specific problem fits into the overall picture.

The suggestions for minor modifications of the package of routines fall into four areas, namely the increase or decrease of storage allocations, increase in the capacity of the mapping package, the changing of control words in the GETEM. routine, and the changes in command titles.

Presently, storage allocation is done with compile time parameters. Therefore, if the amount of available machine storage area is changes, it is a simple task to raise or lower the amount of storage area available by merely changing the parameter cards. For example, if the dimension of a certain array is set by the parameter UPPER, a conditional statement for terminating a process for the Ith member of the array is made as follows: WHENEVER I .G. UPPER, thus eliminating the necessity for changing

indices as well as array dimensions. Also, the necessity to work with more than twelve variables might arise. In this case, it would merely be necessary to raise the NVDIM parameter. Care should be taken to also lower the NRBITS parameter so that the product of these two parameters is always less than or equal to the size of one machine word. Note that in making any of these changes, special care must be taken in the case of the machine language subroutines which do not use these parameters.

There might be advantages to increasing the capacity of the map package to seven or even eight variables insofar as with the present scheme for displaying a six variable map, the horizontal symmetries would be vertical as well, and some symmetries would be fairly obvious. A good way of doing this would be to increase the UPPER parameter to either 127 or 255, and make a few changes in the map routine itself. These changes would be to increase the length of the N array, and modify the print section to make either two or four passes with appropriate index to print the extended map. In addition, the NUMMAP format would have to be changed to handle the integers over 100.

If the names of the GROUP instructions seem unwieldy, and other names are desired, these changes can be made by changing the conditional statements in GETEM. For example, if the word "pair" seemed more desirable than the word "group", the statement, WHENEVER PKWD .RS. 6 .E. $OGROUP$ would be changes to WHENEVER PKWD .RS. 12 .E.$OOPAIR$.

Disenchantment with command names is bound to occur. For example, there is the confusion inherent with the PRIMES command to generate a list of prime implicants. The word PRIMES can easily be taken for that of the

complementation operation. However under the current restriction of using legal MAD variable names, PRIMES seemed like a pretty good choice. If, however, other words seem more applicable, the USER interface program has been provided, and the only necessary change is the change in the name of the entry. For example, to change the command for generating a list of prime implicants to IMPLIC., merely change the MAD statement ENTRY TO PRIMES. to ENTRY TO IMPLIC. in the USER routine.

The major modifications that suggest themselves seem to fall into two different categories, namely things that the package of programs should do as a part of implementing computer aided design, and modifications to improve the package of routines as a package of routines irrespective of their specific function. In the former category are such problems as removing some of the restrictions on the input format such as handling parenthesis in literal form, and allowing block notation when the input is in the form of a list of standard product numbers; providing for use of the duality principle to enable the routines to work for products of sums; solution of the multiple output problem, and adding more modes for the program to operate on. The latter category involves such concepts as an interpretive monitor routine, minimization of routines using input or output, and making the routines more machine independent.

While fairly free of restrictions, the input formats still leave something to be desired. The most obvious, of course, is that a decoding scheme should be developed in order to generate a sum of products when the input is expressed with parenthesisation instead of in normal form. For example, the function could be specified as F=(AB12 * AC34)+D(A' + B12) which

could be converted to the normal form of F = AB12+ AC34+ A'D + B12D. This could be written as a list processing routine and patched into the current input program by inserting a call of PROC.(IN) in LETTER. as the first executable statement of the program. Another nice thing to have in the way of an option on the input format would be to allow block notation when specifying the function as a sum of standard products. For example, the input specification F=1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15 could then be specified as F = 1, 4...12, 15. This could also be written as a list processor, and could be inserted as the first executable statement in the routine NUMBS.

By making use of the duality principle, the addition of product of sums notation could be handled without too much difficulty. One way of approaching this problem would be to add a flag in Program Common storage which would indicate that the function was expressed in standard sum notation instead of standard product notation. This flag would warn the output routines to print the function as a product of sums rather than as a sum of products. This would entail writing additional output routines to print the function in the different form, and some sort of flag in the input routine, perhaps the word SUMS on the input card after the list of maxterm numbers. This might make for problems in inputting a function in literal form. As a sidelight to this, note that with this option, the complement of a function would be easily obtainable insofar as the maxterm numbers of the complement function are 2^N - the minterm numbers where N is the number of variables.

Another suggested direction for extension is the multiple output prime implicant function. This topic was mentioned briefly in the opening chapter. At present, to generate the multiple output prime implicants requires a good deal of work on the part of the user. He must first generate each function in canonical form, compare the terms, generate a set of dummy variables for function tags, type up data cards for this pseudo-function, and finally submit this pseudo-function for generation of prime implicants. One method of attacking this problem might be to begin with the concept of a multiple output monitor which would be told the number of functions. Then by calling READ. the appropriate number of times, the input functions could be moved into temporary storage. Next, the input functions would be compared term by term to generate a tag section, thus generating the pseudo-function, and finally, the pseudo-function could be minimized. However, a different approach might be valuable insofar as the above approach does not yield the proper multiple output minimum covers, which incidentally, is another logical extension of the package.

Finally, consideration should be given to the number of different levels at which the program operates. At present, there are two modes, namely that of student teaching aid, and as an aid to the design engineer. Insofar as there are a multitude of levels of knowledge that a student might posess, a logical extension to the package would be to incorporate more levels of use than are now present. For example, while a beginning student might be helped by voluminous printed information concerning the performance of the iterated consensus, a more highly advanced student who is thoroughly familiar with the iterated consensus would find the same

information rather useless and wasteful of both time and paper. This, and several other examples suggest a further subdivision of operating levels.

From the programming point of view, the necessity of a monitor routine to make the package closed is apparent. The eventual goal is to have the package operating in conversational mode in a real-time, user-interacting computing environment. Therefore, eventually code will have to be generated to determine that the user wishes a list of prime implicants when he types in on his teletype, "Give me a list of the prime implicants." Approaching a monitor routine in this manner should retain all the flexibility present with the present open ended system. In addition, it would relieve the user of the burden of making sure that his command program corresponds correctly to his data cards since he would then only have one package of cards to worry about instead of two. For the present, this would prove advantageous as the system could be run under JSYS with its inherent case in producing updated copies of existing work.

Another rather obvious step is to minimize the number of programs which require system input or output, thus minimizing the number of routines which would have to be modified when the conversion to a real-time system occurs. This could be done by incorporating all existing messages in the ERRORS routine, or by creating a MESAGE routine.

A third step, and perhaps the most important step, is to make the package of routines as much machine and installation independent as possible. This concept is a difficult one to reconcile insofar as it makes machine language routines practically out of the question which

takes away the efficiency inherent in assembler type codes and replaces it with the slower code generated by a compiler. It also makes it impossible to do certain things. For example, the routine RUTH. is impossible to write in most languages. Therefore, the above rule should have an exception to take into account the things that are impractical with compiler generated code, and also to permit short procedures, such as the actual performance of the consensus operation, to be written in machine language. These procedures, by virtue of their brevity should prove to be quite simple to duplicate for a different computer.

This machine independence also involves taking into account such things as computer word size. For example, in the 7094, the machine word is 36 bits which allows for twelve variables, each using three bits of the machine word. In the IBM System/360, however, the machine word is only 32 bits long, thus not providing enough storage room to store twelve variables at three bits per variable. This problem has been somewhat taken care of by the NVDIM and NRBITS parameters described in the previous chapter.

Finally, the problems inherent in using the MAD language must be remarked upon. Despite all of the advantages that MAD has over FORTRAN, and they are too numerous to list here, the unfortunate truth is that MAD is not nearly as universal as FORTRAN, although there appears to be at least one MAD compiler either available or under development for most of the computers in vogue today. However, the idea of eventually going through all of the code generated on all phases of the entire project with the aim of translating the code to a universal language should not be overlooked.

CHAPTER VI

SUMMARY

The opening chapter of this thesis describes the background thought and constraints under which work has proceeded in the design of a package of computer programs for working both with implementing logic design and with teaching the subject of logic design. This design highlights flexibility and ease of use insofar as the user is not required to learn a programming language to use the package of computer routines, nor must he learn a lot of arbitrary specifications in order to enter data into the computer. Furthermore, the various commands that are used have been chosen to parallel the thoughts of the user.

The second chapter deals with the Karnaugh map and it's applicability within the framework set down in the opening chapter. Notation is set up to reduce the covering problem to that of set system manipulation. The dynamic extension of the Karnaugh map, namely that of using unique characters to represent various groupings of product terms is introduced. An algorithm for obtaining a non-redundant cover is given both in terms of set system manipulation and manipulation of the information given on the Karnaugh map. A new, dynamic, approach to the minimization problem is detailed both in terms of the manipulation of set systems, and the manipulation of the information displayed on the Karnaugh map. A set of computer routines designed to accomplish these goals in a real-time, user-interacting, conversational-mode computing environment is described along with the requisite commands in the MAD language for use of these routines.

The fourth chapter provides detailed documentation of the several computer routines, and includes such information as storage requirements,

operating system requirements, along with detailed descriptions of exactly what occurs in each subroutine. This information is presented mainly for those interested in extending or modifying the package of routines

Finally, the fifth chapter delineates some suggestions for extensions of existing work, and new directions for investigation. In addition, some comments on the package of routines as an entity are made. All of the computer programs described here are available from the Electrical Engineering Program Library at the University of Maryland.

APPENDIX A

DESCRIPTION OF THE META LANGUAGE

The metalanguage is merely a notation to describe the syntax of a different notation. (13, 17) The needed definition symbols appear below.

is	is used to denote that the unit on the left is defined in terms of the unit on the right.
or	is used to show possible alternatives for a definition.
< >	is used to define a symbol which is defined in terms of another symbol.
$S_A^B \{ \}$	is used to show that whatever is enclosed between the braces is repeated anywhere from A to B times.

$L\{\langle s\rangle\}$ is $\langle s\rangle$ or $\{ L \{\langle s\rangle\} , \langle s\rangle\}$

Using the above definitions, the following are defined.

<letter> is A or B or ... or Z

<digit> is 0 or 1 or ... or 9

<null> is

<symbol1> is + or - or * or , or . or ' or $ or (or)

<symbol2> is = or /

<symbol> is <symbol1> or <symbol2>

<character> is <letter> or <digit> or <symbol>

<nchar> is <letter> or <digit> or <symbol1>

<rp> is <null> or) or ,

<break> is {{<null> or = or (or .} {<null> or (}}

<product number> is {1 or 2 or ... or 4095}

⟨variable name⟩ is $\{\langle \text{letter} \rangle\ S_0^5 \{\langle \text{digit} \rangle\}\}$

⟨product term⟩ is $S_1^*\{\{\langle \text{variable name} \rangle\}$ or $\{\langle \text{variable name} \rangle\ '\}\}$

⟨fname⟩ is $S_1^{100}\{\langle \text{nchar} \rangle\}$ or $S_1^{100}\{\langle \text{nchar} \rangle\ (\ L\ \{\langle \text{variable name} \rangle\}\ )\}$

⟨funct1⟩ is L $\{\langle \text{product number} \rangle\}$ $\{\langle \text{null} \rangle$ or $\{,(\ L\ \{\langle \text{product number} \rangle\}\ )\}\}$

⟨funct2⟩ is $L+\{\langle \text{product term} \rangle\}$ $\{\langle \text{null} \rangle$ or , $\{L+\{\langle \text{product term} \rangle\}$

or L $\{\langle \text{product term} \rangle\}\}\}$

⟨group command⟩ is UNDO or CONFIRM or CONFRM

⟨group instruction⟩ is ⟨group command⟩ or $\{$REMOVE ⟨char⟩$\}$ or

$\{$GROUP ⟨break⟩$\{\{\langle \text{product term} \rangle$ or L $\{\langle \text{product number} \rangle\}\}$ ⟨rp⟩$\}$

⟨confirm instruction⟩ is ** $\{L+\{\langle \text{product term} \rangle\}$ or L $\{\langle \text{product term} \rangle\}\}$

⟨identifier⟩ is $\{\langle \text{letter} \rangle\ S_0^5\ \{\text{digit}\}\}$

⟨type⟩ is $\{\{S_2^4\ \{\langle \text{letter} \rangle\}\}$ or $\{\langle \text{letter} \rangle \langle \text{letter} \rangle\ \{S_1^2\ \{\langle \text{digit} \rangle\}\}\}\}$

⟨input⟩ is ⟨identifier⟩ or ⟨variable name⟩ or 0 or 1

⟨element⟩ is $\{\{\langle \text{type} \rangle$ or ⟨null⟩$\}$ ⟨identifier⟩ L $\{\langle \text{input} \rangle\}\}$

APPENDIX B

NOTES ON RESTRICTIONS OR ERRORS ON INPUT FORMATS

1. The location on the card that information is punched in is not important. The input routines will automatically left justify and condense out blanks.

2. To indicate that information is carried over to the next data card a slash (/) is used. Information punched to the right of the slash is ignored. A terminating asterisk(*) is deleted if it is used. The input function can have as many as INDIM (presently 1023) non-blank characters.

3. If the input is in the form of a list of standard product numbers, the variables specified in the function name, if there are any variables so specified, are used in the order specified to print the function. If the number of variable names specified is less than the number required, the input routines automatically supply dummy variable names.

4. When the input is in the form of product terms, any specification of variable names in the function title is ignored.

5. On a list of product numbers form of input, don't care terms are preceded by a left parenthesis. The right parenthesis, a comma after the last don't care term, and a comma before the left parenthesis are all optional.

6. On a list of product terms, the terms after the first comma are all considered to be don't care terms. They may be separated by plus signs, commas, or both.

7. Adjacent plus signs, commas, and primes are treated as single. Terms equal to zero (AA' for example) are not added to the function.

8. A variable name must be a single letter followed by a string of digits. If the string of digits makes the length of the variable name greater than six characters, the variable name is truncated to six characters. If a function name is too long, it is truncated to NAMDIM (presently 100) characters.

9. GROUP instruction cards must have the instruction word as the first non-blank spaces on the card. Trailing commas and parenthesis are ignored. Punctuation marks between the control word and the data itself are ignored except in the case of a REMOVE instruction where the unique character might well be a punctuation mark.

10. Product terms for both group commands and result testing lists are processed by the same routines as the input functions, so the same restrictions apply with respect to double commas, double primes and double plus signs.

11. In result testing routine lists, a function name must not be used.

12. Simulator input features.

a. An unrecognizable card causes the card to be ignored and a message printed.

b. An input of 0 is assumed if an INPUTS card does not specify a zero or one.

c. OUTPUTS cards should go at the end of a circuit insofar as an identifier that has not been defines when the OUTPUTS card is read is deleted.

d. When an identifier is multiply defined, the last occurance is used.

e. Specified inputs which are also gate outputs, are treated as gate outputs rather than specified inputs.

f. If an input constant is specified more than once, the last occurance is used.

g. Identifiers or inputs more than six characters long are truncated. Resulting duplications are not detected.

h. Identifiers and inputs containing more than one letter are deleted.

i. A gate with no input has an output of 0.

j. Specifications are continued onto a second card by punching a digit in column 11 of all continuation cards.

k. Remark cards can be inserted into the circuit with an R in column 11.

APPENDIX C

EXAMPLES

Two sample programs are shown. The first demonstrates the use of the map manipulation routines, and the second demonstrates the use of the simulator. The program demonstrating the map package illustrates several things. There are four examples.

The first demonstrates a six variable map with numbers printed for reference, and then the map of the function. Next, six groupings are made by one program command. Notice that grouping "F" is redundant insofar as there is no square on the map which contains just an "F". Then the result tester confirms that the cover is indeed not a minimum cover.

The second example uses a function of four variables with some don't care conditions. Notice that the care and don't care terms are printed separately, and the don't care terms are represented on the map by an "X". Several groupings are made, indicating some bad choices such as groupings not contained in the function, and groupings which are not prime groupings. This example also demonstrates undoing both the last grouping made and an intermediate grouping.

The third example demonstrates the algorithm for further reducing an irredundant covering set. Initially, five groupings cover the function. After removing grouping "A" a good choice seems to be the grouping which will cover squares 1 and 5 insofar as this grouping will make grouping "B" redundant.

The fourth example merely demonstrates a five variable map.

The simulator examples show the use of the simulator, and demonstrate that more than one switching function can be compared with a logic circuit.

SAMPLE PROGRAM
****

```
$IBSYS
$*    2030  C1  A - 5    RING OUT
$PAUSE
$EXECUTE        MAMOS
$ID    SHUB*305/03/101*2MINUTES*100 PAGES                    476005
$COMPILE MAD,EXECUTE
            NORMAL MODE IS INTEGER
            PROGRAM COMMON DUMMY(12188)
START       CONTINUE
            READS.
            PRIMES.
            ESSENS.
            COVERS.
            PRIGRP.
            CONFRM.(1)
            READS.
            GRCUP.(4,5)
            GRCUP.
            READS.
            SETGRP.
            GRCUP.
            READS.
            MAP.
            TRANSFER TO START
            END OF PROGRAM
$BINARY(11)
```

```
EXAMPLE ONE =3,4,5,7,9,13,14,15,60,61,62,63
PROBLEM TWO =3,4,5,7,15,2,11(13,14)
GROUP  5,7,13,15
GROUP  2,3
GROUP.3,15
GROUP  .(9,11
 U   N   D   C
GROUP 9,1  1,13,15,
R  EMOVE   B
CONFIRM
EXAMPLE THREE=A'B'C'+A'CD+BCL+ACD'+B'C'D'
REMOVE A
GROUP A'C'D
REMOVE B
CONFPM
EXAMPLE FOUR=T12345'W12X34+T12345YZ
```

****

EXAMPLE ONE =3,4,5,7,9,13,14,15,60,61,62,63
DUMMY NAMES USED.

THE VARIABLE NAMES ARE A, B, C, D, E, F

THE INPUT FUNCTION IS ...

EXAMPLE ONE = A'B'C'D'EF + A'B'C'D'EF' + A'B'C'DE'F + A'B'C'DEF' + A'B'CD'E'F + A'B'CDE'F + A'B'CDEF' + A'B'CDEF + ABCDE'F'
+ ABCDE'F + ABCDEF' + ABCDEF

THE PRIME IMPLICANTS ARE...

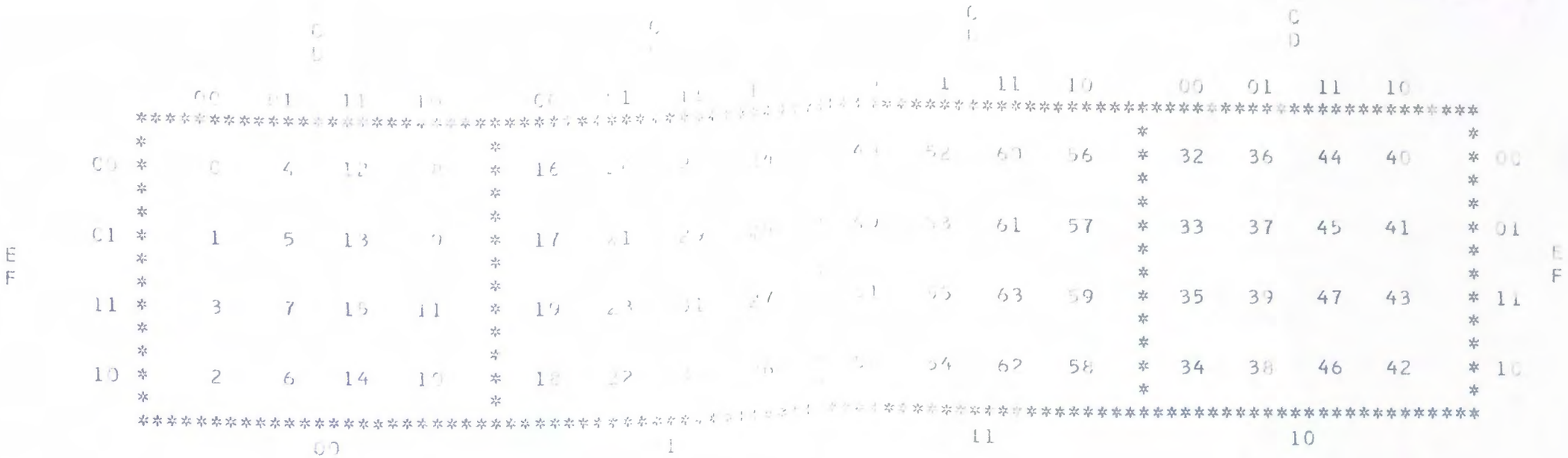
ABCD
A'B'C'DE'
A'B'CE'F
A'B'DF
A'B'CDE
A'B'C'EF

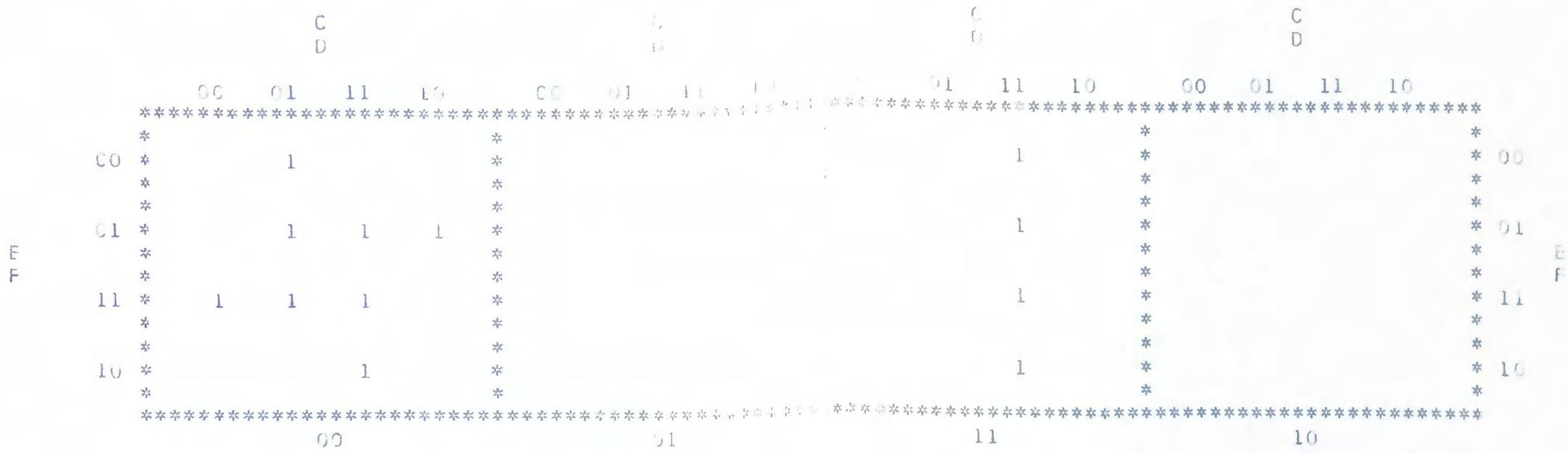
THE ESSENTIAL PRIME IMPLICANTS ARE...

ABCD
A'B'C'EF
A'B'CE'F
A'B'CDE
A'B'C'DE'

THE COVERS ARE...

EXAMPLE ONE = A'B'C'EF + A'B'CE'F + A'B'CDE + ABCD + A'B'C'DE'


CD
EF
00 01 11 10
00 01 11 10
00 4 12 8 16 20 28 24 48 52 60 56 32 36 44 40
01 1 5 13 9 17 21 29 25 49 53 61 57 33 37 45 41
11 3 7 15 11 19 23 31 27 51 55 63 59 35 39 47 43
10 2 6 14 10 18 22 30 26 50 54 62 58 34 38 46 42
00 01 11 10


CD
EF
00 01 11 10
00 01 11 10
00 1 1
01 1 1 1 1
11 1 1 1 1
10 1 1
00 01 11 10

...OUPINGS IMPLIED BY THE PRIME IMPLICANTS.

THE CHARACTER A ON THE MAP REPRESENTS THE TERM ABCD

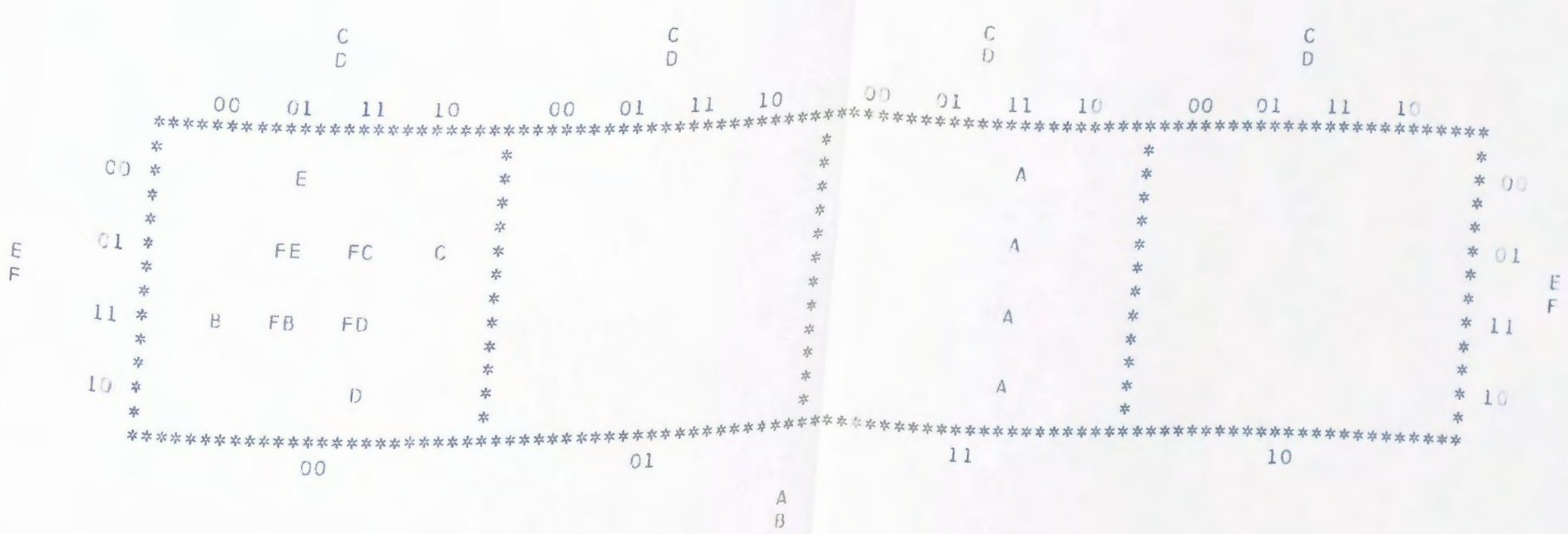
THE CHARACTER B ON THE MAP REPRESENTS THE TERM A'B'C'EF

THE CHARACTER C ON THE MAP REPRESENTS THE TERM A'B'CE'F

THE CHARACTER D ON THE MAP REPRESENTS THE TERM A'B'CDE

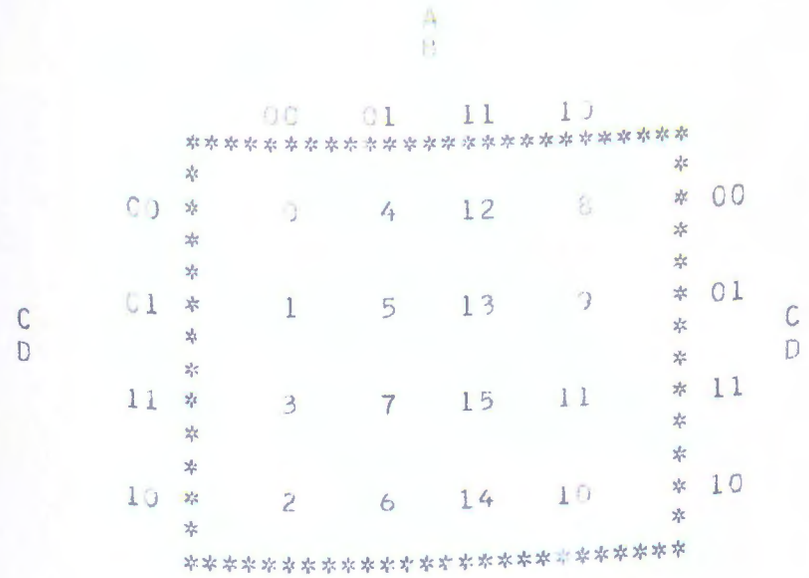
THE CHARACTER E ON THE MAP REPRESENTS THE TERM A'B'C'DE'

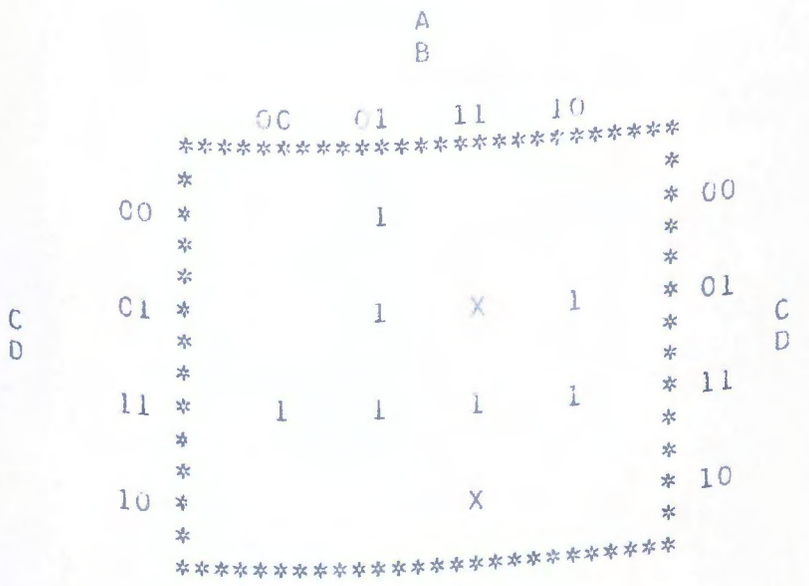
THE CHARACTER F ON THE MAP REPRESENTS THE TERM A'B'DF



```
YOU ARE IN THE RESULT TESTING ROUTINE.
YOU HAVE NOT GIVEN A CORRECT MINIMUM COVER.
YOUR COVER IS  ...
    EXAMPLE ONE   = ABCD + A'B'CDE + A'B'C'DE' + A'B'DF + A'B'C'EF + A'B'CE'F
THE MACHINE COVERS ARE ....
THE COVERS ARE...
    EXAMPLE ONE   = A'B'C'EF + A'B'CE'F + A'B'CDE + ABCD + A'B'C'DE'
```

```
**********************************************************************************************************************************
PROBLEM TWO =3,4,5,7,15,9,11(13,14)
DUMMY NAMES USED.
THE VARIABLE NAMES ARE     A, B, C, D
THE INPUT FUNCTION IS ...
    PROBLEM TWO   = A'B'CD + A'BC'D' + A'BC'D + A'BCD + ABCD + AB'C'D + AB'CD
                AND THE FOLLOWING DON'T CARE TERMS
    ABC'D + ABCD'
```

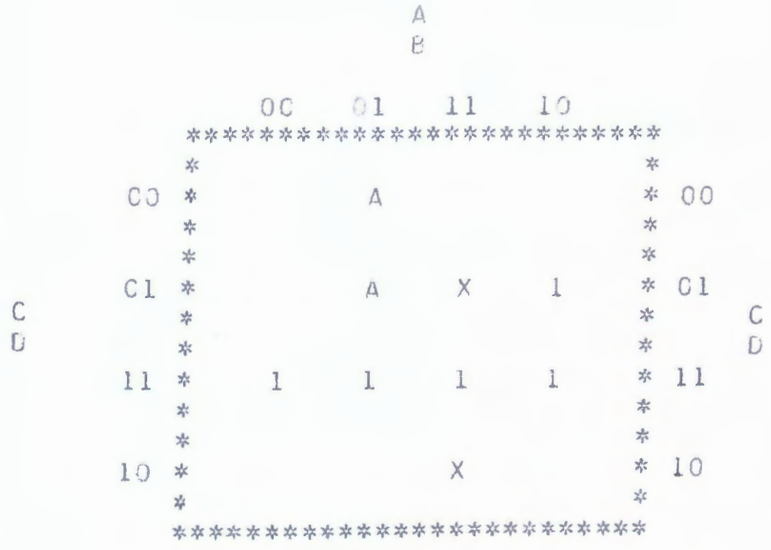

A
B
00 01 11 10
C
D
00
01
11
10
0 4 12 8
1 5 13 9
3 7 15 11
2 6 14 10


A
B
00 01 11 10
C
D
00
01
11
10
1
1 X 1
1 1 1 1
X

THE MINTERMS INCLUDED IN THIS CALL OF GROUP ARE...
 4, 5,

THE CHARACTER A ON THE MAP REPRESENTS THE TERM A'BC'
THE FUNCTION SO FAR IS

    A'BC'



GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

GROUP 5,7,13,15

WE HAVE A GROUPING

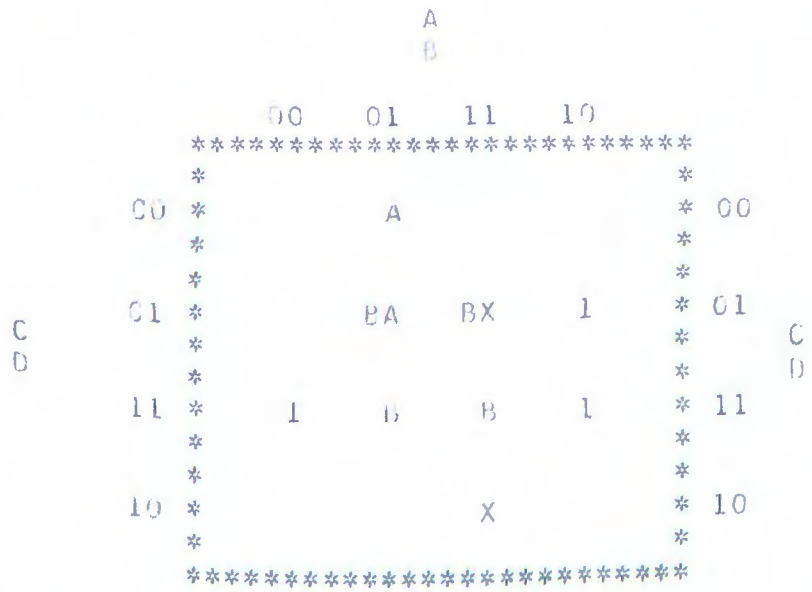
THE MINTERMS INCLUDED IN THIS CALL OF GROUP ARE...
5, 7, 13, 15,

THE CHARACTER B ON THE MAP REPRESENTS THE TERM BD
****GROUPING IS NOT AN ESSENTIAL PRIME IMPLICANT.
AND THERE ARE STILL ESSENTIAL PRIME IMPLICANTS NOT FOUND.

SUGGEST YOU CALL ROUTINE UNDO.

THE FUNCTION SO FAR IS

A'BC' + BD



GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

GROUP 2,3

WE HAVE A GROUPING
THIS GROUPING IS NOT INCLUDED IN THE FUNCTION

GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

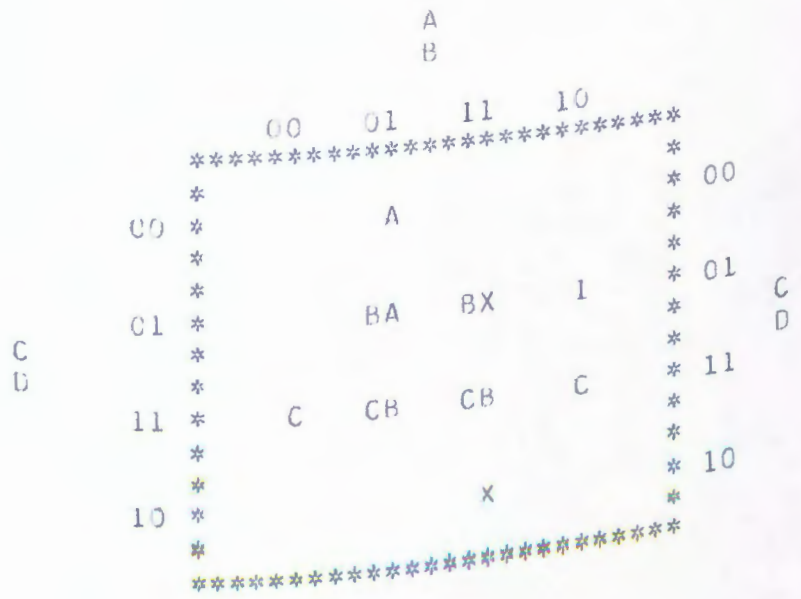
GROUP.3,15

WE HAVE A GROUPING

THE MINTERMS INCLUDED IN THIS CALL OF GROUP ARE...
3, 7, 11, 15,

THE CHARACTER C ON THE MAP REPRESENTS THE TERM CD
THE FUNCTION SO FAR IS

A'BC' + BD + CD



GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

GROUP .(9,11

WE HAVE A GROUPING

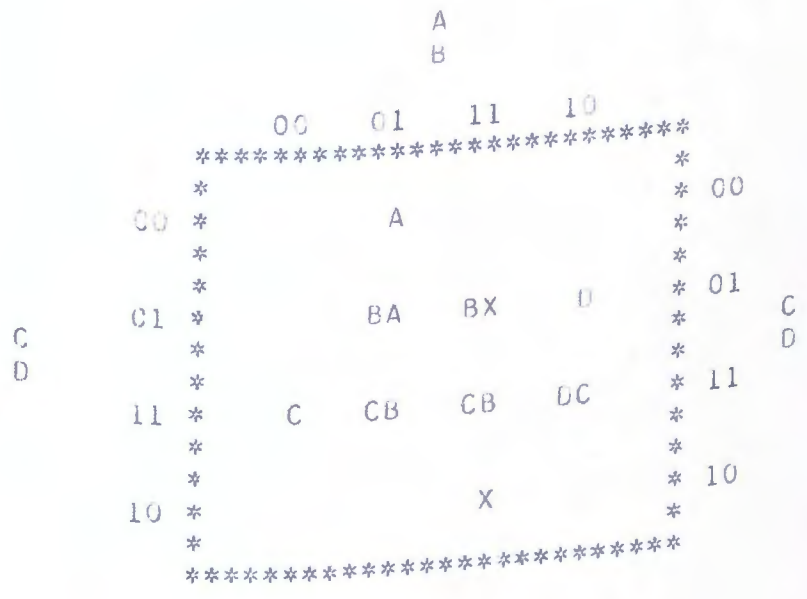
THE MINTERMS INCLUDED IN THIS CALL OF GROUP APE...
9, 11,

THE CHARACTER C ON THE MAP REPRESENTS THE TERM AB'D
*****THE GROUPING IS NOT A PRIME IMPLICANT.

SUGGEST YOU CALL ROUTINE UNDO.

THE FUNCTION SO FAR IS

A'BC' + BD + CD + AB'D



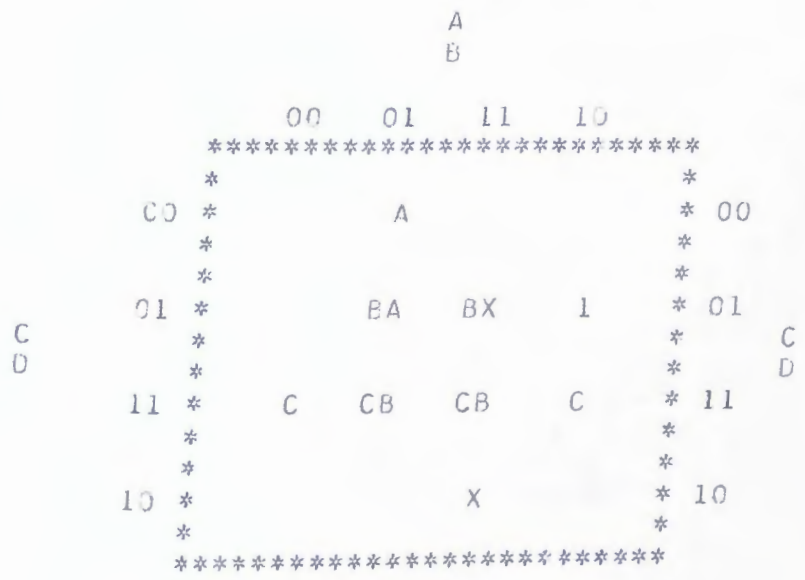
GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

U N D O

WE WISH TO UNDO.

THE FUNCTION SO FAR IS

A'BC' + BD + CD



GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

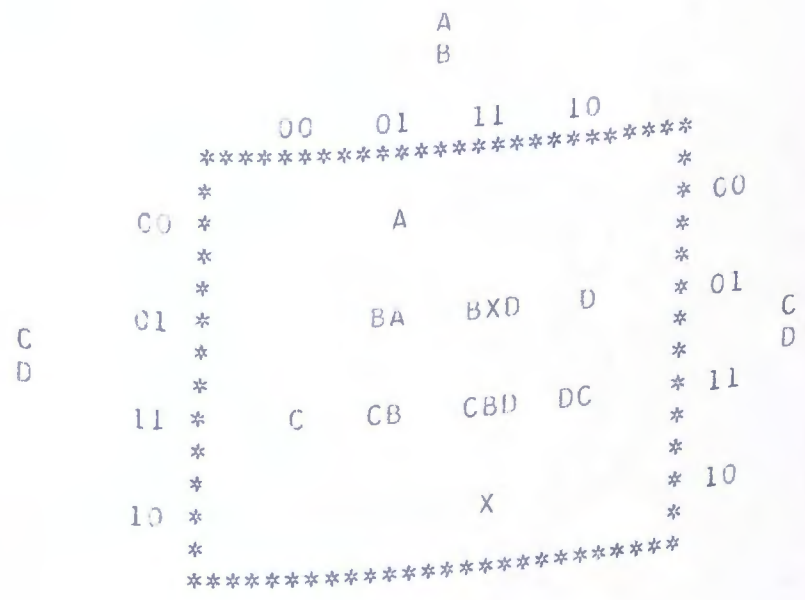
GROUP 9,1 1,13,15,

WE HAVE A GROUPING

THE MINTERMS INCLUDED IN THIS CALL OF GROUP ARE...
9, 11, 13, 15,

THE CHARACTER C ON THE MAP REPRESENTS THE TERM AD
THE FUNCTION SO FAR IS

A'BC' + BD + CD + AD



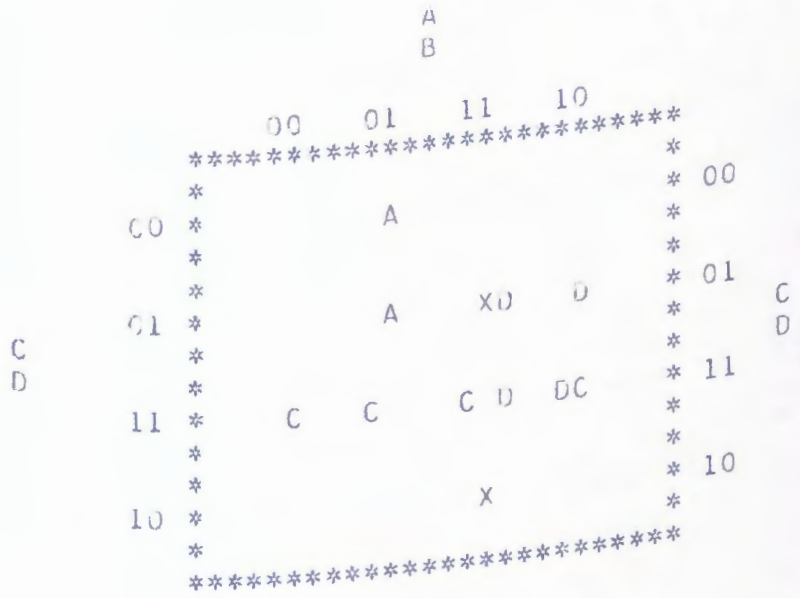
GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

R EMOVE B

WE WISH TO REMOVE THE GROUPING REPRESENTED BY THE CHARACTER B

THE FUNCTION SO FAR IS

A'BC' + AD + CD



GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

CONFIRM

WE WISH TO CONFIRM.

YOU HAVE GIVEN A CORRECT MINIMUM COVER.

GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

EXAMPLE THREE=A'B'C'+A'BD+BCD+ACD'+B'C'D'

WE DO NOT HAVE A GROUPING.

****

EXAMPLE THREE=A'B'C'+A'BD+BCD+ACD'+B'C'D'

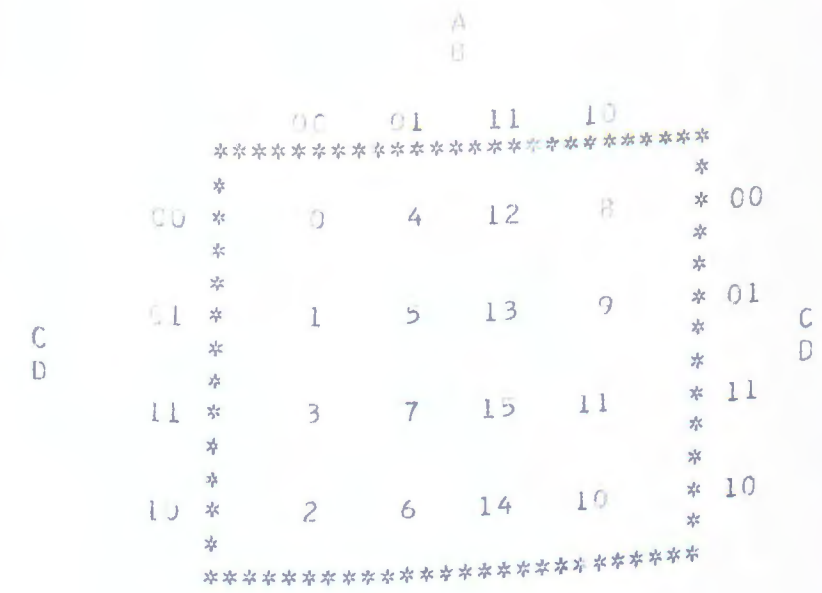
THE VARIABLE NAMES ARE A, B, C, D

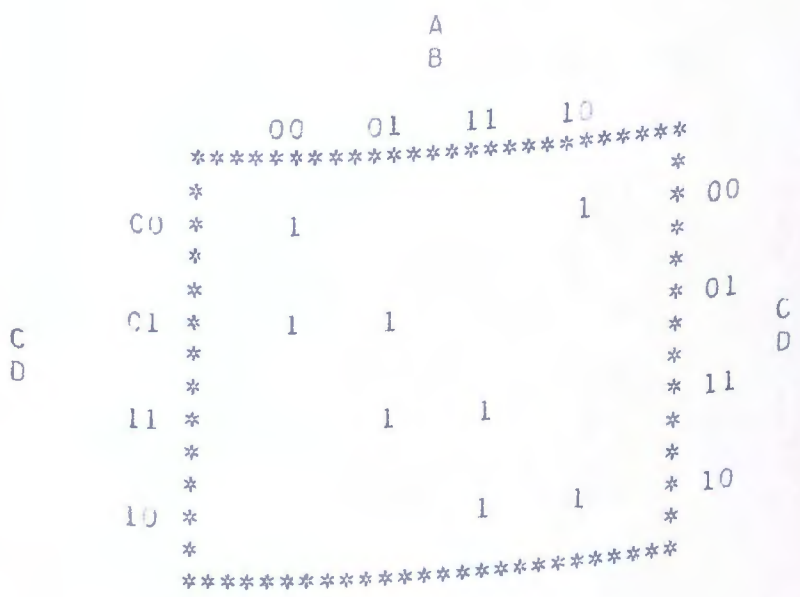
THE INPUT FUNCTION IS ...

EXAMPLE THREE = A'B'C' + A'BD + BCD + ACD' + B'C'D'

THE EXPANDED FUNCTION IS...

EXAMPLE THREE = ABCD + A'BCD + A'BC'D + A'B'C'D + ABCD' + AB'CD' + AB'C'D' + A'B'C'D'


A
B
00 01 11 10
CD
00 0 4 12 8 00
01 1 5 13 9 01
11 3 7 15 11 11
10 2 6 14 10 10
C
D


A
B
00 01 11 10
CD
00 1 1 00
01 1 1 01
11 1 1 11
10 1 1 10
C
D

THE GROUPINGS IMPLIED BY THE INPUT FUNCTION.

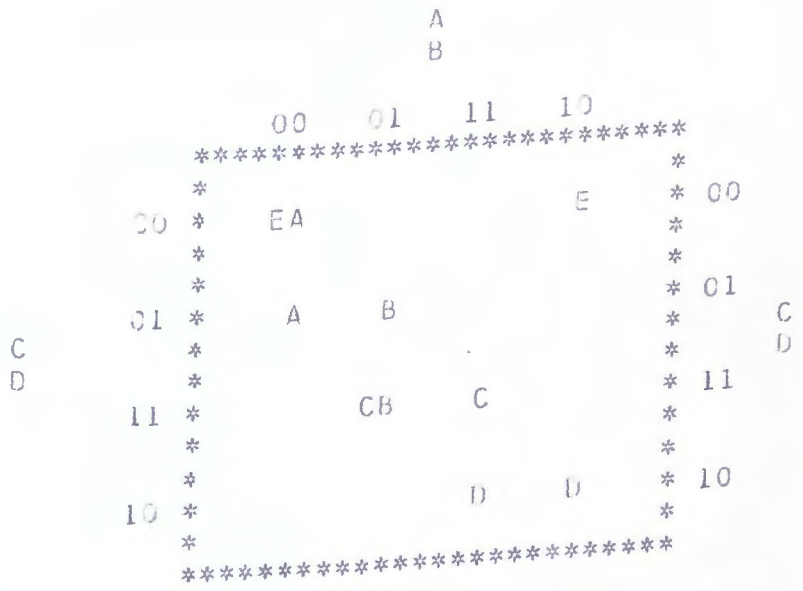
THE CHARACTER A ON THE MAP REPRESENTS THE TERM A'B'C'

THE CHARACTER B ON THE MAP REPRESENTS THE TERM A'BD

THE CHARACTER C ON THE MAP REPRESENTS THE TERM BCD

THE CHARACTER D ON THE MAP REPRESENTS THE TERM ACD'

THE CHARACTER E ON THE MAP REPRESENTS THE TERM B'C'D'



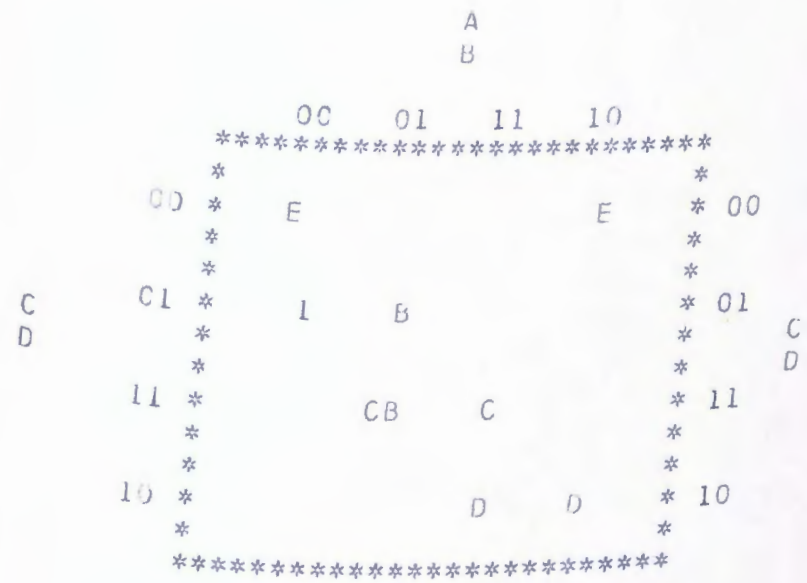
GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

REMOVE A

WE WISH TO REMOVE THE GROUPING REPRESENTED BY THE CHARACTER A

THE FUNCTION SO FAR IS

B'C'D' + A'BD + BCD + ACD'



GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

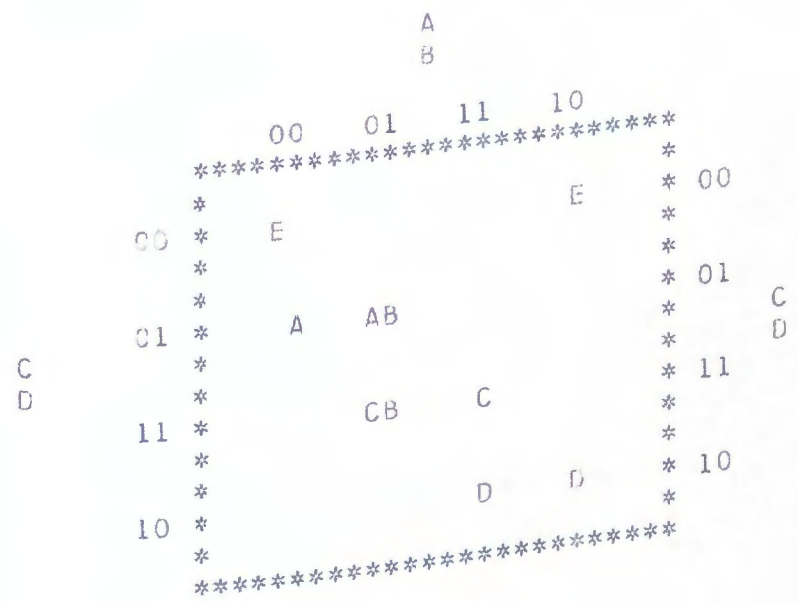
GROUP A'C'D

WE HAVE A GROUPING

THE MINTERMS INCLUDED IN THIS CALL OF GROUP ARE...
1, 5,

THE CHARACTER A ON THE MAP REPRESENTS THE TERM A'C'D
THE FUNCTION SO FAR IS

B'C'D' + A'BD + BCD + ACD' + A'C'D



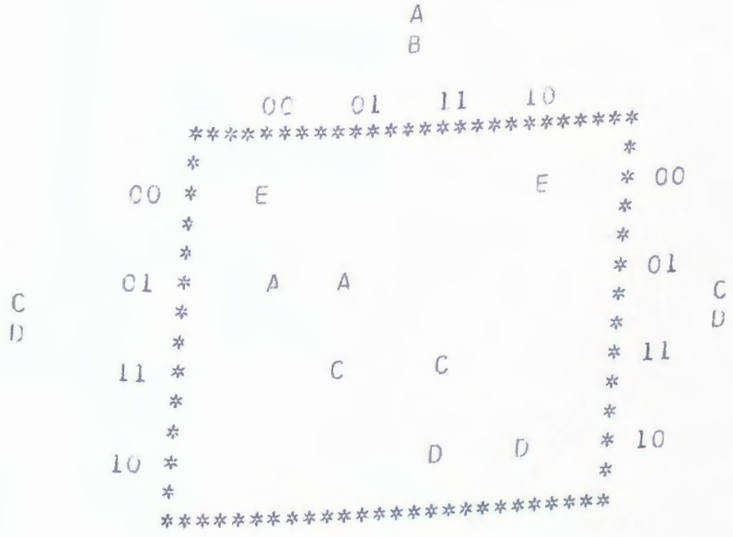
GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE CARDS ARE SCANNED.

REMOVE B

WE WISH TO REMOVE THE GROUPING REPRESENTED BY THE CHARACTER B

THE FUNCTION SO FAR IS

B'C'D' + A'C'D + BCD + ACD'



GROUPING HAS B[illegible]

CONFR[illegible]

WE WISH TO CON[illegible]

YOU HAVE GIVEN [illegible]

GROUPING HAS BEEN CALLED WITHOUT A PARAMETER LIST. THE DEFAULT IS ASSUMED.

EXAMPLE FOUR=T12345'W12X34+T12345YZ

WE DO NOT HAVE A GROUPING.

****

EXAMPLE FOUR=T12345'W12X34+T12345YZ

THE VARIABLE NAMES ARE T12345, W12, X34, Y, Z

THE INPUT FUNCTION IS ...

EXAMPLE FOUR = T12345'W12X34 + T12345YZ

THE EXPANDED FUNCTION IS...

EXAMPLE FOUR = T12345W12X34YZ + T12345'W12X34YZ + T12345W12'X34YZ + T12345W12X34'YZ + T12345W12'X34'YZ + T12345'W12X34Y'Z
+ T12345'W12X34YZ' + T12345'W12X34Y'Z'

T12345

YZ \ WX (T = 0)	00	01	11	10	YZ \ WX (T = 1)	00	01	11	10
00	0	4	12	8	00	16	20	28	24
01	1	5	13	9	01	17	21	29	25
11	3	7	15	11	11	19	23	31	27
10	2	6	14	10	10	18	22	30	26

T12345

YZ \ WX (T = 0)	00	01	11	10	YZ \ WX (T = 1)	00	01	11	10
00			1		00				
01			1		01				
11			1		11	1	1	1	1
10			1		10				

SAMPLE PROGRAM
****

```
$IBSYS
$*  TAPE 406  01  A 5  RING OUT
$PAUSE
$EXECUTE        MAMUS
$ID    SHUB*305/03/101*2M*200P*16    SIMLAT PROGRAM                    47/002
$COMPILE MAD,EXECUTE,PRINT OBJECT
          NORMAL MODE IS INTEGER
          PROGRAM COMMON DUMMY(13600)
START     CONTINUE
          SIMLAT.
          CHECK.
          PRIMES.
          ESSENS.
          COVERS.
          TRANSFER TO START
          END OF PROGRAM
$BINARY(11)
G1        A0,B0,C,D
G2        A0,B0,C0,D
G3        A,B,C0,D0
G4        A,B,C0,D
G5        A,B,C,D
G6        A,B,C,D0
 NOT A0   A
 NOT B0   B
 NOT C0   C
```

```
NOT  D0    D
OR   F2    G1,G2,G3,G4,G5,G6
           END OF CIRCUIT
F2(A,B,C,D)=3,1,12,13,14,15
F2(A,B,C,D)=0,3,7,13,14,15
           END OF CIRCUIT
NOT  A0    A
NOT  B0    B
NOT  C0    C
I1         A0,B0,C0
I2         A0,B0,C
I3         A,B0,C0
I4         A,B0,C
INPUTS       B=C
OR   F1    I1,I2,I3,I4
OUTPUTS    I1,I2,I3,I4,F1
           E'T
           F1=0,1,4,5
```

```
==========
G1
          AO,BO,C,D
==========
G2
          AO,BO,CO,D
==========
G3
          A,B,CO,DO
==========
G4
          A,B,CO,D
==========
G5
          A,B,C,D
==========
G6
          A,B,C,DO
==========
NOT AO    A
==========
NOT BO    B
==========
NOT CO    C
==========
NOT DO    D
==========
OR  F2
          G1,G2,G3,G4,G5,G6
==========
          END OF CIRCUIT
```

-THIS PAGE IS FOR REFERENCE TO THIS CIRCUIT-

THE NORMAL GATE TYPE IS ASSUMED TO BE 'AND'

THE IDENTIFIERS ARE ...

G1 G2 G3 G4 G5 G6 A0 B0 C0 D0 F2

THE FOLLOWING INPUTS ARE UNSPECIFIED...

C D A B

SIMULATION CAUSED BY A CONTROL CARD.

THIS IS THE OUTPUT FROM SIMLAT

C	D	A	B	G1	G2	G3	G4	G5	G6	A0	B0	C0	D0	F2
0	0	0	0	0	0	0	0	C	0	1	1	1	1	0
0	0	0	1	0	0	0	0	C	0	1	0	1	1	0
0	0	1	0	0	0	0	0	C	0	0	1	1	1	0
0	0	1	1	0	0	1	0	0	0	0	0	1	1	1
0	1	0	0	0	1	0	0	0	0	1	1	1	0	1
0	1	0	1	0	0	0	0	C	0	1	0	1	0	0
0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
0	1	1	1	0	0	0	1	0	0	0	0	1	0	1
1	0	0	0	0	0	0	0	C	0	1	1	0	1	0
1	0	0	1	0	0	0	0	0	0	1	0	0	1	0
1	0	1	0	0	0	0	0	0	0	0	1	0	1	0
1	0	1	1	0	0	0	0	0	1	0	0	0	1	1
1	1	0	0	1	0	0	0	0	0	1	1	0	0	1
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	1	0	0	0
1	1	1	1	0	0	0	0	1	0	0	0	0	0	1

ALL COMBINATIONS TESTED. SIMLAT. IS COMPLETE.

****

F2(A,B,C,D)=3,1,12,13,14,15

THE VARIABLE NAMES ARE A, B, C, D

THE INPUT FUNCTION IS ...

F2(A,B,C,D) = A'B'CD + A'B'C'D + ABC'D' + ABC'D + ABCD' + ABCD

NO ERRORS HAVE BEEN FOUND.

****

F2(A,B,C,D)=0,3,7,13,14,15

THE VARIABLE NAMES ARE A, B, C, D

THE INPUT FUNCTION IS ...

F2(A,B,C,D) = A'B'C'D' + A'B'CD + A'BCD + ABC'D + ABCD' + ABCD

THE FOLLOWING ERRORS HAVE BEEN DETECTED (NUMBERS ARE DECIMAL)...

0 EQUATION VALUE IS 1
1 EQUATION VALUE IS 0
7 EQUATION VALUE IS 1
12 EQUATION VALUE IS 0
END OF CIRCUIT

THE PRIME IMPLICANTS ARE...

A'CD
BCD
ABD
ABC
A'B'C'D'

THE ESSENTIAL PRIME IMPLICANTS ARE...

A'CD
ABD
ABC
A'B'C'D'

THE COVERS ARE...

F2(A,B,C,D) = A'CD + ABD + ABC + A'B'C'D'

```
NOT A0      A

NOT B0      B

NOT C0      C

I1          A0,B0,C0

I2          A0,B0,C

I3          A,B0,C0

I4          A,B0,C

INPUTS      E=0

OR  F1      I1,I2,I3,I4

OUTPUTS     I1,I2,I3,I4,F1

            E'T
```

-THIS PAGE IS FOR REFERENCE TO THIS CIRCUIT-

THE NORMAL GATE TYPE IS ASSUMED TO BE 'AND'

THE IDENTIFIERS ARE ...
AC BD CO I1 I2 I3 I4 F1

THE FOLLOWING INPUTS = 0 ...
B

THE FOLLOWING INPUTS ARE UNSPECIFIED...
A C

SIMULATION CAUSED BY A CONTROL CARD.

THIS IS THE OUTPUT FROM SIMLAT

A	C	I1	I2	I3	I4	F1
0	0	1	0	0	0	1
0	1	0	1	0	0	1
1	0	0	0	1	0	1
1	1	0	0	0	1	1

ALL COMBINATIONS TESTED. SIMLAT. IS COMPLETE.

****

F1=0,1,4,5
DUMMY NAMES USED.

THE VARIABLE NAMES ARE A, B, C

THE INPUT FUNCTION IS ...

F1 = A'B'C' + A'B'C + AB'C' + AB'C

NO ERRORS HAVE BEEN FOUND.

THE PRIME IMPLICANTS ARE...

B'

THE ESSENTIAL PRIME IMPLICANTS ARE...

B'

THE COVERS ARE...

F1 = B'

SOURCES CONSULTED

(1) B. Arden, An Introduction To Digital Computing, Reading, Massachusetts, Addison Wesley, 1963.

(2) B. Arden, B. Galler, and R. Graham, The Michigan Algorithm Decoder, Michigan University, 1966.

(3) A. Beam, The MAMOS Manual, Technical Report TR-65-15, NSG-398, University of Maryland and NASA, April 1965.

(4) S. Caldwell, Switching Circuits and Logical Design, New York, John Wiley and Sons, 1962.

(5) M. Harrison, Introduction to Switching and Automata Theory, New York, McGraw Hill, 1965.

(6) IBM Corporation, 7094 Principles of Operation, File No. 7094-01, Form A22-6703-1, August 1963.

(7) M. Karnaugh, "The Map Method for Synthesis of Combinatorial Logic Circuits", Transactions AIEE, Part 1, Communications and Electronics Volume 72, pages 593-599, November 1953.

(8) C. Keyserling, A Set of Computer Subroutines to Minimize Boolean Equations, University of Maryland, College Park, Maryland, 1968.

(9) M. Krieger, Basic Switching Circuit Theory, New York, The MacMillan Co., 1967.

(10) E. McCluskey, Introduction to the Theory of Switching Circuits, New York, McGraw Hill, 1965.

(11) A. Marcovitz and J. Pugsley, An Introduction to Switching System Design, University of Maryland, College Park, Maryland 1967.

(12) A. Marcovitz and J. Pugsley, "A Set of Computer Routines for Teaching Logic Design," American Society for Engineering Education, Annual Meeting, Washington State University, Event No. 67.1, June 1966.

(13) A. Marcovitz and E. Schweppe, An Introduction to Algorithmic Methods Using the MAD Language, New York, The MacMillan Co., 1966.

(14) M. Marcus, Switching Circuits for Engineers, Englewood Cliffs, New Jersey, Prentice Hall, 1967.

(15) R. Miller, Switching Theory Volume 1: Combinatorial Circuits, New York, John Wiley and Sons, 1965.

(16) M. Phister, Logical Design of Digital Computers, New York, John Wiley and Sons, 1961.

(17) J. Piersall, Computer Routines For Logic Design, University of Maryland, College Park, Maryland 1967.

(18) R. Prather, Introduction to Switching Theory: A Mathematical Approach Boston, Massachusetts, Allyn and Bacon, 1967.

(19) H. Torng, Introduction to the Logical Design of Switching Systems, Reading, Massachusetts, Addison Wesley, 1964.