TM-2621/003/00 (DRAFT)

TRACE-- Model II User's Guide

Timeshared Routines for Analysis, Classification and Evaluation

9 October 1967

This document has been approved for public release and sale; its distribution is unlimited.

190

DRAFT

TM-2621/003/00

TECHNICAL MEMORANDUM

(TM Series)

TRACE -- Model II User's Guide

Timeshared Routines for Analysis, Classification and Evaluation

by

Richard P. Esada

9 October 1967

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA
90406

The work reported herein was supported by SDC and contract DAHC15-67-C-0277, Bargaining and Negotiation Behavior, for the Advanced Research Projects Agency.

(page 2 blank)

ABSTRACT

This document presents a user's description of the TRACE system, which provides an on-line technique for scanning data and deriving variables. It is divided into two main sections: the first a tutorial guide introducing the user to the basic principles of the system, and the second a reference guide to the entire body of the TRACE program. The user is shown how to initiate an interaction with the time-sharing system, how to employ every capability of TRACE, what errors may be expected in operation, and what statistical products may be derived through use of the program. A complete index allows the user to refer readily to any portion of the document.

(page 4 blank)

ACKNOWLEDGMENT

The author wishes to thank the developers of the TRACE II system for their technical advice and cooperation in reviewing this document:

Robert J. Meeker
William H. Moore, Jr.
Gerald H. Shure

In working closely with this writer, they shared in creating and solving the sample problems that form the framework of the tutorial section, as well as contributing to the reference section.

FOREWORD

TRACE II is a system of computer programs designed as an instrument for prestatistical manipulations of large, complex collections of data. It is a system for those users who have a stockpile of data whose characteristics and perhaps even content is unknown. It is a tool for classifying, grouping and summarizing data, and for exploring relationships that may exist in the data from different points of view. TRACE, being a system that operates interactively with a user under time-sharing, permits an exciting interplay between the user's conjectural and judgmental skills and the computer's capacity for rapid and accurate data processing.

Unlike the typical data management system which is concerned basically with data retrieval, TRACE directs itself primarily to the construction of new data bases from old, to the derivation of new variables from existing ones. It is further distinguished from rival systems in its adaptability to widely differing data base structures. There are no rigid formatting requirements on the TRACE user; he may manipulate his data freely without having to write specific programs for this effect. It is TRACE's responsibility to construct and maintain the data base.

TRACE is written in the time-sharing system version of the JOVIAL language (JTS) for the AN/FSQ-32 computer at System Development Corporation. In a time-sharing system, a number of users share a large computer and divide the available time among themselves. Because of the computer's speed, all programs operate virtually simultaneously. The users interact with their programs from teletype or display console terminals that can be remotely located from the computer; for example, a user in Boston may participate in sharing SDC's computer in Santa Monica. The user is said to be "on-line" in the sense that his data and programs

are directly available to him from storage in the computer memory or associated random access files. Unless the computer is overloaded, the user receives an almost immediate response to any request he initiates at his terminal.

TRACE has a prodigious repertoire of services to offer the user. It can perform many statistical operations, such as Chi-square, Mann-Whitney, standard deviation; it can select variables, assign intervals, select subsets of data, configure complex indices, evaluate indices, and control iteration of these procedures. Many of these operations ordinarily present sizable practical problems to the user and are outside of, or only awkwardly or inefficiently satisfied by, standard programs. Besides these features, TRACE can perform desk calculations, list input and derived variable values, allow the user to query the data base and retrieve data, include all types of conditions in any derivation, output information in punched card format, construct tables of correlations between derived variables, and perform many kinds of editing services. A list of the statistical operations possible is presented in Appendix 9.

The following publications may be useful as references to the present document:

Shure, G. H., Meeker, R. J., and Moore, W. H., Jr. TRACE -- Timeshared Routines for Analysis, Classification and Evaluation. Paper presented at AFIPS Spring Joint Computer Conference, Atlantic City, N. J., April 18-20, 1967. Published in AFIPS Conference Proceedings, Vol. 30, 1967, pp. 525-530.

Meeker, R. J. User Interaction with the TRACE System. SDC document TM-2621/002/00. 18 pp. Paper presented at the American Psychological Association meetings, New York City, N. Y., September 3, 1966.

(page 8 blank)

Shure, G. H. TRACE -- Timeshared Routines for Analysis, Classification and Evaluation. SDC document TM-2621/001/00, 12 October 1966. 8 pp. Paper presented at the American Psychological Association meetings, New York City, N. Y., September 3, 1966.

Moore, W. H., Jr., Meeker, R. J., Shure, G. H. TRACE -- Model I: Timeshared Routines for Analysis, Classification and Evaluation. SDC document TM-2621. 3 September 1965. 57 pp.

PREFACE

This is a user's manual for the TRACE system written not only for the user but also literally by the user. It attempts to repeat for the reader the very same flight of knowledge that brought an understanding of the system to this writer. It is not intended as primarily a reference manual for the sophisticated systems user and it is not a primer for the absolutely innocent--there is no possibility that an instructional manual can be written to please both at once, not to mention the greater number of users who fall between these two extremes. Roughly put, it is a kind of compromise, trying to satisfy in some way every possible type of user but, in doing that, necessarily failing to satisfy fully every user.

In front of all other considerations, however, is the inescapable fact that it must teach the system to the most naive of users; if it thereby seems at times to linger on points obvious to the experienced user, it is only because of that one fact.

The reader will find, as he continues into this manual, that it is of two separate moods, almost two personalities: one is particularly sympathetic to the user with only little background in the subject; the other, dominant in later portions of this document, assumes that the reader has become familiar with the system, hopefully through study of the earlier part of this document. The first section of this document tries to lead the beginning user cautiously toward the heart of TRACE, almost hand-in-hand. The later section exposes every facet of the system in as terse an approach as possible--it is actually the reference section.

The first section is presented in the form of a dialogue between a hypothetical teacher of the system and a new user. Although not quite approaching an actual conversation, the dialogue does serve to voice two points of view: the teacher carries the line of instruction; the user plays the role of the questioner, the voice that prevents the teacher from lapsing into an endless monologue, the conscience of the manual dividing the flow of instruction into easily assimilable waves. It is this section that tangles with the most subtle intricacies of TRACE.

The second section--the reference section, attacks every operation of the system, each message in the communication between the user and the system. It is meant to be treated as a partner to the first section and to this effect there are reference pointers that link the two parts. You will notice that at the foot of every page in each section is a reference number. In general, this number refers to the particular section and subsection of a page. The first digit in the number is the section number, the second the subsection; for example, every page in part 3 of the second section will be labeled 2.3. This number also serves to link a discourse in the first section to a discussion on the same topic in the second. For example, the concept of "bracketing" is outlined in the second section within a half page; in the first section virtually an

PREFACE (cont'd)

entire subsection revolves around it. Therefore, the reader will find in the reference section a reminder that the subject is amplified in the dialogue section, for example: (See 1.4)

The appendices to this document virtually form a third section themselves. In these pages the user will discover exactly how to put himself in a position to use the TRACE program: how to make up his card deck, how to become a time-sharing user, how to bring the TRACE program into the time-sharing system, how to save his data for another run, and what troubles may be encountered during a run. The part on error messages describes some of the inner workings of the TRACE system that may not be evident from a reading of the two main sections.

A brief note on metalanguage: whenever a system message carries within it a name created by the user, for example, a name applied to a variable in the data base, this document will present that name in lower case to differentiate it from the rest of the message, which consists of system words printed all in upper case. Examples:

FOR ID = value DS-VRB variable:name HAS 32 VALUES
ROW IS row:variable:name CODED W

Most teletype machines employ the symbol Ø as the symbol for zero in order to distinguish that number from the letter O (upper case). In this document, the symbol Ø will appear only in contexts where such a misunderstanding could readily take place, for example, in the statement:

AM = M + N + O OTR Ø

This symbol will also be used in an illustration of an actual teletype display.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

(page 14 blank)

SECTION 1

A DIALOGUE ON TRACE

SECTION 1

INTRODUCTION

This first section is the retracing of the path taken by this writer in his efforts to understand the TRACE system. It should show the reader exactly how a prospective user with absolutely no previous knowledge of the system arrived at a point where he could justifiably call himself a TRACE user.

The approach is through the medium of problems, starting with a relatively simple problem and building towards more and more complicated problems. In trying to recapture the actual path into understanding, the writer has not, however, pointed out every little stepping stone. Sometimes a step is omitted; sometimes one is not discussed as fully as seems necessary. Whenever the reader wishes a more detailed explanation of any operation, he may satisfy himself by jumping to the appropriate discussion in the reference section or in the appendices. The reader is actually referred at times to specific points in these sections.

The rhythm of the dialogue is not mechanical. Its purpose is not to call attention to itself--only to be an accompaniment to the instruction. At times it may involve the user and the teacher in an interchange of ideas, at other times it may almost completely disappear leaving the teacher in a dialogue with the system, still other times it may reveal the teacher in a virtual monologue. Italics mark the voice of the user, elite type that of the teacher.

A word in defense of TRACE: The reader should not be led into the belief that all interactions with the TRACE system will necessarily be as involved as those found in this section. All the problems investigated in this section come from a common arrangement of data, an arrangement that has not been constructed to suit even a single one of the problems, let alone the TRACE system itself. The approach to understanding is here more than a mere approach; it is an actual attack on TRACE--to demonstrate some of its versatility in handling data analysis problems.

PART 1

The Problem Environment

First of all there is the question of finding a suitable environment for our problems. Which of the many environments that TRACE can accommodate appears most suited to the task of introducing that system?

We should look to one whose terminology is as universally familiar as possible and that has a resilience affording a succession of interesting and valuable problems, each somewhat more probing than the previous. Several environments suggest themselves: bookkeeping systems featuring payroll and inventory problems, analyses of psychological and sociological surveys, schedule-making for professional sports leagues, game theory situations. None of these, however, seems able to support at once our needs for depth, flexibility, and nearly universal understandability--even though all would make perfectly good subjects for work on the TRACE system. One that does appear quite adaptable to our needs is that of an educational system: its terms are completely familiar, it offers complexities enough for a large diversity of problems; in all respects it seems wholly acceptable for our purposes.

Would you describe the actual environment that we shall be using in our forthcoming exploration of the TRACE system?

It is a typical school system, perhaps one in a city of a half million people. As a whole, the system is divided into districts; these districts each hold a certain number of schools; each school has its individual complement of administrators, teachers and students. In fact, these components of the system that I have just mentioned belong to the information that will appear on the data cards we shall use in our problems.

Precisely what is the makeup of our input data?

The following elements of information are included in our data base (the entire collection of data): the various districts in the system, the schools within the districts, the principal of each school, the teachers in each school, the students in each school, the grade level of each student, an attendance and personal conduct record for each student, the subjects taken by each student together with the grade received and the teacher giving that grade, and finally the period of time that marks off each grade given. The districts are identified by means of numerical codes, for example, 1, 2, 3, 4 .... This coding device

is also used for the schools, the teachers, the principals, the students, and even the subjects. Each of these elements is assigned a distinct code within its class: for a system having 20,000 students, there would be 20,000 unique codes. The grades are given in alphabetical form, that is, A, B+, C-, and so forth. The attendance and conduct records are also of a normal form.

How is all this information put together on data cards? Is there a special format that must be followed for use in the TRACE system or can this system handle any arrangement of data conceivable?

TRACE can handle virtually any arrangement of data as long as the individual items are in some logical relationship with each other. A commonly followed arrangement and perhaps the most reasonable is that of letting each data card stand for a single value of some one variable, for example, students. Therefore, if you have a thousand values for students, you will also have a thousand cards in your input deck. The other information would then have to fit these cards or else be stored on another set of cards. For example, if you wished to include a value for district on each card, these values would have to correspond to the values of students on those cards: student 23 and student 23's district on one card, student 24 and his district on another, and so forth.

If a value were omitted from some cards, the arrangement of the information on all the cards should be such that any omission would be readily discerned, in other words, included logically if not physically. The answer again to your question about the input formats that can be handled by TRACE is that it can accommodate any system of inputting information that has a logical order.

How shall we arrange the information for our particular problem environment? Furthermore, how many general ways are there of arranging it?

Our data cards will have been set up so that, for a given time period, each card will reflect all the information available for a single student and only for that student. This allows us to use fewer actual cards than would be allowed by any other arrangement without getting into any tangled or crowded scheme for the data. Other arrangements would include setting up the cards on the basis of any of the other variables, for example, school, grade level, district, grade. However, basing our arrangement on values for school, for example, would mean an exorbitant amount of information directed to one card or the extensions of that card. (TRACE allows any number of cards to be treated as one logical card.) It would be far too cumbersome to allot to one logical card all the available information for all the students belonging to the one school represented by that card. This same objection also holds true for the variables of grade level and district. On the other hand, it

would not be an unwelcome arrangement were our cards to be based on subjects. In this case, the cards would have to be grouped together so that all the cards for the same student would be contiguous in the card deck. In other words, we would have here a set of logical cards, each consisting of perhaps six physical cards, for example, one card for each subject with a total of six subjects per student. The number of logical cards would be the total number of students in the school system. Thus, this method is quite similar to our own, the only difference being that our cards contain all the subject information for a particular student on the same card.

How does the element of time enter into the setting up of our data deck?

Each of the cards that I have described can contain only information about a student for a single time period, for example, a marking period, a semester, a school year. This is so mostly for reason of space: more than this information could not be squeezed into a single card and for our problem environment I thought it best that an economy both of cards and space within a card be practised, if only to make my demonstrations more nearly like a real life situation. In our data deck we will find that there exists a card for each month of the school year--a total of ten altogether--for each student. An interesting feature of TRACE is that these cards need not be in any order, although a strictly kept arrangement may be helpful for some operations. Since I do plan to use the advantages of ordered data, all ten cards for each student will be contiguous and in ascending order.

How will the information arranged in this way actually appear on the punched cards?

Every card in our data base will be punched as follows: first will appear the district in column 1, allowing the user up to ten districts (from 0 to 9); next the school in columns 2 and 3, allowing a total of 100 schools (0 to 99); then the principal in columns 4 and 5; the student's grade level in columns 6 and 7; the student's identification number itself in columns 8 through 12, allowing up to 100,00 students (0 to 99,999); next a series of academic records based on the student's subjects and grades. Each series will consist of the following: the first two columns to the subject; the second two to the grade; the next four to the teacher of the subject, allowing up to 10,000 teachers (from 0 to 99,000). We can arbitrarily set a limit to the number of records that can appear in a series: six. If for some reason a student has actually taken more than this number of subjects for a certain month, the records beyond the sixth will be lost. If the case is that a student has taken fewer than six subjects for any month, only records for the subjects taken will appear--the columns not filled will remain blanks. After these

columns will appear the student's attendance record in two columns, the conduct record in two columns, and finally the identification number for the month, requiring one column. (September = 0, October = 1, etc.)

Can you reproduce a fairly typical example of one of these data cards?

Here is a card for student 123 from district 4 for the month of April.

```
408090700123O3A-002707B+009913B-004510B+011114C+002219A 006718B+7
```

His card reveals that besides being from district 4, he goes to school 8, has a principal with the identification number of 09, is in the seventh grade, received a grade of A- in the subject identified by the code 03 and taught by teacher 27, received grades of B+, B-, B+, C+, and A in his other subjects, which are also described, attended classes for 18 days in the month, and won a personal conduct rating of B+.

Since there is no fixed scheme for entering information into data cards, how does TRACE know what our variables are and where they are?

You simply tell TRACE what your variables are, where they are located, and their format by means of another set of cards that precedes the data cards in your input deck. This set of cards, called the "dictionary," contains the names of the variables, their starting locations, the total number of columns assigned, the number of skipped columns between different values for the same variable on the same card if this occurs, and other similar information. A full description of the format for the TRACE dictionary is given in Appendix 3. In this dictionary, each variable commands one description card. So, if we had a data deck of a thousand cards, each card containing six variables, we would have to introduce it with a dictionary of six cards. If the data deck consisted of four cards, each card holding 60 variables, we would need 60 cards for the dictionary.

Our data cards each have eleven variables. What are they to be named?

A variable in TRACE can be referred to by a string of characters up to eight in length. Other restrictions are detailed in Appendix 2. The variables in our own problem environment can be named as follows:

variable	variable name
district	DISTRICT
school	SCHOOL
principal	PRINCIP
student's grade level	LEVEL
student	STUDENT
subject	SUBJECT
grade (for the subject concerned)	GRADE
teacher	TEACHER
attendance in days	ATTEND
personal conduct record	CONDUCT
month of the school year	MONTH

All these variables are in integer form, with the exception of the variables GRADE and CONDUCT, which are alphanumeric.

Is our input deck now completed?

Except for the mandatory control cards, which occur at the beginning and end of our deck and at the interface between the dictionary and the data cards, our input deck has been established. These control cards are discussed in Appendix 3. Appendix 3 also contains a diagram and chart of our input deck, showing precisely how our input deck was set up.

We are now ready to begin our interaction with the time-sharing system and eventually with the TRACE program itself. The former interaction involves logging ourselves into the time-sharing system and the loading of the TRACE program into that system. These steps, described fully in Appendix 4, bring us eventually to our first communications with the TRACE system. Its keynote message tells us that our interaction with the time-sharing system's executive is finished and that we can begin to input our data and define our first problem.

PART 2

Initiating the TRACE Program

What is this keynote message?

The following lines are typed out by the system for the user at this point:

```
YOU ARE STARTING TRACE  0.1
NAME YOUR DATA BASE
 ( )NAME
*
```

Notice that the first line declares a fact and the second makes a request. A TRACE message may also at times contain a question which the system expects the user to answer. In none of these cases, however, does TRACE complete a line by punctuation mark, for example, a question mark. It does, however, use what might be called a punctuation mark in ending a message: the last line of every message contains a single character--the asterisk. The user must then type in his response--every message from TRACE demands a response of some kind--on the same line as that asterisk, continuing it on further lines if necessary. Except for certain types of responses that will be discussed later, the user's response ends with a carriage return. This carriage return actually carries the user's response to the TRACE program and is also a signal to it to send its next message.

What is the meaning of this first message?

Besides telling us that the TRACE program has been loaded and that we may begin our inputting of data, the message asks us to name the data base with which we wish to work. Here the system wishes to know whether the data base for the problem is one already on disc or whether it must be input. If the name given is recognized by the system as one already on disc, it allows us to begin any operation we wish; if not, the system leads us into the mechanism for inputting a data base. The data base may already be on disc, for example, if we had to leave our interaction briefly because of some error, and then tried to recover by reloading the TRACE program.

How is the response to this message to be typed in? The fourth line in it seems to indicate a format for this response.

All messages that emanate from the system contain within themselves the form which the user is expected to follow in typing his responses. In this case, the form is very simple and is contained in one line: the user types the name of the data base (a maximum of six characters) and presses the carriage return. The characters to be entered during a particular response are always indicated by enclosing parentheses. In this case the parentheses exist but do not enclose anything. This gives us the option of entering any legal string of characters into our response. Notice that the word NAME appears just to the right of the parentheses; this merely specifies the type of information that is to go into the response. If the fourth line of this message were the following:

( ) (NAME)

we would be expected to enter not only the appropriate data base name but also the word NAME immediately after that name--with at least one space separating the two. If the message contained two lines exhibiting forms for possible responses, as in:

(A)LPHA

(B)ETA (G)AMMA

we would then have a choice between the two forms: we could type either the letter A or the two letters B and G. Remember that the letters appearing outside the parentheses are merely for elaboration. If the message presented the two lines:

( ) ALPHA

( ) BETA ( ) GAMMA

we would be expected to type either a value for ALPHA or two values: one for BETA and one for GAMMA. These latter two values must be separated by at least one space--and this is actually the general rule for all spacing in TRACE responses: entities in responses must be separated by one or more spaces. There is no other means of separating entities, for example, by commas.

What is to be our actual response to this first message from TRACE?

Our response should consist of the data base name for our data. It can be simply:

* EDUC

Notice that I typed the name right after the asterisk. Now, if I press the carriage return key, I shall obtain the next message from the system. However, before I do that, we should investigate what would happen were our data base name an already established name, that is, our data base already on disc. In that case the system would not need to input any data and would therefore take a different course after our response to the current message. It would, instead of helping us to input our data, jump to a point where it literally introduces itself to the user. I shall assume a hypothetical case so that we can see just how the system goes about introducing itself.

Why should the system choose ever to stand up and talk about itself? Isn't TRACE simply a network of questions and requests?

In a way, TRACE is really that. But it is also structured quite rigidly into several distinct units, each with a separate and important function, each actually qualifying as a subprogram. It is the list of subprograms that is typed out for the user at this point and that serves to outline for him what can be expected from the TRACE program. Actually, the message we have received is a communication from the executive program of TRACE. From now on, all interaction between us and the system will be carried on through its various subprograms. The list of subprograms is presented in the following message:

```
BASIC OPTIONS 0.0
(I)NPUT 1
(DE)RIVE 2
(C)ONSTRUCT 3
(DI)SPLAY 4
(E)DIT 5
(O)UTPUT 6
(S)TATISTICS 7
*
```

Notice that there are seven lines containing parentheses; there are therefore seven possible responses to this message: each response being a call to a subprogram of TRACE. Each subprogram name has been chosen so that it helps to explain by itself the function of that program. For example, the DERIVE subprogram derives results, the CONSTRUCT subprogram constructs tables of results, the DISPLAY subprogram displays these tables. The subprogram currently required is called INPUT and obviously serves to input our data

base. These subprograms are called by typing in the letter enclosed in parentheses. Incidentally, no harm will be done if we type in the remaining letters of the program, for example, the whole word INPUT.

Note: the reference section of this document is set up so that each subprogram of TRACE has devoted to it an entire subsection.

PART 3

Inputting Our Data

If we now add a carriage return to our response, which is that of a data base name, do we automatically enter the input subprogram?

Yes. The system now sends us every message in the input subprogram in order. The first two messages are quite perfunctory, referring to our input device, and we shall therefore not devote any time to a discussion of them now. A full description of these two messages as well as the complete input routine interaction is presented in Section 2.1. The first message from this routine that deserves discussion at this stage is the third message:

```
MISSING DATA  1.11
SET ARITHMETIC VALUES TO OTHER THAN ZERO
 ( )VALUE
 (N)O CHANGE
*
```

Here the system wishes to know what it should do about missing arithmetic data, that is, the blank spaces on data cards where numeric data (data for integer or floating point variables) might be expected to appear but which doesn't because either no information exists for these spaces or because some keypunching error occurred. Normally, the input routine treats missing numeric data as zero data; this convention is assented to by typing in the second response, the letter N. However, if we wish to have these blank card spaces treated as a particular number, we may type in that number at this point. (For missing numerical data, only numerals can be used as substitutes for the normal zero.) For our purposes, a blank can very well be treated as zero information; I shall therefore type the letter N.

```
* N
```

The carriage return then produces the next message from the input routine:

```
SET ALPHANUMERIC TO VALUE OTHER THAN BLANK
 ( )VALUE
 (N)O CHANGE
*
```

This message concerns a request similar to the previous; it asks the user to consider the possibility of substituting some value for the missing alphanumeric values in his input--other than the system convention of using a Hollerith blank (octal 60) for this purpose. This option is useful when we wish to refer to these blank values for computational purposes since it is impossible to distinguish a meaningful blank on the teletype from one that is merely a separator; that is, the computer does not recognize blanks in a teletype message as anything other than separators. Therefore, a character other than a blank must be used, for example, a colon. It seems quite unlikely that a colon could ever legitimately appear in our input data. My response to this message could then simply be:

```
* :
```

A carriage return summons the next message from the input routine:

```
GIVE ESTIMATE OF NUMBER OF DATA BLOCKS
*
```

This message is a request for an estimate of the length of the data base that we are inputting. A new term is unveiled here--data blocks. If you remember I mentioned earlier that a logical data card could consist of more than one physical card. For example, if our school system were that of certain European countries, a student could very easily find himself taking as many as twelve subjects concurrently. In this type of case, since one of our student records occupies six columns, a total of 72 card columns would be needed to accommodate just these records. This is clearly too much for one card; it really demands two cards per student. It is still one logical card per student even though two physical cards are used. This logical card is called a "block." It is the set of cards that contains one, and only one, value for each variable in the dictionary (unless the variable is a so-called "string variable." This shall be discussed at further length later in this section). In our data base, each block is one card long. The length of a block in terms of physical cards is one of the items that is set into the dictionary preceding the data.

I must type in an estimate here, preferably a generous overestimate, since an underestimate would result in blocks not being processed. My response could look like the following:

```
* 100000
```

Note: commas cannot be used in typing in numbers, e.g.:

```
* 100,000
```

A little more interplay now takes place between ourselves and the system, and the data base has been input. The details of the messages and responses responsible for this are given in Section 2.1. The input routine closes its work with the following message:

```
(C)ONTINUE WITH INPUT ROUTINE
(B)ASIC OPTION
*
```

This message asks us to either select another subprogram (basic option) or else indicate that we wish to input another set of data. This latter course would be necessary were our data in two or more separate sections, for example, two tape files, a tape file plus a disc file, a disc file plus teletype input. The input subprogram can handle only one of these sections of data--called a data set--at a time. In other words, all the data that is entered during any one interaction with the input routine is called a data set. This term will be met at various times during our exploration of the TRACE system. If we wish to call another program, we must type the letter B followed by a space followed by the appropriate code letter of the program. For example, to call the construct table program, we would type:

```
* B C
```

To continue inputting data, we would merely type the letter C after the asterisk. Since, however, we are finished with our input, we shall leave this program and try one more operation that is related to inputting data but that is not actually part of the input subprogram's functions. This function lies in the framework of the so-called edit routine, which I shall call by typing:

```
* B E
```

The carriage return elicits the following initial message from that routine:

```
EDIT  5.0
 (D)ELETE
 (L)IST
 (C)HANGE
 (F)ORM
 (R)ENAME
 (A)SSIGN BY REPLACEMENT
 (B)ASIC OPTIONS
*
```

This message is a list of the various functions of the edit program, which can accomplish some very useful things, like eliminating entire data sets, or specific blocks of data; it can change the names of variables and even the values of variables. The user finds the particular function in which he is interested and types in the corresponding code to call it. These functions are all described in Section 2.5. For our case the function labeled (A)SSIGN BY REPLACEMENT has an immediate use. It is this function that is used to change the actual values of a variable--from the original form that is punched into the data cards into any form that the user specifies. For example, he could ask that all values of zero for the variable called MONTH be transformed to one's. Our data base offers a good candidate for this type of value changing: note that all the variables but two are expressed in numbers only--the two exceptions are the variables GRADE and CONDUCT. As alphanumeric variables, these are both expressed in letters and symbols, for example, B+. This form is obviously impervious to arithmetic calculation--how does one obtain the mean of alphabetical grades? Or how does the user add together a series of letters? The solution lies in converting these values to numerical values. This being the job of the ASSIGN function, I call that function by typing the letter A.

```
* A
```

The system returns the first message of that function:

```
ASSIGN VALUES  5.31
 ( )NAME OF VARIABLE
*
```

The user is now expected to supply the name of the variable whose values he wishes changed. I then type in the appropriate name:

```
* GRADE
```

The next message from the function is:

```
 ( )OLD VALUE = ( )NEW VALUE
END WITH //
*
```

This message asks us to type in a series of values in the form shown, that is, the original value followed by an equals sign followed by the new value. A new series can then follow, in this case, on a second line. Each series needs a line for itself and the last line ends the response. And here we meet a new device.

So far we have been accustomed to ending a response with a carriage return, but this carriage return also is what delivers the response to the system. What happens when the entire response cannot be accommodated by one line? TRACE acknowledges the cases where this can happen and allows the user to use another way of ending his responses--a double slash. This is then followed by the usual carriage return for entering the response into the system. This device can only be used when TRACE permits it. Most often, the TRACE message itself will carry a line indicating the need for such an ending; for example, in our latest message it was:

```
END WITH //
```

The user must then end his response with a double slash, whether the response contains a few characters or whether it extends over several teletype lines. For the response to our current message, I shall type the following:

```
A = 9   (cr)
B = 8   (cr)
C = 7   (cr)
D = 6   (cr)
F = 5   (cr)
: = 5   (cr)
+ = 8   (cr)
- = 2 // (cr)
```

(The symbol (cr) stands for: carriage return.)

After each of these carriage returns not preceded by a //, TRACE issues the message:

```
CONTINUE
```

which is nothing more than a reminder to the user that more information is expected during this particular response. Following this message, TRACE causes another carriage return.

This operation converts all the original alphanumeric values into new values; for example, a value of A- becomes 92, C+ becomes 78, B (actually B: since we changed blanks to colons) becomes 85, and so forth. Notice that I could not have referred directly to the input blank following the grades A, B, C, D, and F in this operation--TRACE cannot recognize blanks as entities in a response; it sees them only as separators.

After the final line in our response (ended with both a double slash and a carriage return), TRACE issues the following message:

```
COMPLETED
EDIT  5.0
 (D)ELETE
 (L)IST
.
.
.
.
 (B)ASIC OPTIONS
*
```

We are now given the opportunity of either continuing with another function of the edit routine, or else going to another subprogram. In either case, I shall employ a new device in framing our responses. This one concerns the common modes of travel through the network of TRACE messages and subprograms.

What are these modes of travel through the TRACE system? Thus far we have only moved from one routine to another by typing the code letter of the subprogram desired.

You may have noticed that our previous jumps from one subprogram to another were made by typing in a letter code suggested by the system itself. For example, if we had started this interaction with the TRACE program with an old data base, we would have left the TRACE executive by typing the letter I immediately after the executive had presented us with a list of codes that could be used to call the various programs. We left the input routine after we reached the last function in it and were again presented with the opportunity to call another program. This time we were asked to either call another program by means of a code (e.g., B E) or else return to the beginning of the input program. Within the input program we traveled from one function to another

automatically; within the edit program we were allowed to select a function by typing in an appropriate code. There exists, nevertheless, still a more general way of moving from one subprogram to another or, within a subprogram, of moving from one function to another. This is done by typing in the command GOTO followed either by a letter code or by a reference number. If you would look back to the various messages received from the system so far, you will notice that every first line (sometimes second) of a message introducing a subprogram has a number to the right--the same is true for functions within the subprograms. For example, the first message from the input routine carries the number 1.0, the fourth (I skipped the second and third) the number 1.11, and the edit function just completed 5.31. Moreover, each of the subprograms listed by the executive is followed by its basic reference number, e.g., input is 1, display is 4. Each of these numbers can be used as a transfer point in TRACE, depending on the circumstances. If the transfer is to be from one subprogram to another, the user must type the subprogram's code (e.g., DE) after the GOTO command; if the transfer will take place entirely witnin one subprogram, he must use the reference number.

To illustrate: when I was about to leave the input subprograr for the edit program, I could have typed:

GOTO E instead of: B E

Once inside the edit routine, I could have typed:

GOTO 5.31 instead of: A

in order to call that particular funciton. It is not possible, however, to go from a particular point within one subprogram to a specific point inside another; travel between different subprograms can only be toward the beginning message of the subprogram being entered.

At this moment in our exploration of TRACE, we are ready to compute the solution to our first problem. We must therefore call upon the derivation subprogram. I can type:

GOTO DE

PART 4

The First Problem

We have completed inputting our data base and are now ready for our first problem.

The first problem is:

How many students are there in each school in the entire school system?

Our first move is to call the derivation subprogram, which will do the work of solving this problem. We have just called it by typing GOTO DE. We then receive the following initial message from the derivation routine:

```
DERIVATION  2.0
 (S)TART
 (L)IST AVAILABLE VARIABLES
*
```

This message asks the user whether he wishes to proceed immediately with the next step in the routine or whether he wishes first to have all the variables currently available to him listed by the system. He might choose this latter course if he had earlier done a lot of editing of his data, for example, deleting various blocks of data or changing variable names and values. Since this type of editing hasn't taken place and since our store of variables is only eleven, I am prepared to start the sequence of messages that constitutes the computational work of the subprogram. I type the following:

```
* S
```

The system returns the message that starts the chain of events leading to the eventual derivation of the problem's solution:

```
CASE INDEX  2.1
 ( )CHANGE TO
 (N)O CHANGE
*
```

This message is asking us to tell the subprogram at this earliest point in the derivation process the frame of reference, so to speak, for the results of that derivation. It happens that TRACE in building a derivation always sorts the results on the values of some variable--called the case index variable. The response to this current message will actually be a command to TRACE telling it how to sort the results of our derivation. As you will see later, both responses to this message convey a variable name to the system; that variable is then recognized as the sort variable for the derivation.

There are several ways of looking at this concept of case index; one we can examine right now. TRACE does all its deriving by means of the following operations: counting, summing, averaging, and finding a median; making Boolean algebraic tests; plus the usual arithmetic operations of adding, subtracting, multiplying, and dividing; plus such common operations as computing square roots, logarithms, and trigonometric functions. However, before it begins any derivation containing these operations, it must know just to which variable the results are to pertain. It may count the number of appearances of a certain variable in the data base, or sum the values of this variable, but it still wishes to know a second order of pertinence. That is, telling it to compute the sum of the values for some variable or some portion of that variable is not enough; you must tell it to compute that sum in terms of some other variable--per some other variable (of course, it could in some cases be the same variable). TRACE would then compute a result appropriate or pertinent to every value of the case index variable.

A pair of hypothetical problems to illustrate:

(1) What is the average grade per student in school 11 of district 2?

(2) How many teachers have given out at least 20 A's to their students in the final month of the school year?

The frame of reference in the first problem is "student" because the results of average grades pertain to students; the case index is therefore the variable STUDENT. When TRACE computes averages of the values for the variable GRADE, these averages will be applied to the occurrences of unique values for STUDENT (according to the conditions expressed in the problem). In other words, every qualifying value for STUDENT will have associated with it the appropriate average grade.

In setting up the derivation statement for the first working problem of ours later, you will notice that that statement does not contain any reference to the conditions (restrictions) attendant to the problem, for example, in the first hypothetical problem, the restriction that the students must be associated only with school 11 of district 2. These restrictions are relegated to a separate statement. TRACE then requires three statements for a derivation: first, a case index; second, a list of restrictions; finally, the computation statement. So, in this hypothetical problem, the user would first tell the

system that the case index is the variable STUDENT, then the restrictions about school 11 of district 2, and lastly that the computations consist of finding averages for the values of GRADE.

The matter of determining the case index for the second hypothetical problem is a somewhat different story. Here there is no obvious "per" variable. The derivation statement will ask for a counting of teachers and, in this case, will also hold the condition about the 20 A's given out. The restriction statement will stress the matter of the final month. However, where is there a visible case index? The answer is that visible or not, there must be a case index specified. For this problem, that index could be the variable DISTRICT, or SCHOOL, or even TEACHER. If the case index were given as DISTRICT then, TRACE would compute the results so that every unique value for DISTRICT would have associated with it the appropriate number of teachers who have given out 20 A's in June. If SCHOOL, the number of teachers per school. If TEACHER, the number of teachers per teacher, which is equivalent to associating each qualifying teacher with herself. As you will see later, there is also a way of obtaining a grand total of results, so that the specifying of a case index becomes entirely academic.

So, in our current working problem, the case index is clearly: SCHOOL. How many students <u>per</u> school in the system? Therefore, the restriction statement must have something to do with the entire system, i.e., the entire data base, and the derivation statement must concern itself with determining a count of students.

Correct. Notice that the current message from the system allows us two possible answers: either we submit the name of the variable we have selected as the case index, or else type N. This latter response would give us as our case index the variable whose name appears on the END CASE control card (see Appendix 3). This particular variable is chosen so that it will suit the majority of problems run on the data set. Following my entering of a case index variable, the subprogram will issue a message asking for a restriction statement:

```
*SCHOOL
RESTRICT DATA  2.11
 ( )RESTRICTIONS END WITH //
 (N)O
*
```

As you can see, we are asked to specify the restrictions on the current case index if there are restrictions, otherwise to indicate that there are none. The admonition to the user to end his restrictions with a double slash implies that the system is prepared to accept a lengthy series of restrictions. This is true; we can type in any number of restrictions at this point, closing the series with a //. The form of each series is as follows: variable name followed by an equals sign followed by the value(s) defining the restrictions. Series are separated from each other by a space and the first series is heralded by the word FOR. To illustrate, in the first of the two hypothetical examples I introduced a while ago to demonstrate the use of the case index, we would restrict the case index (in other words, the computation itself) to those students who went to school 11 in district 2. The correct message to the computer to achieve this would be:

```
FOR SCHOOL = 11 DISTRICT = 2 //
```

Of course, if district 2 were the only district that held a school with the value of 11, that second restriction need not have been entered. Similarly, if we had wanted to have the result computed for all schools in district 2, we should have omitted the first restriction.

Since our first real problem calls for the number of students in each school in the <u>entire</u> system, there is at first sight no apparent restriction on the derivation. However, in our particular data base, each student is represented by ten cards; therefore, each student could be counted ten times if there were no restriction. We can avoid this duplication of counting by introducing a restriction that limits the counting process to one card per student. This is most conveniently done by simply directing the derivation to a single month, for example, June. My restriction statement then becomes:

```
* MONTH = 9
```

The system next asks us for our computation statement:

```
GIVE DERIVATION STATEMENT  2.2
 ( )END WITH //
*
```

Again (because of the //) we are allowed to submit a series of lines; however, these lines must all be combined into one composite derivation statement by means of the connecting words AND and OR, which I shall discuss in detail later. At any rate, whether short and singular, or long and composite, the derivation statement ends with a double slash.

The simplest derivation statement will have a skeleton that looks like this:

(result type) = (computational expression)

where the result type is limited to one of three forms: AM, FM and IM; but where the right hand side can run on for several lines, containing arithmetic operators, relational tests, variables, all held together by the conjunctions AND and OR. The three forms legal for the left side are the following:

AM	alphanumeric result
FM	floating point result
IM	integer result

The left side of the derivation statement therefore tells the system how to store the derived results. The results collectively, from any one derivation statement, form a variable--called either a derived variable or a measure.

Using again my hypothetical case about the average number of students in school 11 of district 2, I would necessarily want the results expressed in floating point, since the computation involves addition and division. My derivation statement for that problem would then begin with FM. On the other hand, for the case about counting the number of teachers who had given out more than 20 A's last month, the results could be presented in integer form, since each result is merely a whole number. Of course, asking for floating point instead would certainly do no harm. For our problem, either IM or FM is suitable since we are merely counting a number of students.

Again looking to the first hypothetical problem: all that would remain a mystery to the system at this point would be the nature of the calculation; it already would know what the calculation will pertain to (the case index) and already would have received the restrictions. The necessary calculation would be merely the average of all the qualifying grades. The TRACE term for this operation is MEN, meaning the arithmetic average or mean. I would thus have typed the following derivation statement for the problem involving the average grade for students in school 11 of district 2:

```
FM = MEN GRADE //
```

This one statement would mean the end of the work of deriving the result. The remaining interaction with TRACE would concern itself with displaying the results.

But now back to our real working problem:

How many students are there in each school in the system?

Remember that I mentioned earlier the four statistical operations in TRACE: counting, summing, averaging and finding a median. You have already seen how one of these could be used in a derivation statement--the operator MEN for averaging. Our problem calls for a counting operation, the operator NBR. The remaining operators are SUM for summing, and MDN for obtaining a median. Our derivation statement then can easily resolve itself into the following:

```
IM = NBR STUDENT //
```

It, of course, could also have been presented as:

```
FM = NBR STUDENT //
```

What would happen if we had used the operator SUM instead of NBR? Doesn't it look as though this operator might give us the sum of the students?

The difference between the work of the operators SUM, MEN, and MDN and the effect of the operator NBR is that the former act upon the values of the variables referred to, whereas the latter acts only upon the occurrences of variables; that is, it counts the occurrences of discrete values for a particular variable. If we had used the operator SUM in our derivation statement instead of NBR, we would have gained the sum of the values, that is, the identification numbers of the students. In other words, 1 + 2 + 3 ... +n.

This statement ends the work of derivation. A carriage return causes TRACE to start computation, the completion of which is signaled by the following message:

```
IM001
IS PROGRAM NAME  2.21
ASSIGN
 ( )USER NAME
 (N)O
*
```

This message tells the user what the system will name the derived variable (measure). It builds this name out of two components: the code for the result type that the user typed into the left hand side of his derivation statement plus a number that reflects this variable's chronological position among all derived variables. For example, if a user had entered six derivation statements with result types in the following order: FM, AM, FM, FM, AM, and IM; the system names would be the following: FM001, AM002, FM003, FM004, AM005, and IM006.

The user is, however, given the option of rejecting and replacing the system name. If he chooses to accept the system name, he types the response N. If he chooses to assign a name of his own manufacture to the variable just derived, he types in that name. This name must obey the general rules for TRACE names. For the results to our current problem, we can then either accept the system-assigned name of IM001 or else construct a new name. Suppose I type in the name STD# to replace IM001:

```
* STD#
```

The derivation subprogram now submits its final message:

```
DERIVATION COMPLETE  2.22
DERIVATION  2.0
 (S)TART
 (L)IST AVAILABLE VARIABLES
*
```

The program is showing us that it has concluded its work for the current derivation and is ready to accept another derivation problem. It also makes itself available for the listing of the variables now outstanding, whether input or derived. We also have the right to exit from this routine by means of a GOTO command. Since we do not have another problem at the moment, nor is it necessary to have our variables listed, we should consider looking at the results just derived. One of the simplest ways of doing this is by way of the edit subprogram, the same program that permitted us to change the values of our variable called GRADE. One of its capabilities is that of listing the values of any variable, including derived variables. I can summon this routine by typing:

```
* GOTO E
```

With the reappearance of the edit subprogram, the same list of functions that was presented us at its first appearance is again shown us. If you recall, the second function was represented then as:

(L)IST

This is the function that serves to list the values of any specified variable. I can call it by typing:

```
* L
```

That function now introduces itself with the following message:

```
LIST VALUES  5.2
 ( )NAMES
 (D)ATA BASE INDICES
 (FOR) ( )NAME (=)  ( )VALUES (FIND)  ( )NAMES
END WITH (//)
*
```

The second line requests us to enter the name of the variable whose values we wish listed; the third and fourth lines happen to be of no consequence at the moment so we shall ignore them (they are discussed at length in the reference section); the last line tells us that our response must be ended with a double slash. Since my request is simply that of having the values for the derived variable STD# listed, I shall type the following:

```
* STD# //
```

The next message from the edit routine makes a statement of fact and then asks us to make a judgment on the basis of that fact:

```
TOTAL NUMBER OF BLOCKS = 98
GIVE BOTH  5.21
 ( )STARTING BLOCK  ( )NUMBER OF BLOCKS
*
```

Notice that this message first tells us how many blocks of data have been produced for our derived variable; it then asks us to state how many of those blocks we wish listed at this point.

The user must provide the system with both the number of the first block to be printed (assuming that he knows what that block number is) and the total number of blocks to be printed during this forthcoming listing. This type of freedom is especially valuable if, as an example, I did not need to have any blocks of results displayed but the last seven of a total of 200. I would then type:

```
* 194 7 //
```

For our current problem, I wish to have every block printed: I therefore type:

```
* 1 98 //
```

where n is the same number appearing in the first line of the current message from the subprogram.

A word about the meaning of the term "block" in this context. As you know, I have already used this term as a synonym for a logical card. In this context, however, it seems related not to a characteristic of our input but to our newly derived results. It is still, nevertheless, related to our input: there is one derived value for each block mentioned in this message, but there is also a case index value for every derived value since derivations occur only for occurrences of the case index variable outstanding and the case index must be an input variable. Therefore, "block" refers both to an occurrence of a case index value and a derived value.

The next message from the system is the actual listing of the values desired, together with the block numbers and the associated case index values. The listing is presented in the form of three columns: the first (leftmost) containing the block number and headed by the letters BLKN, the second the values of the variable in question and headed by the name of that variable, and the third displaying the values of the case index variable that correspond to both the block number and the derived variable value. The system orders the listing so that case index values appear consecutively, starting from the lowest value; the block numbers occur in succession, from 1 through n.

A listing for our current derivation will resemble the following:

```
BLKN    STD#    SCHOOL
1       436     1
2       306     2
3       554     3
4       379     5
5       826     11
6       474     12
:
93      390     145
GIVE BOTH  5.21
 ( )STARTING BLOCK  ( )NUMBER OF BLOCKS
*
```

Notice that the subprogram allows us now to return to the point where we may specify another portion of the same derived variable values. We cannot, however, there ask for a listing of the values for another variable. In our case, we have received a listing of all the values for our derived variable STD# and therefore have no further use for this particular mechanism of the listing function; nor do we have any more use for the edit program at all since our derviation is now complete and the results fully exhibited. We are actually now ready for another problem.

PART 5

The Second Problem

The second problem is:

> *How many students are there in each school according to the following categories: grade levels 1 through 6, 7 through 8, and 9 through 12?*

We can assume that there are several types of schools in the system: some that have all grade levels from 1 through 12; some that are structured on grades 1 through 6, some only on grades 7 and 8, and so forth.

At first sight, this problem seems quite similar to the first. The derivation statement will be the same, the case index will remain the variable SCHOOL--only the restrictions are apparently different. In this case there must be three restrictions resulting in the equivalent of three restriction statements:

```
FOR LEVEL = 1 - 6 //
FOR LEVEL = 7 - 8 //
FOR LEVEL = 9 - 12 //
```

From what I've learned from solving the first problem, this current problem seems to lead into three separate derivations: each beginning with one of the three restriction statements that you have just described, but continuing with identical case index statements and derivation statements. However, if that were true and the problem actually asked for the number of students per school for each possible grade level, we would have to present the system with twelve distinct derivations. And if our problem were one wherein the number of restrictions ran into larger numbers, it would mean our spending a great deal of time at the teletype typing in duplicate statements. Is it possible to shorten this process by our typing as the one and only restriction statement a composite of the singly appearing restriction statements that we would otherwise have to enter? In other words:

```
*FOR LEVEL = 1 - 6 LEVEL = 7 - 8 LEVEL = 9 - 12 //
```

No. The TRACE system tries to obey all the restrictions given it in a single statement simultaneously. For example, suppose that we had a data base wherein the school identification numbers were not unique within districts and you wished certain results for all schools numbered 13 and all schools in the

district numbered 4. You could not obtain these results by entering as your sole restriction statement:

```
*FOR SCHOOL = 13 DISTRICT = 4 //
```

The system would derive results only for those schools in district 4 that were labeled with the identification number 13--both restrictions in that one statement are obeyed simultaneously if possible. The system would under no circumstances first concern itself with all the schools labeled 13, deriving results for that group, and then consider the schools in district 4. The restriction statement that you have typed in then, in effect, restricts the derivation to the condition that all the individual restrictions be met at once--this is impossible; a student can be in only one grade level at a time. Therefore, your composite statement actually leads to a null derivation.

Isn't there some way the user can avoid typing in series of statements when the only difference between any two series is perhaps only a single statement or even single parameter?

Since this is a frequent situation met by the user in handling problems on a large data base, TRACE does have a shorthand method of dealing with it. This method is called "bracketing" and can be used either in restriction statements or in derivation statements. It allows us to leave one or more elements in either or both of these statements in a derivation unspecified; TRACE then asks us to supply it with the missing parameters. In fact, it does this repeatedly until we tell it to stop. In this way, a single series of statements (one case index statement, one restriction statement, one derivation statement) for a derivation serves to accomplish what otherwise might have required a multiple series of statements. We can use our current problem as an example to illustrate this method. First of all, I shall call the derivation routine:

```
GO TO DE
DERIVATION  2.0
 (S)TART
 (L)IST AVAILABLE VARIABLES
* S
```

```
CASE INDEX  2.1
 ( )CHANGE TO
 (N)O CHANGE
* SCHOOL
RESTRICT DATA  1.11
 ( )RESTRICTIONS END WITH //
 (N)O
*
```

And here is the beginning of our shorthand way of dealing with multiple restrictions:

```
*FOR LEVEL = [1] - [6]  MONTH = 9 //
GIVE DERIVATION STATEMENT  2.2
END WITH //
*IM = NBR STUDENT IF ATTEND NQ Ø //
GIVE 2 PARAMETERS
 ( )VALUES
 (S)TART
*
```

Notice that the bracketing occurs twice in the same statement: the restriction statement. As I have said, this tells TRACE that a <u>series</u> of values will be provided for the parameters in question. I have typed in the initial values for the two parameters, but actually any string of six or fewer characters would do; for example, I could have typed:

```
FOR LEVEL = [FIRST] - [SECOND]
```

or even:

```
FOR LEVEL = [A] - [B]
```

since any element in the statement may be parameterized. In this latter case, the system would ask me for three parameters.

Notice next that the system begins to ask for missing parameters immediately after the entry of the derivation statement. This is the line:

GIVE 2 PARAMETERS

followed by an asterisk that permits the user to type in the appropriate values or else return to the beginning of the routine. He may also, of course, type in a GOTO command in order to move to the end of the derivation program. The request for parameters concerns all bracketed parameters in the derivation thus far; since we have bracketed two, it is a request for two parameters. A cycle is thus instituted: the system requests parameters; the user supplies them; the system runs a derivation on the basis of these values; it again asks for a set of parameters; and so forth until the user stops the cycle by typing in either a GOTO command or the response S.

What is the meaning of the clause that you have added to the original derivation statement? Namely:

IF ATTEND NQ Ø

This is what might be called an IF clause, or a conditional clause. It tells the system to derive the result in question only if the condition expressed by the clause is met. In this case, I've asked the system to discount any students who haven't attended at least one day during the term. It is only a perfunctory caution in our example, but it does serve a logical purpose. It could very well be that the student roll was made up before the first day of classes and has not been updated to reflect the movement of students out of the system before that first day.

The term NQ apparently means: not equal to. What other terms are accepted by TRACE in conditional clauses?

TRACE recognizes the following: EQ as equal to, LS as less than, LQ as less than or equal to, GR as greater than, and GQ as greater than or equal to.

I would like to deviate a little now and present another derivation statement for analysis:

*IM = ATTEND IF DISTRICT EQ 1 OTR Ø //

Here you see a new element in derivation statements: the clause OTR Ø. This asks the system to set the result to zero if the condition set by the IF clause is not met. It means literally: otherwise the result is zero. I could also have set this alternative clause to some computational expression. This alternative is useful since TRACE does not provide any value at all for the derivation whenever the conditional clause (IF clause) for a particular case index value fails. With an OTR, the user can control values in case of this type of failure.

Why didn't you attach this alternative clause to our current derivation statement? It would then be:

```
*IM = NBR STUDENT IF ATTEND NQ Ø OTR Ø //
```

TRACE does not allow this freedom with statements beginning with the so-called statistical operators: NBR, SUM, MEN, and MDN. All other derivation statements, however, may carry this clause.

Is there any way in which a derivation statement can be amplified to include more than a single condition, in other words, comprising two or more IF clauses bound together by logical "ands" and "ors"?

The conjunctions AND and OR can be used to connect what are theoretically separate IF clauses in themselves into one IF clause, which may be drawn out to indefinite length. They cannot be used to join different variables within a clause, only the different conditions. For example, the following derivation statement is a legal one:

```
FM = SQR A + TAN B IF C GR D AND E LS F OTR SIN G //
```

This tells the system to compute the sum of the square root of A and the tangent of B if C is greater than D and E is less than F; if either of these conditions is not met, the result is, instead, the sin of G. The AND in this statement connects the phrases C GR D and E LS F to make one composite IF clause; it does not connect D and E.

The following is also legal:

```
FM = SQR A + TAN B IF C GR D OR E LS F OTR SIN G //
```

This tells the system to compute the result if either C is greater than D or E is less than F; if neither is met, then compute the alternative result. The following is legal:

```
FM = SUM A IF B GR C OR D GR E AND F GR G //
```

This states that B must be greater than C or else both must be greater than E and F greater than G. It does not mean that either B must be greater than C or D greater than E, and that furthermore F must be greater than G. When both AND's and OR's appear in the same statement, the OR acts to create new phrases within the IF clause, not the AND--the AND then only belongs to the OR phrase. For example, the current example is treated as though the following parenthesized phrase existed:

```
FM = SUM A IF B GR C OR  (D GR E AND F GR G) //
```

The following pair of statements can also illustrate this relationship:

```
FM = SUM A IF B GR C OR D GR E AND F GR G OR H LS I OR J GR K AND L GR M //
FM = SUM A IF B GR C OR (D GR E AND F GR G) OR (H LS I) OR (J GR K AND L GR M) /.
```

The following statement is not legal:

```
FM = SUM A IF B GR C OR G AND E GR F //
```

A conjunction cannot connect two variables after the same operator. In this case, the clause does not mean that B must be greater than either C or G in one half of the clause. The correct clause for this intention would be:

```
IF E GR F AND B GR C OR B GR G
```

It also would not be:

```
IF B GR C OR B GR G AND E GR F
```

since the condition beginning with the AND belongs to the OR phrase, and this is not the apparent intent of the first incorrect clause.

What other arithmetic operators besides + does TRACE recognize, and in what order if several occur in the same statement?

TRACE recognizes the arithmetic operators +, -, / and * for addition, subtraction, division, and multiplication, respectively. When they occur together in a formula, TRACE recognizes and executes them from left to right in two passes: first recognizing divisions and multiplications, then the additions and the subtractions. For example,

A + B * C * D - E / F

means: multiply B and C, then multiply this product by D and save the new product; then divide E by F; then add A to the saved product; finally subtract E / F from this sum.

You said earlier that we could have included a bracketed parameter in the derivation statement. Isn't it then possible that we might have inserted our restrictions into the derivation statement as a conditional clause? That is, our restriction statement would be:

*N

meaning no restriction at all, and the derivation statement would become:

IM = NBR STUDENT IF LEVEL GQ [1] AND LEVEL LQ [6] AND ATTEND NQ Ø //

Yes, that works in this case. In fact, trying to place the restricting factors into the derivation statement instead of the restriction statement generally succeeds whenever you are working with a statistical operator, such as MDN, MEN, NBR or SUM. But this can lead to dangerous conditions under other circumstances. You really have to understand just what takes place in the system when you enter a restriction statement and when you ask for a particular derivation.

I would guess that a restriction statement immediately sets aside part of our data base so that no matter what the derivation statement asks for, the system will never bother with that part of the data that has been excluded by the restriction statement. In other words, a restriction statement divides a data base into two sections, one to be used by the system in deriving the results demanded by the ensuing derivation statements, the other not to be used at all. At least, not until a new restriction statement or new restriction parameter comes along to disturb this division of the data base.

That is precisely correct. A conditional clause in the derivation statement, on the other hand, doesn't separate one part of a data base from another. It merely asks the system to go from one block of data to another and inspect

it to see whether the stated conditions are met. In this case, there is usually a value resulting from the inspection: the proper value if the conditions are met, no value at all if not.

Now that you have shown me the rules for forming derivations, we might return to our current problem. We are at the point where TRACE begins the cycle of derivations:

```
* 1 6
IM002
IS PROGRAM NAME  2.21
ASSIGN
 ( )USER NAME
 (N)O
*N
GIVE 2 PARAMETERS
 ( )VALUES
 (S)TART
* 7 8
IM003
IS PROGRAM NAME  2.21
ASSIGN
 ( )USER NAME
 (N)O
*N
GIVE 2 PARAMETERS
( )VALUES
(S)TART
* 9 12
IM004
IS PROGRAM NAME  2.21
ASSIGN
 ( )USER NAME
 (N)O
*N
```

```
GIVE 2 PARAMETERS
 ( )VALUES
 (S)TART
*
```

Our derivation cycle completed, we are now ready to look at the results. We can again consult the edit program for this, as we did for examining the results to the first problem.

```
*GOTO E
```

After the edit routine presents its line-up of functions, we select the listing function:

```
*L
LIST VALUES  5.2
 ( )NAMES
 (D)ATA BASE INDICES
 (FOR)  ( )NAME (=)  ( )VALUES  (FIND)  ( )NAMES
END WITH (//)
*IM002 //
TOTAL NUMBER OF BLOCKS = 18
GIVE BOTH  5.21
 ( )STARTING BLOCK  ( )NUMBER OF BLOCKS
* 1 18
BLKN      IM002     SCHOOL
1         436       1
2         622       4
3         537       7
.
.
.
18        701       32
GIVE BOTH  5.21
 ( )STARTING BLOCK  ( )NUMBER OF BLOCKS
*GOTO  5.2
```

```
LIST VALUES  5.2
 ( )NAMES
 (D)ATA BASE INDICES
 (FOR)  ( )NAME (=)  ( )VALUES  (FIND)  ( )NAMES
END WITH  (//)
*IM003 //
```

We continue the listing process with each of the remaining two derived variables: IM003 for the variable showing us the information for grade levels 7 - 8, and IM004 for levels 9 - 12.

This listing function is not the only way of obtaining an exhibit of our derivation results. The display program may also be used to this purpose. To use this program, which displays only derived variable values in table form, we enter the construct table routine, devise a one-dimensional table consisting only of the values for our derived variable, and then request a display from that program. I shall very briefly describe the procedure--the messages and responses are discussed fully in the reference section.

```
GOTO C
 (S)TEP BY STEP
 ( )SHORTHAND
 (L)IST SET UP TABLES
*  TAB R IM002 40 A
ROW IS IM002  CODED A
CASE INDEX IS SCHOOL
TABLE TAB  SPECIFIED  3.6
 (D)ISPLAY THIS TABLE
 (C)ONSTRUCT OTHER TABLE
 (B)ASIC OPTIONS
*
```

I have used the so-called "shorthand" method of constructing the table for the first variable. We can now either have the results for that variable displayed or go on to construct the tables for the other variables and have them displayed consecutively later. This particular table has been constructed so that all the values for IM002 will be presented in a single column, leaving room for a maximum of 40 values--the table was named TAB. All this information was given in the response: R, standing for "row variable," actually tells TRACE to put the values in a column; A signifies that <u>all</u> values are to be recognized.

PART 6

The Third Problem

The third problem is:

> *What is the average grade over all subjects for each student in the school or schools supervised by principal 23 for the three month period--September through November?*

This problem brings us into the area of the so-called "string variables." You may recall that on our data cards are three variables--GRADE, SUBJECT and TEACHER--whose values are separated into six subidvisions each, each subdivision consisting of two characters and each considered to be a value in itself. Each subdivision is separated from another in the same variable by four columns or skips. It is these subdivisions and skips that distinguish a string variable from a non-string variable. Whereas a non-string variable, like SCHOOL, is always treated in its entirety, a string variable can be recognized as any one of its subdivisions at a time; it can be recognized as two or more of its subdivisions; it can even be recognized in its entirety, that is, as a non-string variable. For example, there are six values or subdivisions to GRADE. You are actually allowed to refer to only a single one of these in the derivation statement, or to several that appear consecutively in a block--if you refer to several, you are referring to them as separate values.

How does TRACE handle these string variables internally?

TRACE does this: for each block (logical card) containing perhaps several string variables, it forms individual strings containing only values of the same variable, that is, without any skips. Of course, it is possible to declare string variables initially without any skips, running the separate values one after the other. So, for our input of three string variables, TRACE forms three homogeneous strings for every block: one for GRADE, one for SUBJECT, and one for TEACHER. There is no relationship from one block to another, however. Within each of these strings, the values or subdivisions are placed according to their original input order. Therefore, there is a one-to-one relationship between members of the strings that matches the original. For example, the fifth history grade for a given student will be matched to the teacher for that subject--the fifth value in the string for TEACHER for that particular student for the same month.

Let us start this derivation. We need consider only the restriction, case

index and derivation statements for the derivation since other interactions with TRACE will be similar to those we employed for the previous problems. First of all, the case index statement:

```
CASE INDEX = STUDENT //
```

The next step is the restriction statement:

```
FOR PRINCIP = 23 MONTH = 0 - 2 //
```

This statement tells TRACE to bring into core only those blocks of information that have the value of 23 for PRINCIP and one of the three values: 0, 1 or 2 for MONTH. Therefore, three blocks (logical cards) per student will be transferred from disc to core.

Now if GRADE were not a string variable, the proper derivation statement would be:

```
FM = MEN GRADE EACH //
```

However, since it is a string variable, the statement becomes:

```
FM = MEN SCAN ACROSS GRADE 1-6 EACH //
```

What is the meaning of the terms SCAN ACROSS and the range 1-6?

The terms SCAN ACROSS are always used together and refer only to string variables. These tell the system to regard the variable in terms of individual subdivisions. Our variable GRADE is a string variable consisting of six stated subdivisions, although it is possible that some of these may not hold any information. Prefacing that variable name in a derivation statement by SCAN ACROSS means that the system will treat each subdivision separately as an independent value, as though each were a separate non-string variable. The suffix of a range, such as 1-6, tells the system just which values are to be looked at. No range appended to the variable name will result in having each subdivision recognized. Note that the suffix can only be a range of consecutively appearing subdivisions and that the dash is not separated from the digits by spaces.

What would happen if a string variable name appeared in a derivation statement without the SCAN ACROSS preface?

TRACE would treat the whole string variable as a single entity. For example, GRADE with its six subdivisions of two characters each would be recognized as a non-string variable twelve characters long. This effect has its uses, but not for our purposes.

What TRACE really does then, on orders from this derivation statement, is literally to scan across the working string of values for GRADE and compute a mean of all the values specified in the statement--for this particular statement, all six of them.

What is the meaning of the term EACH that ends the derivation statement?

The answer leads us back into a discussion of case index and the inner mechanisms of the system. I mentioned earlier that TRACE makes a working subset of the data base according to the case index chosen and the restrictions specified. This subset takes the form of a table, 2800 entries in length at maximum capacity. Each occurrence of a case index value is then made an entry, later to be matched in the same table with the corresponding derived value. These case index values are arranged in ascending order. For example, in the first two derivations the index was the variable SCHOOL. This meant that each time the system met an occurrence for SCHOOL in the data base that satisfied the restrictions for the derivation, it placed the value it found there in this table. The final form of this table before derivation processes began then would have shown all the occurrences of the lowest value for SCHOOL at the beginning of the table, followed by all the occurrences of the next lowest value, and so on until the highest value was taken care of. Since each occurrence represented one student (the restriction statement confined the derivation to one block per student), we actually had in that table even before the derivation began the correspondence we wanted between those two variables: SCHOOL and STUDENT. The derivation process consisted merely of counting the correspondences.

Let us examine the case index table as it is used by TRACE in the current problem. First of all, the restriction statement has eliminated all students from consideration but those having an associated value of 23 for the variable PRINCIP. It has furthermore eliminated every block for these qualifying students but the ones having values of 0, 1 or 2 for MONTH. Let us say that principal 23 has one school under his jurisdiction with a total of 400 students. There are then 1200 occurrences of STUDENT in this case index table, three per student.

The term EACH is directly related to this table, as are the other terms in the same category. These terms, called "selective operators," are used at the end of most derivation statements and serve to instruct TRACE in building sets of derived values. I did not include any of these operators in constructing the derivation statements for the first two problems because, in the absence

of any selective operator, a derivation statement that begins with a statistical operator (for example, NBR) is treated by the system as though it had an implicit EACH. This particular operator means: recognize each occurrence of the case index variable. It turned out that was our intent in both derivations.

Now suppose the derivation process for this case index table is beginning. Since the term EACH has been specified, TRACE will recognize and produce a value for each member of the table. Therefore, immediately after this stage of the derivation, the table will look like this:

case index value(STUDENT)	derived value(mean grade)
1	85
1	88
1	79
2	65
2	72
2	71
.	.
.	.
.	.
400	79
400	86
400	89

However, the operator EACH causes a second stage in the derivation process. This time the system repeats the derivation on the individual derived values so that a new table is created consisting of only unique case index values and associated derived values that replace all the individual values that earlier belonged to the various occurrences of the case index values.

In other words, our table is recast into the following form:

case index value	derived value	
1	84	(mean of 85, 88, 79)
2	69	(mean of 65, 72, 71)
.	.	
.	.	
.	.	
400	85	(mean of 79, 86, 89)

And this is precisely what we wished from the derivation: a derived value for each unique value for STUDENT, equal to the mean of all the values for GRADE that qualified for the derivation. Similarly, in the first two derivations, the second stage of the derivation counted all the occurrences of each unique case index value.

What are the other selective operators?

They are the following: FIRST, LAST, ALL, and ONE. The first two refer directly to the occurrences of case index values in this table. They say respectively: recognize only the first occurrence of each unique case index value in the table and derive the designated value for it; recognize only the last occurrence of each unique case index value and make the requested derivation. These two operators would have been useful in either of our first two derivations since they would have confined those derivations to one derived value (count) per student. Instead, I had made the stipulation a part of the restriction statement when I typed:

```
FOR MONTH = 9
```

This is actually the main use of FIRST and LAST: to confine a derivation to one value per unique case index value.

Do these two operators have any use in referring to specific case index values? Suppose that we not only wanted the mean grades for the three month period, but also the mean grades for September alone. Could we have parameterized the last term in the derivation statement and run it twice--once with EACH and the next with FIRST?

No. FIRST and LAST can only refer to the case index table order. This table is not set up on the basis of the order in which the case index variable occurrences occur in the input data set; so you have no assurance that the first occurrence in every set of unique case index values represents September, even though it did in the input.

How do the terms ALL and ONE work?

The operator ALL asks the system to conduct still a third stage in its derivation, this time reducing the table to one case index value (a fictional one) and one derived value. This derivation stage then yields a grand derivation for the case index variable taken as a whole. For example, for the first two

derivations, an ALL operator would have given us the grand total of all the students in the system. The current derivation would have produced a grand average of all grades for all the qualifying students.

This operator, incidentally, cannot produce a derived variable. The value derived is not stored as a variable--TRACE presents it immediately to the user via teletype and then returns to the beginning of the derivation subprogram.

The operator ONE pertains solely to derivation statements that contain conditional (IF) clauses. This operator asks the system to recognize only the first occurrence of each unique case index value that meets the condition spelled out in the derivation statement. Suppose in this current derivation we were only interested in getting a sample mean grade for those students whose attendance fell below a certain level for any one month. We would not be interested in knowing about more than one month. I could then use as my derivation statement:

```
FM = MEN SCAN ACROSS GRADE 1-6 IF ATTEND LS 15 ONE //
```

Now, the very first time TRACE finds satisfaction of the condition within any of the three occurrences per unique case index value in the table, it will derive a result and then go on to the next unique value.

What would happen if you had ended this statement with the operator FIRST?

The system would have recognized only the first occurrence of each unique value, whether or not it satisfied the condition. Incidentally, before we consider any other aspect of selective operators or our current derivation, I would like to stress a very important point that concerns conditional clauses in derivation statements that tend to behave as restrictions.

As I have already mentioned, a restriction statement acts to form a working subset of the data base that is carried into core. This subset is then used to fill the case index table that we have been discussing. Obviously, the smaller this subset the better chance the user has of fulfilling his derivation goal; a subset that cannot be accommodated by this table would have to be divided further artificially. A conditional clause in a derivation statement, now, does not divide the data base at all. The derivation statement appears too late on the scene for such an effect. Therefore, setting into the derivation statement as a conditional clause what could easily have been a restriction in the restriction statement has a wasteful effect--more data than is actually needed is moved into core and more case index occurrences than are necessary for the derivation will be placed in the table. And there may not be room for this excess.

(page 62 blank)

What is the outcome when the user fails to end a derivation statement with a selective operator?

This depends on the type of derivation statement. If the statement begins with a statistical operator, TRACE treats it as though an EACH terminated it. If the statement is not a statistical one, for example:

IM = SQR ALPHA + SQR BETA //

it still recognizes an implicit EACH, but this operator has a different effect upon non-statistical derivations. Here the operator EACH effects a one-stage derivation, not two; that is, each occurrence in the case index table is associated with a derived value at the end of the derivation, rather than each unique value. The other operators: ALL, FIRST, LAST, and ONE behave in this type of derivation just as they did in the statistical.

I have one last question concerning our derivation. Suppose a particular student has not taken the full complement of six subjects but only five, or even less. Our derivation statement nevertheless requests the system to recognize all the subdivisions for GRADE, regardless of content. What happens when no information exists for one or more subdivisions--does TRACE include this empty data in the derivation, perhaps treating it as zero information?

TRACE does not work with blank values or with values that have been changed from blanks to some other set of characters by means of the appropriate function in the input routine. These subdivisions are simply omitted from consideration in the derivation and if that derivation is the determination of a mean, for example, the mean is taken of only the valid subdivision values.

PART 7

The Fourth Problem

The fourth problem is:

> *What is the average grade in English for every student in the system taking that subject?*

By asking that the result be pertinent to conceivably every student in the system, you have introduced a problem beyond your stated problem. This is the matter of having an enormous case index length. The case index in this problem is evidently the variable STUDENT, as it was in the previous problem. However, in that problem you had stressed that the derivation be run only for those students having the same principal. Here, possibly every student in the system is to be considered. Remember: the variable STUDENT was set up to contain a maximum--and therefore a potential--length of 100,000 values. TRACE makes its derivations on the basis of the values in the case index, one result for each value, and it uses an inner table for this purpose. (This inner table was discussed extensively during our last derivation.) Its length defines the maximum number of values that can be handled as case index values. The actual length of this table is 2,800 values; therefore only that number of values for STUDENT can serve as the case index at one time. This means then that we cannot rely upon one derivation to deliver all the results required--we shall have to employ bracketing measures in our derivation and restriction statements so that a series of derivations can be run encompassing all the values for students. The total number of values for STUDENT, which is necessary before we can begin our derivations, may be obtained by a derivation that uses a variable like DISTRICT or SCHOOL as the case index, has no restrictions, and works with the following derivation statement:

```
IM = NBR STUDENT ALL //
```

We now know the number of case index values we have to work with. Let us say that the number is 10,000. This then means that four derivation runs shall be required for each derivation that hopes to encompass all the values for STUDENT as case index, each one deriving results for, say, 2500 students. Our parameterized restriction statement becomes:

```
FOR STUDENT = [1] - [N] //
```

I would then type in my derivation statement, which would be good for all four derivation runs. The values that I would type in as the missing parameters

are the following:

1 2500

2501 5000

5001 7500

7501 10000

What is the derivation statement for this problem?

First of all, let us assume that the identification code for English is 12. The derivation statement then may look like the following:

FM = MEN COR Ø GRADE 1-6 IF COR Ø SUBJECT 1-6 EQ 12 ALL //

I have terminated the statement with the system word ALL because, taking your statement of the problem literally, you wish one general average--applicable to all the students for all the months of the school year. If you had wanted these average grades for each student, I would have typed EACH instead of ALL. And if you had wanted the results in terms of a particular kind of student, I could have included that fact as well in setting up my restriction statement.

What is the meaning of the word COR?

This term stands for "corresponding to" and refers to the working set of stringed values that I mentioned during our discussion of the last problem. Remember that all values for SUBJECT are set into one long string and all values for GRADE are similarly set into a string. These strings can now be compared, as they must in solving our current problem. The arrangement in each string is the original input arrangement, that is, the first value that occurs in a particular card or block for SUBJECT is also the first value that occurs in the working string for that variable. The term COR Ø means a one-to-one correspondence between the strings in question or, in other words, the correspondence is incremented or changed by a value of zero. If I had typed a digit other than Ø in this place, it would have meant comparing elements in the two strings that were in a displaced relationship.

Suppose we wanted to compare the second value for SUBJECT in each block with the first value for GRADE, the third value of the former with the second of the latter, and so forth? This comparison of the nth value of SUBJECT with

the n-1th value of GRADE would require a derivation statement of the form:

FM = MEN COR -1 GRADE 1-6 IF COR Ø SUBJECT 1-6 EQ 12 ALL

What happens at the first matching? It looks as though the first value of SUBJECT in each string is being matched with a nonexistent value; there is no n-1 value for GRADE.

Correct. TRACE does only what you instruct it to do. There will be no match for the first value of SUBJECT. There will be only five matches per string, not six.

Suppose you had typed in the statement:

FM = MEN COR Ø GRADE 1-6 IF COR +1 SUBJECT 1-6 EQ 12 ALL

It seems as though this statement should accomplish the same results as the previous statement, except that the missing match will occur at the final value of each SUBJECT string rather than at the first.

That is also true.

It seems, however, that we cannot really obtain a final answer to the problem by these means--what we have are four partial answers that only as a whole yield a result. How can we obtain a comprehensive result by this method?

Whenever a derivation is done in sections like this, the comprehensive result can usually be obtained by writing one or more further derivation statements using the variables derived from the sectional derivations. Depending on the nature of the derivation, this can be a trivial matter, for example, merely adding togethe a series of sums to obtain a grand sum, or quite involved, as happens to be the case with our derivation. Suppose we look at our problem again. We are interested only in those students who are taking English. This number might be 95 per cent of the total or it might be 50 per cent; whatever it is, it certainly cannot be presumed to be the same for each of the four segments into which we have arbitrarily divided our data base. One way of solving this is by way of recalculating this grand average using the partial grand averages and the number of students each partial average represents. In other words, we multiply each partial average by the appropriate number of students and then divide the sum of those products by the sum of all the

students. We can determine how many such students there are in each segment easily enough, for example, by adding another derivation statement to our parameterized derivation:

```
FM = NBR STUDENT IF SUBJECT EQ 12 ONE //
```

The system word ONE is used because we want a student counted only once: remember that each student has information for ten months, so that this condition could be fulfilled a total of ten times for a particular student.

It then becomes a relatively simple matter of calculating a true grand average. This is done through a feature of TRACE that functions as a virtual "desk calculator." This feature is built into the derivation routine and takes the place of an ordinary derivation statement. We can reach it, however, without recourse to case index or restriction statements, which are meaningless in this context. First, I call the derivation program, then immediately jump to the derivation statement message. Instead of typing in a normal derivation statement, I enter the following:

```
*COMP =
```

followed by the computation expression that I need. Suppose that the four derivation runs returned the following partial averages:

```
77    82    81    77
```

and we discovered that these averages represented the following numbers of students respectively:

```
2401    2378    2338    2117
```

Our desk calculator statement might then look like this:

```
*COMP = ( ( 77 * 24Ø1 ) + ( 82 * 2378 ) + ( 81 * 2338 ) + ( 77 * 2117 ) ) /
                    ( 24Ø1 + 2378 + 2338 + 2117 ) //
```

What is another way of solving this problem? I imagine that further derivations could be used to obtain a grand result.

We can approach this another way. First, the initial derivation statement could end with the word EACH instead of ALL.

```
FM = MEN COR Ø GRADE 1-6 IF COR Ø SUBJECT 1-6 EQ 12 EACH //
```

Each of the four derivation runs would then yield all the individual grade averages for the students included within the corresponding ranges of values for STUDENT. For example, the first derivation run would produce 2401 values--average grades in English for 2401 students. If we could combine the four sets of values, we would then find ourselves with a variable that represented all the individual average grade values. TRACE does provide a method for this type of value-combining: the CMB statement in the derivation routine (another variation of the standard derivation statement) which, for this example, might look like the following:

```
FM = CMB R1 R2 R3 R4 //
```

where R1, R2, R3, and R4 are the four partial derived variables. The new derived variable (which we can call R5), containing now <u>all</u> the values of the four variables, can be used to determine a grand average of all the individual averages.

The simplest way of doing this is by constructing a one-dimensional table for display. TRACE will then provide us with a mean of all the values of the variable specified for the table. This is done automatically: after any display of a table having at least one numeric variable, TRACE prints the mean, median and standard deviation for each of the numeric variables. So, for this table, I can enter the construct table routine and type the following shorthand description:

```
ENGLISH R R5 1 S
```

This tells the system that the table name is ENGLISH, that the table will display values of R5 in a column, that there is to be only one interval and that is to be a specified one. I can then specify for that interval the whole range of possible values for R5, namely: 1 100.

The resulting table will then show us only one cell, containing the total number of values of R5. However, below the table will appear, as one of the statistical characteristics of the table, the mean of the values.

PART 8

The Fifth Problem

Can you construct a table that will display the correlation between the average grade of a student and his attendance record?

This is a legitimate request, because a two-dimensional table must consist of values whose variables have the same case index. In our problem, both derived variables would have the same index. Let us assume that we have already derived both variables; the average grade for each student in the system for all his subjects for the entire school year we shall call MENGRADE; the year-long attendance record we shall call DAYS. What we are actually looking for in the table is the frequency with which certain grade intervals match particular ranges of total days of class attendance. For example, at one extreme is a table that compares all possible values of MENGRADE with all possible values of DAYS. This resultant table would roughly look like the following:

(values of MENGRADE)

(values of DAYS)	0	1	2	3	4	.	.	.	.	.	.	97	98	99	100
0															
1															
2															
3															
4															
.															
.															
.															
.															
159												2			
160												1			

As you can see, most of the intervals would be empty; very few students would have average grades lower than 60 or higher than 95--very few would have perfect attendance records or would have missed more than 20 percent of the school year. On this table, I have indicated that two students had achieved a grade average of 97 together with one absence, one student had the same

average and a perfect record. All the other cells listed here are empty; most of the matches would fall at less extreme levels.

A more reasonable outlook might result in a table that, for example, compared MENGRADE values between 50 and 100 with DAYS values between 100 and 160. There are now several ways in which this type of table could be set up. You could still ask for a matching of absolute values as we theoretically had for the previous table. You could, on the other hand, ask for a matching of certain meaningful intervals, for example, MENGRADE values between 50 and 60, 61 -70, 71 - 80, 81 - 90, 91 - 100; and DAYS values having the ranges: 100 - 110, 111 - 120, 121 - 130, 131 - 140, 141 - 150, 151 - 160. In this case, you would be specifying to the system exactly how you wanted your intervals set up.

On the other hand, you could ask TRACE to determine your intervals for you. You might not know just how your values were distributed and therefore would hesitate to designate intervals that would be relatively meaningless. For example, in our problem, suppose there was such a distribution of values for DAYS that 90 percent fell between 135 and 143 days. This distribution would probably have been unforeseen by the user. It might have been wise then in setting up the table to let TRACE draw up its own intervals and perhaps discover this important range of values. This TRACE can do in two ways. It can create intervals so that each interval contains as nearly as possible the same number of cases, that is, frequencies. Or, it can create intervals so that they are all of equal width, that is, the ranges of values in all intervals would be as nearly equal as possible. It is the first of these two options that would probably best solve this distribution. Suppose we asked TRACE to draw up ten intervals. It would probably then set one interval for all values below 135 or perhaps 136, and another for values above 143, or again, perhaps 142. It would then save eight intervals for the densest distribution, that is, 135 to 143.

To illustrate the first of these two TRACE-determined interval settings, let us assume that the intervals for MENGRADE are as specified above, i.e., 50 - 60, 61 - 70, etc., and that we have asked the system to draw up ten intervals so that each contains an equal number of matches. The table then might look like this:

		(intervals of MENGRADE)								
		50-60	61-70	.	.	.	.	.	.	91-100
	0-128	23	18							2
	129-132	29	21							2
	133-135	20	16							2
(intervals of DAYS)	.									
	.									
	.									
	153-160	18	17							19

In this table, the row totals would be fairly equal. The intervals for DAYS are the row variables, the intervals for MENGRADE the column variables.

We could also have asked for a table whose intervals for DAYS were of equal width. TRACE would take our specified number of intervals, for example, 10, and draw up intervals so that the lowest interval began at the lowest value found, and the tenth ended at the highest.

Of the several variations possible, it seems most appropriate for us to use the combination of user-specified intervals for MENGRADE, and the system-determined equal numbered intervals for DAYS. Let us assume that we have entered the subprogram that constructs tables. Since the construct table subprogram is discussed fully in its reference section, I shall not explain in any detail what each message in that program means. The message flow is as follows:

```
CONSTRUCT TABLE  3.0
 (S)TEP BY STEP
 ( )SHORTHAND
 (L)IST SET UP TABLES
* S
```

Here I have asked for a train of messages that I shall use in setting up my table. In the so-called "shorthand" method, I would have set into one line all the information required by TRACE in manufacturing this table.

```
SPECIFY ROW VARIABLE  3.1
 ( )NAME
 (N)ONE
* DAYS
NUMBER OF INTERVALS  3.31
 ( ) 1 TO 10 FOR COL   1 TO 40 FOR ROW IF DISPLAYED ON TTY
* 10
RESPOND WITH:  3.311
 (N) FOR EQUAL NUMBER
 (W) FOR EQUAL WIDTH
 (S) FOR SPECIFIED WIDTH
 (A) FOR ALL VALUES
* N
SPECIFY COL VARIABLE  3.2
 ( )NAME
 (N)ONE
* MENGRADE
NUMBER OF INTERVALS  3.31
 ( ) 1 TO 10 FOR COL  1 TO 40 FOR ROW IF DISPLAYED ON TTY
* 5
RESPOND WITH:  3.311
 (M) FOR EQUAL NUMBER
 (W) FOR EQUAL WIDTH
 (S) FOR SPECIFIED WIDTH
 (A) FOR ALL VALUES
* S
 ( )GIVE LOWER AND UPPER LIMITS FOR EACH INTERVAL
* 50 60 61 70 71 80 81 90 91 100
MODIFY OR NAME TABLE  3.4
 (R)OW
 (C)OLUMN
 ( )TABLE NAME
* GR+ATTEN
```

```
ROW IS DAYS CODED I
COL IS MENGRADE CODED F
CASE INDEX IS STUDENT
TABLE GR+ATTEN SPECIFIED  3.6
 (D)ISPLAY THIS TABLE
 (C)ONSTRUCT OTHER TABLE
 (B)ASIC OPTIONS
* D
DISPLAY ON  4.1
 (S)COPE
 (T)TY
* T
TABLE GR+ATTEN
ROW   DAYS
COL   MENGRADE
```

Now will follow the same type of table that I outlined earlier, having the same intervals for MENGRADE and DAYS. There is one convention in TRACE's display of tables that must be brought up, however. When either a row or column variable is arithmetic (integer or floating point), the displayed table does not exhibit both the lower and upper limits of every interval. This it does only for the lowest interval; the others are identified only by the upper limit. Our column variable headings then would read: 50, 60, 70, 80, 90, and 100.

The display program does add a few features to the table, however, At the end of each row and below each column will be found three new values; they are in order of appearance: rejected values, total values, and percent of this total compared to the grand total. The rejected values are those matches which cannot be placed in any interval, for example, for the first row a student who attended class for 125 days and achieved an average grade of 45 would not qualify for inclusion in any of the intervals--his case index value would be a rejected value. Similarly, a student who achieved a grade average of 85 and whose attendance record gave a value of 260 because of a keypunching error would also be a reject, this time a reject in a column, not a row. Missing data also qualify as rejects.

Below the table proper will appear a grand total of case index values. A list of various statistical factors will also be typed. These are listed and defined in Section 2.4.

PART 9

Other Data Bases

What other environments specifically can you describe as suitable for TRACE?

Besides this educational system that we have been working with, there are many others that we could have considered: personnel records, census taking, surveys, experiments, for example. I can briefly give you some sample data bases for these environments:

census-taking

district

address

family (total membership)

member

income education distance from work marital status age

In this data base, the information for member would be repeated for each member in the family. The data cards could be set up to accommodate any number of members.

personnel

name

birthdate

social security no.

marital status

department

current salary

current job title

previous job title date received

salary date of increase

education

aptitude test score

Two groups of information can be repeated, one within the other. For each employee, there may be several previous job titles; within each job title, there may be several different salaries. For example, we could set up our card input so that following the current job title was allotted space for six previous job titles, within each of these six areas room for six salaries.

On the other hand, this data structure might be changed so that the variable for "previous job title," for example, is placed on the same level as "current job title." This would mean a repetition of information for that variable as often as that occurring for the current job title variable. This might then result in a flexibility in setting up the card deck. For example, previously one block (one logical card) might have been set to contain all the information for a single employee. Now a separate block would be necessary for each previous job title. Again, six blocks might be allotted for this purpose.

survey

respondent

date

answer to question 1

answer to question 2

.

.

.

answer to question n

Here is probably the simplest of data structures: none of the information is repeated.

experiment

condition

subcondition

pair

member

trial

outcome turn behavior questionnaire response

In this sample psychological experiment data base, the trial information would be repeated for each trial. There may be several trials for each member.

Notice what the variables corresponding to STUDENT in our working data base would be for each of these data bases: member in the first, either name or social security no. in the second, respondent in the third, and member in the last. The values for these variables would be unique in their respective data sets; that is, none of the values could be repeated within the same data set.

Would you briefly define a problem in one of these environments and demonstrate how TRACE would go about solving it?

Let us look at the final example of our new data bases. Suppose this were an experiment wherein several pairs of subjects, under certain conditions, were to be tested in order that some correlation might be found between each pair's behavior during the test and a questionnaire response on the one hand, and the outcome of the test itself. There would be a possible correlation for each trial of the test. Suppose the data looked like the following for the variables in question (one trial):

PAIR	MEMBER	RESPONSE	BEHAVIOR	OUTCOME
1	1	C	N	L
1	2	B	C	M
2	1	C	C	L
2	2	A	E	H
3	1	A	N	M
3	2	A	C	M
.				
.				
.				
n				

Each pair has two members; a response can be A for good, B for fair, and C for bad; behavior can be cooperative(C), non-cooperative(N), or extemporaneous(E); the outcome can be low(L), high(H), or medium(M).

Since the results are to be in terms of PAIR, that variable is the case index. What restrictions we apply will depend upon the number of trials we want to consider: all trials would mean no restrictions; otherwise, we must mention in the restriction statement the specific trials we are interested in.

The next step is to find a derived variable that will express both a value for RESPONSE and for BEHAVIOR, and not for each member of each pair (as in the data base) but for the pair as an entity itself. This requires the variable to express four "ideas" or values: the response of the first member plus his behavior, the response and behavior of the second.

Probably the simplest and most general way of producing this variable is by first deriving a variable for each of these four components, and then coalescing the results into one variable. We can obtain derived variables for the components (which are simply input variables to begin with) by parameterizing one restriction statement and one derivation statement. Having specified PAIR as the case index, we type the following restriction statement:

```
FOR MEMBER EQ [1] //
```

and, as the derivation statement:

```
AM = [R] //
```

We must then run through the derivation four times. Perhaps the first two times with RESPONSE as the derivation statement parameter and 1 and 2 as the restriction statement parameters. The third and fourth runs would have BEHAVIOR as the derivation statement parameter, with corresponding alternation in the restriction statement.

By use of the MRG statement (a variation of the standard derivation statement) in the derivation subprogram, we can now merge two variables at a time into one. This MRG statement would be repeated until all four variables were merged together. For example (suppose the four variables were named T1, T2, T3, and T4):

```
AM = T1 MRG T2 //
```

(name this variable T5)

AM = T3 MRG T4 //

(name this variable T6)

AM = T5 MRG T6 //

The resulting values for the first three pairs would look like the following:

PAIR 1 CBNC

PAIR 2 CACE

PAIR 3 AANC

We could have merged the separate values in a different order, of course. At any rate, there are 81 possible values for our new variable.

The next step is to find a derived variable describing the cumulative outcome for each pair. This is done in a like manner. The results might be:

PAIR 1 LM

PAIR 2 LH

PAIR 3 MM

The final step would be to set up a table that displayed the correlations. Since there are 81 possible values for the response and behavior variable, there cannot be an interval for each possibility in one table (a table can hold a maximum of 40 rows and ten columns). The solution then is either to construct two or more tables or else to construct a table such that each interval represents two or more possible values of the response and behavior variable. This problem will not exist for the display of the outcome. Here we can have only nine possible values--within the maximum of ten columns.

SUMMARY TO SECTION 1

We have now looked at the essence of TRACE and also examined a large body of its particulars. We have seen it handle a rather typical data base in a variety of problems and have described a group of other data environments. In helping the reader approach the role of a TRACE user, this first section has shown him nearly all he needs to know before actually constructing a data base of his own and presenting a problem to the system. The following section and the appendices are designed to instruct the user who has reached just that point. This document does not attempt to show him whether his particular data base can fit TRACE or how his individual problems can be solved; it simply provides him with a framework upon which he can play his imagination and knowledge.

(page 82 blank)

SECTION 2

A REFERENCE WORK ON TRACE

SECTION 2

INTRODUCTION

This reference section is meant to be used in two ways. First, it may help to clarify some portions of the first section of this document. The user may at times find himself turning from a discussion of a particular problem in that section to its counterpart in this section in order to view it from a more detached perspective. Second, it should serve as a reference to all actual work done with TRACE. Every message is described--both the requests from the system and the expected responses from the user.

Each subprogram commands a separate subsection; each subsection is built upon the messages that compose it. These messages are discussed in the order they are met with during an actual interaction with the system. Each message with its discussion is enclosed within black borders.

Messages from the system always make a request, sometimes preceded by a statement of fact. The request line is always followed by one or more lines that contain and explain the form to be used by the user in responding to the request. These lines are always identifiable by the presence of parentheses. A pair of parentheses that encloses an entity, for example, (B), declares that that entity must be typed by the user in his response. A pair of parentheses that encloses nothing is always followed by an explanatory word or words that are not to be typed themselves but do suggest what is to be typed; for example, ( ) NAMES declares that the user must type in a name or a set of names. These response forms may be intermingled on one line or may appear on different lines. If on different lines, a choice is offered. Example:

```
GIVE DISC INFORMATION
 ( )NAMES
(B) ( )VALUES
 (C)
*
```

In this hypothetical message, the user has three options: he may type in a name or a set of names; he may type in the letter B followed by a set of values; or he may type merely the letter C.

All messages from the system end with an asterisk, generally on a line all by itself. The response from the user is to be entered on this line after the asterisk, separated from the asterisk by one or more spaces if he wishes. The response is then conveyed to the system by means of a carriage return.

The user need not obey the stated options i. responding to a message. He may alternatively type in a command that begins with the following:

GOTO

and is completed by either the reference number of a message (many messages contain at the end of their first lines a number code, e.g., 3.11) or by the code letter or letters of one of the subprograms. The system then brings the user to the point requested. If the passage is to be from one subprogram to another, the user must follow the command GOTO by the code letter of the program visited. If the passage is within a single program, for example, from one function to another, the user must type the reference number of the message desired. The user may <u>not</u> travel from one subprogram to another by typing the reference number of the first message of the program visited, for example:

GOTO 3.0

It is recommended that you do not try to travel from a point within one sub-program to another point <u>within</u> another subprogram. Travel from one program to another should be to the beginning message of the new program.

A response that is constructed of information that can cross into a second teletype line is ended not by a carriage return alone, but by this preceded by a double slash. The system understands then that a carriage return alone does not end a line. This convention is shown within any message that elicits this type of response; it appears as the following line:

END WITH //

Note: there must be at least one space between the double slash and the last character of the statement. Usually, when a response is to be terminated by a double slash, the system issues a perfunctory message after each line reminding the user that more information is expected. This is the brief message:

CONTINUE

followed by a carriage return. The system does not send this message after the terminating double slash.

Whenever a series of values occurring consecutively in the data base is to appear in a response, it may be represented by the beginning and ending value

separated by a dash mark. For example, the values for a particular variable may be: 34, 35, 36, 37, 38, and 39. These may be typed as: 34-39 or 34 - 39. The matter of an intervening space between the dash and a value depends on the individual message.

Since this reference section will most likely be visited quite often by the user and usually for relatively brief periods, he may find it convenient to place plastic index tabs on the beginning pages of each section part, that is, the beginning of the subprogram descriptions. This device could also be used advantageously for various other frequently visited pages, for example, the description of the input dictionary in the appendices.

The following is a list of the TRACE II subprograms together with their corresponding code letters. These codes are used to obtain entry to the various subprograms.

Input	I
Derivation	DE
Construct table	C
Display	DI
Edit	E
Output	O
Statistics	S

SECTION 2--TABLE OF CONTENTS

SECTION 2--TABLE OF CONTENTS (cont'd)

PART 1

The Input Subprogram

The user's entire collection of data, called the data base, may consist of up to ten data sets. A data set is defined as the amount of data brought into the system during any one run of the input routine. It may be a file from disc, a file from tape, or consist of teletype input. The data base itself may hold sets coming from any of these sources or all of them. Each data set is described by one dictionary; therefore, each data set has only one logical card form.

The purpose of the input routine is to transfer a data set from the input device to disc, where it remains throughout the entire interaction with the TRACE program. Various sections of the data base are then brought into core whenever necessary. Once having input a data set, the user cannot reenter the input routine for the purpose of working with that set; he must use the edit program for making any alterations, or else he may delete the entire set. The form of the data set is outlined in Appendix 3 this subsection treats the disposition of a data set after it has already been assembled and after the user has already entered the time-sharing system. This latter step is outlined in Appendix 4.

The user will note that the input device selected will determine the type of messages received. Some are common to all devices, others are peculiar to a particular device. This subsection will therefore be divided into three parts, each describing the series of messages sent for a particular device. The messages that are common to all three will be described fully in the first of these parts, and repeated without explanation in the others. These divisions deal with tape input, disc input, and teletype input, in that order.

```
     SELECT INPUT DEVICE  1.10
      (T)APE
      (TT)Y
      ( )DISC NAME - 51 CARDS/SECTOR -
     *
```

This is the first message of the input routine; it is received automatically by the user as soon as he has entered the TRACE system and named his data base, assuming this data base is a new one. If it is one that is already known to the system by having been previously stored on disc, the input routine is not entered and the user is expected to begin working with his data base.

The first response states that the current data set resides on tape, having been prestored from cards, and is to be entered via tape drive. The second response signals that the data will be entered via the teletype. The third response is used whenever the data base is on disc; the user is asked to provide the disc file name and is also reminded that a disc sector must hold 51 cards per sector.

--Tape Input Messages--

ENTER REEL NO FILE TAPE = *

The first message received by the user after he has indicated that his data base is on prestored tape, it asks him to supply the reel number for the tape containing his data base or, if the data is on more than one tape, the reel number of the tape about to be used. After he has typed in the reel number, the system delivers the message:

$WAIT

This reminds the user that a computer operator is retrieving the tape requested from the tape library and will then mount it on a tape drive. TRACE does not communicate another message until these physical operations have been completed. The system then notifies the user by the following message that the tape has been mounted:

$FILE TAPE = * DRIVE m REEL n

where m is the tape drive number and n is the reel number.

MISSING DATA 1.11

SET ARITHMETIC VALUES TO OTHER THAN ZERO

( )VALUE

(N)O CHANGE

*

This is the first of two messages concerning the user's disposition of missing data, that is, blank columns on the data cards. Both allow him to set the missing values to any desired values, thus circumventing the system's way of dealing with such data. The user must, however, remain consistent in his choice from one data set to another; that is, missing values should be treated equally throughout the entire data base.

This first message, dealing with missing arithmetic values, asks the user to either accept the system's convention of treating each missing value as an arithmetic zero--the second response--or else specify the value to be used by the system in replacing the blanks. You are restricted to specifying only digits as the replacement values.

SET ALPHANUMERIC TO VALUE OTHER THAN BLANK

( )VALUE

(N)O CHANGE

*

This second message dealing with missing data values asks the user to either accept the system's convention of treating each missing value for an alphanumeric variable as a Hollerith blank (octal 60) or else specify the value to be used by the system in replacing the blanks. Any teletype character can be selected by the user for this purpose except for the following:

$! " / + - *) ( ] [

GIVE ESTIMATE OF NUMBER OF DATA BLOCKS

*

The user is expected to reply with an estimate (preferably a slight overestimate) of the total number of blocks (logical cards) in the current data base. The system then uses this estimate in setting up tables that can hold as large as possible a portion of the data in core before transfer to disc storage. This estimate plays a part in allowing the system to process more than one variable at a time.

This message is issued only if the user has not already indicated on one of his input control cards (See Appendix 3) what the total length of the data base is expected to be.

Caution: if the estimate is lower than the actual size of the data base, data blocks beyond the specified size will not be recognized.

```
VARIABLES STATUS:
list of variable names and statuses
  :
  :
INDICATE VARIABLES TO BE READ  1.12
 ( )NAMES
 (A)LL
*
```

The message begins by presenting to the user a list of the variables of the current data set (all the variables found on the current tape) together with their residences, that is, whether they are still on tape or whether they are already on disc. This message is delivered whenever the system has looked at the user's dictionary and is about to read in the data. Therefore, the first time that this message is sent to the user, all the residences (statuses) are TAPE; that is, no data has yet been stored on disc. However, if only a portion of the available variables have had their values read in during a previous input run (TRACE either reads in every value of a particular variable or it reads no values), the list can resemble the following:

```
SCHOOL      TAPE
STUDENT     DISC
DISTRICT    DISC
```

This listing tells the user that TRACE, during the previous run through the input routine, read in only the values for the variables SCHOOL and DISTRICT. The values for DISTRICT remain unread on tape. He has now on this subsequent run through the routine, the opportunity of asking the system to try again to read these values. After every listing of the read-in variables, TRACE appends a request for the reading of more variables.

The reason all the variables may not be accommodated during any one run is that TRACE works with a limited core space for transferring values from tape to disc.

It may actually be true that this situation is repeated many times: the user has a very large data set; he asks the system to read all his variables; he is informed that only a portion of them has been read; he again asks for a reading of all the variables; the system again informs him that more of his variables have been processed but not all; the user repeats his demands until the system indicates that all variables are residing on disc.

A similar procedure is followed when the data base consists of more than one data set (more than one tape): the system looks at the new dictionary and puts out an appropriate listing of variable names and statuses.

As is evident, the user in responding need not automatically ask for a reading of all his variables--his problem may not require every variable. He may type in only those variables that are necessary and that seem capable of being handled by the system in one pass.

Following the user's response to this message, the system begins to accept the variables specified. There is therefore a pause at this point in the system/user communication--until the data has been processed.

VARIABLES READ IN:
list of variables read in
(D)EFILE AND RETURN
(R)EWIND FOR MORE VARIABLES
*

This message, issued after each reading of a specified set of variables (see last message), allows the user to see what variables have actually been placed in disc storage. He then has two response possibilities.

The first response declares that the user is finished with the present tape and wishes either to work with the data on disc or else has another tape, disc or teletype input that he wishes processed. This response takes him to the end of the input routine and its final message (message 1.20), where he may take appropriate measures.

The second response is to be used if the user wishes to input more data from the same tape. The system then returns him to message 1.12, giving him a listing of variables and statuses and asking him to name a new set of variables to be read.

```
      INPUT COMPLETE  1.20
       (C)ONTINUE WITH INPUT ROUTINE
       (B)ASIC OPTION
      *
```

This is the final message of the interaction between the user and the input routine; however, if he has another tape, he has the option of returning to the very beginning of this tape read cycle, that is, the message that asks for the reel number of the tape. The tape read cycle of messages is then repeated in its entirety for the new tape. For this, you type the first response.

Otherwise, having no other tapes to be read, the user types the letter B followed by the appropriate code of a subprogram.

--Disc Input Messages--

This cycle is precisely the same as that for reading variables from tape, but for one exception. The message that asks for the reel number of the tape is omitted; in other words, the system, having received from the user the name of the disc file, proceeds immediately to message 1.11 (about missing data).

Note: when the listing of variables and their statuses appears, the correspondence is still TAPE for the variables not yet read in, and DISC for those already placed in disc storage. (Disc storage here means the disc that TRACE uses to store all input, whether this input emanated from another disc, or from tape, or via teletype.)

--Teletype Input Messages--

The following two messages are the first in the teletype input cycle:

```
MISSING DATA 1.11
SET ARITHMETIC VALUES TO OTHER THAN ZERO
 ( )VALUE
 (N)O CHANGE
*
```

```
SET ALPHANUMERIC TO VALUE OTHER THAN BLANK
 ( )VALUE
 (N)O CHANGE
*
```

The first message peculiar to the teletype cycle is:

```
GIVE DATA DESCRIPTIONS - ONE PER TTY LINE  1.30
FOR ARITHMETIC VARIABLES:
 ( )NAME  ( ) I OR F
FOR ALPHANUMERIC VARIABLES:
 ( )NAME  (A)  ( )TOTAL # CHRS  ( )# CHRS PER SUBDIVISION
TERMINATE BY:  (END) (CASE)  ( )NAME
*
```

The user is now asked to type a list of variables, one to a line and each variable named identified with respect to type. What the user is communicating to the system at this point is the dictionary for his data set.

For arithmetic variables, you type the variable name followed by either the letter I for integer or the letter F for floating point (space between name and code, of course).

(cont'd)
For alphanumeric variables, the name is followed by the letter A, then the total number of characters per block, finally the number of characters per subdivision--the two figures are equal (or the second can be zero) for ordinary variables, unequal for string variables.

The dictionary is terminated by a line that begins with the words END CASE followed by the name of the variable that is most likely to be chosen as the case index variable. This variable name is mostly a convenience for the system; it can be changed during use.

```
     GIVE CHARACTER TO DENOTE A BLANK  1.32
      ( )CHR
     *
```

This message is forthcoming only if at least one of the variables named in the dictionary is alphanumeric. This message asks the user to provide the system with a symbol to use and to recognize if real blanks are to be contained within the alphanumeric data. This is necessary because the system recognizes blanks only as separators between entities on a line of type; it cannot perceive the difference between an intended blank, for example, a blank between two words that form a single entity as in: MR SMITH, and a separating blank. Any character but the following may be chosen:

```
     $  !  "  /  *  -  +  )  (  ]  [
```

```
     GIVE NUMBER OF BLOCKS FOR THIS DATA SET  1.31
      ( )NUMBER
     *
```

The user types in the expected number of blocks for the current data set, preferably a slight overestimate.

(page 100 blank)

The system now begins a train of messages, one for each variable named in the dictionary. Each of these messages is a request for all the values that belong to the associated variable. The message names the variable and also tells the user how many values it expects, for example:

```
FOR DISTRICT GIVE 24 VALUES
*
```

The user then proceeds to type in all the values belonging to the variable DISTRICT. If the end of the teletype line occurs before all the values have been typed, the user ends the line with the usual carriage return which the system answers as follows:

```
CONTINUE
*
```

He then tries to complete his input for the variable. If the same situation is again encountered, TRACE recognizes it and sends another CONTINUE message.

Warning: if a value is so long that it extends beyond the limit of a teletype line, this is signaled to the system by pressing a line feed before the carriage return.

```
INPUT COMPLETE  1.20
 (C)ONTINUE WITH INPUT ROUTINE
 (B)ASIC OPTION
*
```

As in the other cycles, this final message asks the user to either select another subprogram or else return to the beginning of this subprogram, where he may enter another data set.

PART 2

The Derivation Subprogram

This is the routine that does the actual work of deriving the solutions to the various problems proposed by the user. The routine is entered specifically to create so-called "derived variables," that is, variables that owe their existence either to other derived variables or to input variables. Each derived variable consists of a collection of individual values that are the results of a particular derivation--the solution to a problem.

Since the mechanism of this subprogram has been thoroughly described and analysed within the dialogue section of this document, no attempt will be made to undertake a similar type of examination of every topic here. Instead, the reader will be referred to appropriate parts of that first section.

This current subsection is divided into three parts: the first treats the sequence of messages and responses that form the program; the second explores the various components of the statement that controls the derivation process--the derivation statement; the third is a demonstration of how the derivation routine can be used to simulate a desk calculator.

--Sequence of Messages--

The first message in the derivation routine:

```
DERIVATION  2.0
 (S)TART
 (L)IST AVAILABLE VARIABLES
*
```

Use the first response if you are starting a derivation. This will bring you to the first working message of the routine (message 2.1). The second response is used if you wish to receive a listing of all the variables currently available. It is usually typed when so much editing and deriving have been done that the current list of variables is far different from the original set of input variables.

There are three ways of using (typing) the second response:

(1)	*L D	this asks the system to list all the derived variables together with such information as the case index variable, the number of values in the variable, and its type
(2)	*L n	where n is the number of a data set. This asks the system to list variable names and associated information for every variable in the designated data set
(3)	*L ALL	this asks the system to list both derived variables and all the input variables, according to data set, giving the associated information

```
CASE INDEX  2.1
 ( )
 (N)O CHANGE
*
```

The first response is the name of a variable to be used as the case index for the forthcoming derivation. The second response tells the system to use as the case index the variable whose name appears on the END CASE control card in the data deck, if there is one. (See Appendix 3.) This variable is usually the one around which the data has been constructed and therefore would be expected to serve as the case index for most of the problems run on the data. Otherwise, the user must type in the name of another variable (the first response). No harm will be done by typing in the name of the variable (first response) even if it already appears on the END CASE card.

The concept of case index is explored in great detail in the dialogue section, especially Parts 4 and 6. In essence, it means asking the system to form a subset from your data base consisting of all the values of a designated variable and running a derivation so that a result will be associated with each of these values. This case index variable is then a variable to which the results of a derivation pertain. For example:

average salary (results) per employee (case index)

Only input variables can serve as case index variables; a derived variable can, however, be converted to an input variable by means of the data set forming function (message 5.4) in the edit subprogram.

Alphanumeric variables may qualify for service as case index variables only if they do not consist of more than eight characters each. Such a variable must furthermore have as its first character one of the digits, 0-9. There are no restrictions on integer and floating point variables.

Note: if a derived variable is to appear in a derivation statement, the case index of the derivation must be the same as that variable's case index. If two or more derived variables appear in the same statement, both must have the same case index.

```
RESTRICT DATA  2.11
 ( )RESTRICTIONS END WITH //
 (N)O
*
```

The restriction statement acts to partition from the data base a working subset that is carried into core. This subset is then further partitioned or sorted by means of the case index statement. For example, if a data base contained a variable representing employees and also a variable for their age brackets, a problem could be stated restricting the results to those employees associated with a particular age bracket, or with any one of several specified age brackets. The restriction is always in terms of actual values of the restricting variable, for example:

```
FOR AGE = 21 - 35 //
```

or

```
FOR AGE = A B C //
```

Each restriction statement begins with the word FOR and ends with a double slash. The word FOR is not be repeated even if several variables are named in the statement.

Examples of restriction statements:

```
FOR EMPLOYEE = XYZ //
FOR AM003 = 2 3 4 5 6 8 11 15 //
FOR AGE = 21 - 35 //
FOR X = 21 Y = 25 Z = 1009 1010 1012 //
```

Note: when a series of consecutively appearing values is denoted by the beginning and ending value separated by a dash, there must be at least one space separating the values from the dash, for example:

```
FOR AGE = 21 - 35 //
```

Any variable may serve as a restriction variable. However, if alphanumeric, it must not exceed eight characters in length. If the variable is an input variable, it must be in the same data sets as the case index variable for the same derivation. A derived variable must have as its case index the variable

named as the case index for the derivation. If a string variable is named, all characters in that variable are to be considered together in the restriction; for example, if GRADE is a string variable eight characters long, made up of four two-character subdivisions, the restriction statement containing it must use as restricting values eight-character values. (If GRADE is longer than eight characters, it cannot be used at all.) The user who wishes to employ specific characters of a string variable to restrict a derivation may include this type of restriction in a conditional clause derivation statement).

Whenever variables occur in series in a single restriction statement, TRACE recognizes them as connected by an implicit "and" and therefore proceeds to devise the subset on the basis that each individual restriction must be satisfied. For example, in the statement:

```
FOR ALPHA = X BETA = Y BETA = Z //
```

TRACE considers as qualifying for the working subset only those values which meet all three criteria.

On the other hand, when values of a variable occur in series in a restriction statement, TRACE recognizes them as connected by an "or" and therefore proceeds to devise the subset to include satisfaction of any one of the restricting values. For example, in the statement:

```
FOR ALPHA = X Y Z //
```

TRACE considers as qualifying for the working subset all those values that can meet at least one of these criteria.

The restriction statement is explored at great length in the dialogue section of this document, especially Parts 4 and 5.

```
GIVE DERIVATION STATEMENT  2.2
 ( )END WITH //
*
```

The process of solving problems and deriving results can only be fully understood by reading the entire dialogue section. This statement, which handles all the work of the derivation, is in essence the equation for the problem at hand. The various components of this statement and the tools available to the user in constructing it are presented in the second part of this subsection.

```
system variable name
IS PROGRAM NAME  2.21
ASSIGN
 ( )USER NAME
 (N)O
*
```

Immediately following the derivation operation, the system sends this message to the user informing him of the name given by the system to the variable just derived. This name consists of the left hand side of the responsible derivation statement (that is, AM, FM or IM) plus a number that reflects that variable's chronological position among all derived variables. For example, the first derived variable is labeled AM001 if it is alphanumeric or IM001 if integer; the 39th derived variable is labeled AM039 if it is alphanumeric, IM039 if integer, FM039 if floating point.

The user, however, is given the opportunity to label the variable as he wishes, thus cancelling the system name. He therefore either responds to this message by typing in the appropriate name or else types N, signaling approval of the system name. The user's name must be limited to eight characters.

If during the derivation TRACE has discovered missing values in the data necessary for this particular derivation, an additional line is printed. This line will <u>precede</u> the current message and will have the following form:

```
n MISSING VALUES FOUND  FN.01
```

where n is the actual number of missing values found during the derivation. Missing values affect a derivation only in producing invalid or zero values for the variable being derived. Some of these may show up as rejected values in a displayed table later.

The last message from the subprogram is:

```
DERIVATION COMPLETE  2.22
 (D)ERIVE OTHER MEASURES
 (B)ASIC OPTIONS
*
```

The derivation is complete. You are given the option of either returning to the beginning of the routine in order to ask for further derivations or else selecting another subprogram.

--The Components of the Derivation Statement--

The basic form of the derivation statement is:

(result type) = (derivation expression) //

(This statement always ends with a double slash.)

The result type consists of one of the following:

AM alphanumeric
IM integer
FM floating point

The particular result type used governs the entire derivation: the variable derived from it will be of that type no matter what variable types are contained in the derivation statement.

Following the appropriate result type is an equals sign, both preceded and succeeded by at least one space. Then follows the body of the derivation statement. This may consist of the following components:

1. variables
2. constants
3. arithmetic formulas
   operators
   parentheses
4. functional operators
5. relational operators
6. conditional connectives
7. conjunctions
8. statistical operators
9. selective operators
10. manipulative operators
11. bracketing
12. string variable operators

(1) variables

These may be either input or derived variables.

(2) constants

These may be either numerical or alphanumeric, for example: 122 3.9 B

(3) arithmetic formulas

Arithmetic formulas make use of arithmetic operators and parentheses. The operators are the following:

+	addition	*	multiplication
-	subtraction	/	division

Parentheses are used to enclose certain portions of an arithmetic formula when the user wishes a particular order of operations, even if the system would follow that order were the parentheses omitted. Finding no parentheses, TRACE attacks an arithmetic formula from left to right, recognizing division and multiplication before addition and subtraction. In other words, it starts looking at the various operations from the left; when it meets either a division or a multiplication, it performs that operation and saves the result; it continues this course to the end of the string of operations (the formula); then it returns to the beginning of the formula and recognizes the addition and subtraction operators. For example, in the formula:

```
2 - X / 3 * Z
```

the first operation is that of dividing the value of the variable X by 3; then this result is multiplied by the value for Z; finally this result is subtracted from 2. To request this procedure by using parentheses, the user should type:

```
2 - ( ( X / 3 ) * Z )
```

In this case, the parentheses would be superfluous; however, they are important if the user wanted another disposition of the formula. For example, here are a few examples of what parentheses can do to this formula:

```
( ( 2 - X ) / 3 ) * Z     ( 2 - ( X / 3 ) ) * Z     ( 2 - X ) / ( 3 * Z )
```

Note that every arithmetic operator and parenthesis in a formula is an entity in itself within the derivation statement. Therefore, each such element must be separated from a neighboring element by at least one space. In other words, a construction such as:

```
(X/3)
```

will be treated as a variable name consisting of five characters.

(4) functional operators

These consist of the following:

SIN	sine
COS	cosine
TAN	tangent
ARS	arcsine
ARC	arccosine
ART	arctangent
SQR	square root
ABS	absolute value
LOG m BSE n	logarithm of m to the base of n (n, as well as m, may be either a constant or a variable name)
EXP	exponentiation

The user may employ either a variable or a constant following any of these operators; he may actually use any legal computational expression, for example:

```
TAN ALPHA
SQR ( 1 - ALPHA / 2 )
LOG 39 BSE 7
ALPHA EXP 4
ABS ( ARC ALPHA + LOG BETA BSE 2 )
```

Note: the value following any of the trigonometric function operators, e.g., SIN, will be treated as a radian value.

(5) relational operators

There are six of these operators:

EQ	equal to
NQ	not equal to
LS	less than
LQ	less than or equal to
GR	greater than
GQ	greater than or equal to

These are used to compare one computational expression with another, for example:

```
DISTRICT EQ 34
DISTRICT + ALPHA NQ FM003 - 88
1 LS ALPHA * ( BETA / GAMMA ) + DISTRICT - SQR PHI
```

(6) conditional connectives

There are two of these connectives: IF and OTR. Each connects two sections (clauses) of a derivation statement. The IF is used if the section following it is a condition that must be met before the section preceding the IF is to be carried out. The IF therefore literally means "if": do section 1 if the conditions in section 2 are met. The primitive OTR, on the other hand, gives the system an alternative task to carry out if the conditions in an IF section are not met; it necessarily either succeeds an IF section or does not exist at all. For example:

```
FM = ALPHA + 1 IF SCHOOL EQ 22 OTR ALPHA + 2 //
```

The result is the value of ALPHA plus one if the value of SCHOOL is equal to 22; if not, the result is the value of ALPHA plus two. It is naturally not mandatory to place an OTR section following every IF section. In case the conditions of such an IF section (that is, one without a following OTR clause) are not met for a particular case index value, no value at all is derived.

Note: an OTR clause cannot contain any entity other than a computational expression, such as ALPHA + 2, SIN ALPHA, or X. You may not include a

conjunction, or a statistical operator, and further conditional clauses are forbidden. An OTR clause may, moreover, not be appended to any derivation statement that contains a statistical operator.

A derivation statement may contain only one IF connective and therefore only one OTR connective.

The subject of conditional connectives is expanded in Part 5 of the dialogue section.

(7) conjunctions

There are two conjunctions: AND and OR. These mean literally what they say: the AND connects two sections of a derivation statement that are to be treated as a pair; the OR connects two sections that are to be treated as alternatives to each other. For example, the following IF clause (section):

```
IF DISTRICT EQ 34 AND SCHOOL EQ 33
```

means that both conditions must be met if the section preceding this clause is to be carried out. Contrarily, the clause:

```
IF DISTRICT EQ 34 OR SCHOOL EQ 33
```

means that either the first or the second condition may be met to insure the carrying out of the preceding section.

The conjunctions AND and OR can only be used to connect separate phrases (conditions) of an IF clause; they cannot be used to connect different variables in the same phrase. For example, in the statement:

```
FM = SQR A + TAN B IF C GR D AND E LS F OTR SIN G //
```

the AND connects the two conditions: C GR D and E LS F. It does not act to connect D with E; that is, the following statement is invalid:

```
FM = SQR A IF C GR D AND E //
```

AND must be followed by a condition, for example: C GR E.

Whenever both an AND and an OR are to be used in the same statement, the AND phrase becomes subservient to the OR; that is, an AND phrase always belongs to any OR preceding it. For example, in the statement:

```
FM = SUM A IF B GR C OR D GR E AND F GR G //
```

the AND phrase (F GR G) is part of the condition started by the OR. The statement reads as follows:

```
calculate the sum of A if B is greater than C
  if not, calculate this sum if both D is greater than E and F greater
  than G
    if neither condition is satisfied, do not compute any result for the
    current case index value
```

This subject is examined in greater detail in Part 5 of the dialogue section.

(8) statistical operators

There are four of these operators: NBR, SUM, MEN, and MDN. Only one statistical operator may be used per derivation statement and, when used, it must be the first component.

1. NBR — counts the number of occurrences of a variable under specified restrictions and conditions

For example, the statement:

```
FM = NBR SCHOOL IF DISTRICT EQ 34 //
```

asks the system to compute the number of schools in district 34.

2. SUM — computes the sum of the values of a variable

For example, the statement:

```
FM = SUM SCHOOL IF DISTRICT EQ 34 //
```

asks the system to compute the sum of all the values for SCHOOL that correspond to the condition of DISTRICT being 34. If the values for SCHOOL happen to be merely identification numbers, this statement is fairly worthless; however, if the values represent something <u>about</u> the variable SCHOOL--like the number of students--then this statement becomes meaningful.

3. MEN computes the mean (arithmetic average) of the values of a variable

For example, the statement:

FM = MEN ALPHA //

asks the system to compute the mean of all the values for ALPHA, considering the case index and restrictions that are outstanding.

4. MDN computes the median of the values of a variable

For example, the statement:

FM = MDN ALPHA IF ALPHA GR 42 //

asks the system to compute the median of all those values for ALPHA that are greater than 42.

A statistical operator may act upon either an input variable or a derived variable.

(9) selective operators

There are five selective operators: EACH, FIRST, LAST, ALL, and ONE. Only one may appear in a statement and that appearance must be as the final component (before the double slash). These operators instruct TRACE how to work with the case index table (Section 1.6) during a derivation.

EACH This operator tells TRACE to work with each occurrence of a case index value in the case index table, that is, to make a computation for every block of data that passes restrictions and conditions. When this operator appears in a statement headed by a statistical operator, the computation is repeated so that the final results are in terms of unique values of the case index variable.

FIRST This operator tells TRACE to work only with the first occurrences of a case index value in the table.

LAST — This operator tells TRACE to work only with the last occurrence of a case index value in the table.

ALL — This operator tells TRACE to conduct the specified derivation as though there were only one occurrence of the case index value in the entire data base; that is, all the results are boiled down to one result--sums are turned into a grand sum, means are converted into one grand mean, and so on.

These grand derivations, however, cannot be used as derived variables. They are not even stored by TRACE but are immediately released via teletype to the user. After the result has been typed, the system types out the first message of the derivation program. Note: ALL may be used only after statistical operators, for example:

FM = MEN GRADE ALL //

ONE — This operator, used only when the derivation statement contains a conditional clause, tells TRACE to work only with the first occurrence of a case index value that satisfies the condition in the statement.

A derivation statement that does not end with a selective operator is treated as though it held the EACH operator.

The use of these operators and the case index tables referred to are examined exhaustively in the dialogue section, Part 6.

(10) manipulative operators

There are two operators that serve to manipulate the values of derived variables: CMB and MRG. These do not operate on input variables. When used in a statement, the components in that statement are limited to a series of derived variable names and the operators themselves; that is, there is no opportunity to use conjunctions, selective operators, arithmetic formulas, etc. Like other derivation statements, however, they must be preceded by case index and restriction statements.

CMB — This operator tells the system to tack onto the values of one variable all the values of one or more variables so that a new variable arises whose storehouse of values is the total of all the values of the variables mentioned in the statement.

The form of the statement using CMB is:

(result type) = CMB (variable list) //

where (variable list) is two or more variable names. For example, the following is a valid CMB statement:

AM = CMB ALPHA BETA GAMMA //

The system now gives over to the new alphanumeric variable all the values of the three specified variables. These values appear in the new variable in the same order as their original variable names appeared in the statement. For example, suppose ALPHA held three values: A, B and C; BETA four values: MN, OP, QR and ST; GAMMA two values: G1, G2. The new variable would consist of: A, B, D, MN, OP, QR, ST, G1 and G2.

Note: all the variables named in a CMB statement must have the same case index (as in any derivation statement). The restriction statement preceding this statement is not recognized by the program, so that the CMB statement cannot be bound by any restrictions or conditions. The restriction statement therefore may consist simply of the response: N.

MRG	This operator, functioning only with alphanumeric derived variables, takes each value of one, tacks onto it the corresponding value of another (according to ascending order), and thereby creates a new value consisting of the two original values. It repeats this process for each pair of values in the two variables. These two variables therefore must have the same case index.

The form of the statement using MRG is:

(result type) = (variable) MRG (variable) //

For example, the following is a valid MRG statement:

AM = ALPHA MRG BETA //

Suppose the values for ALPHA were A, B, C, D, E, F ... and the values for BETA were B1, B2, B3, B4, B5, B6, ... The new variable would consist of the values: AB1, BB2, CB3, DB4, EB5, FB6, ...

Unlike the CMB derivation, a MRG derivation recognizes the restrictions in the associated restriction statement.

Bracketing is allowed for all components of either type of statement.

The use of a CMB statement is illustrated in Section 1.7; the MRG statement appears in Section 1.9.

(11) bracketing

Any component in either a restriction statement or a derivation statement may be parameterized by means of substituting for that component a bracketed element. Bracketing allows the user to specify one at a time the values for a particular component and thereby run several derivations on the same derivation or restriction statement framework. For example, the user may bracket a variable in the derivation statement and then repeat the derivation for several different variables. This device then saves the user the need to compose many derivation statements, together with the associated case index and restriction statements, for runs that differ only in the identity of a few components.

The user is allowed to bracket as many components in a single derivation statement as he wishes. There are no exceptions to parameterization: conjunctions, statistical operators, even parentheses may be bracketed. There are two basic forms of bracketing: single left bracket or double brackets enclosing an alphanumeric string of six or fewer characters. These two types may be mixed in a single statement. Examples:

```
FM = MEN [A] IF [B] GR [C] //
FM = SUM ALPHA IF ( [A] / [B] ) / ( [D] + [E] ) //
FM = [ [ [ [ [ //
IM = [XYZ] //
```

Note: there must not be a space between a bracket and an enclosed string.

Whenever TRACE encounters bracketing within restriction or derivation statements, it requests values for these components immediately after the user has entered the derivation statement. The request is for the total number of parameterized components in the two statements; the user must then type in a response that contains the missing values, in the order of their substitution. For example, if one component in the restriction statement were

bracketed and two in the derivation statement, a message flow would resemble:

```
GIVE 3 PARAMETERS
 ( )VALUES
 (S)TART
* 34 ALPHA EACH
```

If the user types in the missing values at this point, TRACE assigns a system name to the variable (message 2.21), giving him the option of assigning his own name. The system then returns the same request to the user, allowing him to designate another run.

If the user types in the missing parameters at this point, TRACE issues message 2.21, which assigns a name to the newly derived variable with a provision for accepting a user-assigned name. It then repeats the request for parameters, allowing the user to designate another run on the same derivation framework. To stop this cycle, the user must either type S to send the program back to the first message of the derivation routine, or else a GOTO to another subprogram.

This device of parameterizing is illustrated extensively in Section 1.5. Another use is shown in Section 1.7.

(12) string variable operators

With one exception, whenever a string variable may be referred to in any TRACE statement, so may any subdivision of it be referred to separately. This interior reference is done by following the name of the string variable with digits that denote the subdivisions. For example, the third subdivision of ALPHA would be denoted by:

```
ALPHA 3
```

The user may refer to series of consecutively appearing subdivisions by typing in the beginning and ending members separated by a dash:

```
ALPHA 3-7
```

Note that there is no space between either digit and the dash.

The one exception to this rule is in the matter of restriction statements. Here the string variable may be used only as an entire value; subdivisions are not recognized.

Warning: when working with string variables, remember that subdivisions can be referred to only singly, for example:

```
ALPHA 3
```

or in a series:

```
ALPHA 3-7
```

It is not possible to refer to separated subdivisions in a variable, for example:

```
ALPHA 3 5 7
```

There are two operators to be used in conjunction with string variables in a derivation statement: SCAN ACROSS and COR.

SCAN ACROSS	Appearing just before a string variable in a derivation statement, it tells the system to treat each of the subdivisions of that variable as a separate value. When this operator is absent, the system treats the whole string of subdivisions as one value.

For example, in the following statement:

```
IM = SUM SCAN ACROSS ALPHA 1-9 EACH //
```

each of the nine subdivisions specified will be summed for each occurrence of the variable ALPHA in the data base. If the statement read:

```
IM = SUM ALPHA 1-9 EACH //
```

the nine subdivisions in each occurrence of the variable would be treated as one value and the summing done accordingly.

COR — This operator serves to set up a correspondence between the subdivisions of two or more string variables that appear together in a derivation statement. It appears before the variable name, followed by zero, an increment such as +1, or a decrement such as -5.

For example, if a derivation statement held two string variables: ALPHA and BETA, these variable names preceded respectively by COR Ø and COR +7 would tell the system to compare the first subdivision of ALPHA with the seventh of BETA, the second of ALPHA with the eighth of BETA, and so forth.

The concept of string variables is first touched upon in the dialogue section during Part 6, when the term SCAN ACROSS is introduced. The term COR is introduced in Part 7.

--The Derivation Statement as a Desk Calculator--

The user has in this subprogram a feature that simulates a desk calculator. With this device, the user can devise a string of computations that has no relation to any part of his data base, nor to any derivation. He simply enters the derivation subprogram, jumps to the derivation statement message (via a GOTO) and types in the calculation. This statement has the form:

COMP = (calculation expression) //

The calculation expression may contain any constant, arithmetic operator, parentheses, functional operator--and it permits bracketing. None of the other derivation components may be used, for example, variables and statistical operators. The result of the calculation is immediately presented to the user via teletype from the system. The system then returns message 2.22 indicating that the derivation is complete and requesting that he either start another derivation or else exit to a different subprogram.

Examples:

COMP = SQR (SIN 3.335 - COS 4.595) //

COMP = LOG [BSE [+ [G] //

There is an example of the use of this statement in Part 7 of the dialogue section.

PART 3

The Construct Table Subprogram

This routine is used to construct tables that are to be displayed by the display subprogram. When two-dimensional, these tables act as frequency tables for a pair of variables derived on the same case index. This use of the subprogram is thoroughly illustrated in Problem 5 of the dialogue sectior (Section 1.8). As one-dimensional tables, they serve to yield individual or grouped values of a derived variable. This use is described in Parts 5 and 7 of the dialogue section.

The first message in the construct table program is:

```
CONSTRUCT TABLE  3.0
 (S)TEP BY STEP
 ( )SHORTHAND
 (L)IST SET UP TABLES
*
```

The user is here given the choice of either going through the steps of constructing a table one by one, that is, of accepting a train of messages from the routine with concomitant responses, or of typing in one line of information that in itself is all that the system needs in preparing a table. (He may also choose instead to have previously constructed tables listed.) Choosing and following the shorthand method immediately closes the work of the routine and the user proceeds to the display program after this response. The structure of the shorthand line of information is described at the end of this subsection.

--Step-By-Step Method of Table Construction--

Within this step-by-step method, there are two routes: one for variables that are alphanumeric, the other for those that are either integer or floating point. For two-dimensional tables, either of these routes may be traversed twice during the same table construction or both may be crossed since the same sequences of messages are used for both the description of the row variable and the column variable.

The first sequence of messages describes the row variable:

```
SPECIFY ROW VARIABLE  3.1
 ( )NAME
 (N)ONE
*
```

Two variables are required for a two-dimensional table. One, called a row variable, comprises the values that describe the rows of the table, that is, one value or an interval (or group) of values to a row. The headings for row variables then are to be found to the left of the displayed table--in the first column. Contrarily, a column variable describes the columns of a table and its headings form a row that runs along the top of the table. If the table is one-dimensional, consisting of a single column, it has a row variable and no column variable. The reverse is true for a single row table. For this message then, you respond with the name of the variable you wish to describe the rows of your table. This variable is usually the one (if the table is two-dimensional) that has the greater number of values or intervals.

If the current table has no row variable (response here of N), the system jumps to message 3.2.

(1) Route for alphanumeric variables

This route consists of two messages, one asking for the number of cells to be set up for the particular variable, the other requesting a heading name (tag) for each cell according to row or column.

```
     NUMBER OF ROW (or COL) CLASSES  3.32
      ( ) 1 TO 10 FOR COL  1 TO 40 FOR ROW IF DISPLAYED ON TTY
     *
```

This message asks the user to specify the number of row cells or column cells to be constructed for the variable in question. The first line of the message contains the word ROW for a row variable run, COL for a column variable run. The second line reminds the user that the limit for columns on teletype display is ten, for rows, 40. (CRT displays can accommodate only 25 rows.)

```
     CLASS TAGS AND INCLUDED VALUES  3.321
      ( )TAG = ( )VALUES END WITH //
     *
```

This message, issued in the same form for both types of runs, asks for the headings of the specified cells together with a list of the variable values to be included under each heading. For example, suppose your row variable were ALPHA, consisting of values A through Z. If you wished the first row of cells to represent the values: A, B, C, and D; the second: E, F, G, and H, and so forth--the first row to be called FIRST, the second SECOND, and so forth, you would type the following:

```
     * FIRST = A B C D SECOND = E F G H ..... //
```

(2) Route for integer and floating point variables

This route consists of three messages. They are identical in both types of runs.

```
      NUMBER OF INTERVALS  3.31
       ( ) 1 TO 10 FOR COL  1 TO 40 FOR ROW IF DISPLAYED ON TTY
      *
```

This message asks for the number of intervals to be constructed for the variable in question. These will appear as cells in the displayed table, a limit of ten for columns, 40 for rows on teletype or 25 for rows on CRT (cathode ray tube). Each interval represents a particular value of the variable or a range of values.

```
      RESPOND WITH:  3.311
       (N) FOR EQUAL NUMBER
       (W) FOR EQUAL WIDTH
       (S) FOR SPECIFIED WIDTH
       (A) FOR ALL VALUES
      *
```

You type N if you wish the intervals to represent as nearly equal numbers of case index values (frequencies) as possible, W if you want them to be divided into equal widths (ranges); both responses indicate that you desire the system to set up the intervals. The third response declares that you wish to specify the precise boundaries of each interval. The fourth response is used when you desire every available value of the variable to be shown as a heading value, that is, each value is a whole interval by itself.

Note: this message is not received when the variable is an alphanumeric one since it is not possible to have intervals of alphanumeric values.

```
      ( )GIVE LOWER AND UPPER LIMITS FOR EACH INTERVAL

      *


This message is issued only when the user has indicated in his response to the
previous message that the intervals are to be specified by him and if the row
variable has either integer or floating point values.  The user then types in
the lowest limit followed by the first upper limit followed by the next lower
limit--and so on until the highest limit is reached.  For example:


      * 1 25 30 45 50 75 80 100 //
```

The second sequence of messages describes the column variable:

```
      SPECIFY COL VARIABLE  3.2
       ( )NAME
       (N)ONE
      *
```

This message is similar to message 3.1 (SPECIFY ROW VARIABLE). If there is to be a column variable, the same sequence of messages that succeeds message 3.1 will be sent. The user should respond to them in like manner. If there is no column variable, the system jumps to message 3.4.

Upon ending this second cycle, the system delivers the following message:

```
      MODIFY OR NAME TABLE  3.4
       (R)OW
       (C)OLUMN
       ( )TABLE NAME
      *
```

This is the last message to be sent in the step-by-step method of constructing a table. It allows the user to change either some row variable specification or some column variable; if no changes are necessary, you are obliged to name the table. The first response returns you to message 3.1--SPECIFY ROW VARIABLE, the second to message 3.2--SPECIFY COLUMN VARIABLE.

The final message of the subprogram, the following is issued after either method of constructing a table has been completed:

```
ROW IS  row variable name   CODED  type of interval
COL IS  column variable name   CODED  type of interval
CASE INDEX IS  case index
TABLE   table name   SPECIFIED  3.6
 (D)ISPLAY THIS TABLE
 (C)ONSTRUCT OTHER TABLE
 (B)ASIC OPTIONS
*
```

The first four lines act as a resume of the table construction. They might appear as follows:

```
ROW IS SCHOOL  CODED I
COL IS STUDENT  CODED A
CASE INDEX IS DISTRICT
TABLE  GEOGR  SPECIFIED  3.6
```

The last three lines of the message are response forms: the first leads to the display program, the second back to the beginning of the table construct program, the third to another subprogram.

--Shorthand Method of Table Construction--

The user can circumvent all of the communication between himself and the system involved in the step-by-step method by entering one line of information that specifies all the necessary directions in constructing his table. This line is to be typed in response to the first message of the subprogram--message 3.0. This shorthand line has the following components, typed from left to right (spaces between elements):

(1) the name of the table

(2) the letter R, if there is a row variable--otherwise, skip to component (6)

(3) the name of the row variable

(4) the number of intervals for the row variable

(5) the type of interval--use the same codes as described in message 3.311, that is, N, W, A, or S.

(6) the letter C, if there is a column variable--otherwise, the line is ended

(7) the name of the column variable

(8) the number of intervals for the column variable

(9) the type of interval--same codes as stated in (5)

If the code typed in at either point (5) or (9) is S--for specified intervals--the subprogram sends the appropriate messages for more information about these intervals, similar to the treatment of specified intervals in the step-by-step method. The shorthand method is then concluded by message 3.6, which is a resume of the table just constructed and which also gives the user several options about continuing table construction or going to another subprogram.

PART 4

The Display Subprogram

This subprcgram is a natural partner to the table construct program; its function is that of displaying tables or working with previously displayed tables. In the dialogue section, this subprogram is discussed in Part 8.

This discussion closes with a sample display of a two-dimensional table followed by an explanation of the various features of that display. If the user wishes, he may refer to that sample table and discussion now.

The first message of the display program:

```
DISPLAY  4.0
 ( )NAME TABLE
 (L)IST CONSTRUCTED TABLES
*
```

The user can respond by typing either the name of the table that he wishes displayed or the letter L, which gains him a listing of all the tables thus far constructed. This listing consists of the table name, the row and column variable names, the number of intervals for these variables, and the types of intervals.

```
DISPLAY ON  4.1
 (S)COPE
 (T)TY
*
```

The user types S if the display is to take place on the cathode ray tube, T if on the teletype.

The program now displays the desired table. Following this display, it prints the message:

```
      COMPLETE  4.2
       (W)ORK WITH DISPLAY
       ( )NAME DISPLAY
          (R)  ( )NAME NEW ROW
          (C)  ( )NAME NEW COL
       (B)ASIC OPTIONS
      *
```

You type the first response if you wish to work with the display, for example, identifying the case index values responsible for a particular frequency number in the table. This response brings you to message 4.3.

The second response involves a partial modification of the table just displayed or of any other table already constructed. Here you may change either or both the identities of the row and column variables, thus producing a new table. However, all other features of the table will remain the same, such as the number of intervals, interval limits and types. To request this new table, first type the name of the original table (the table whose framework is to be used for the new table), then the letter R if a row variable is to be changed, then the name of that row variable, then the letter C if a column variable is to be changed, followed by the new column variable name. For example:

```
* QUAD R ABSCISSA C MANTISSA
```

The system will then return you to message 4.1 where you may choose the scope or the teletype for display of the new table. After this is done, the system displays the table and reissues message 4.2.

If you have no more work for the display program at this point, select another subprogram by typing the letter B followed by the appropriate code.

```
WORK OPTIONS END WITH //  4.3
 (L)IST INTERVAL LIMITS
    (R)OW  OR  (C)OL
 (E)XPECTED VALUES
 (A)SSIGN VALUES TO CELLS
    ROW# - COL# = VALUE
 (D)ENTIFY CELLS
    ROW# - COL#
*
```

The user is given four options in working with the current display. If he has selected option 1, 2 or 4, the program returns him to message 4.2 after the option selected has been completed. The third option is completed by a return to the beginning of the subprogram.

(1) He can have the interval limits listed for review purposes--both upper and lower limits for each interval will be listed. A request consists of the letter L followed by the letter R (for row variable) or by the letter C (for column variable).

(2) There is a normal expectancy of frequency values (distribution) that TRACE computes for each display in determining Chi-square (one of the products of a display). If the user wishes to see the table displayed with the expected frequencies shown rather than the observed frequencies, he types the letter E. The new display will then immediately be produced.

Note: since Chi-square is a product of two-dimensional tables, this option has no meaning with respect to one-dimensional tables.

(3) The user can create a new variable (derived) by assigning values to the members of various cells of his display. The number of occurrences then of this new variable having any particular value will depend upon the frequencies in the cells, that is, the number of case index occurrences shown by the cells. To do this, the user designates one or more cells, and then assigns a value to the components. The system will then ask him to name that variable. To illustrate this, look at a display consisting of nine cells--three rows and three columns.

The frequencies are as follows:

4 15 9
6 18 6
8 24 4

To create a variable using these frequencies, the user's statement may resemble the following:

*A 1-2 = 1 2-2 = 2 3-2 = 3 E-2 = 4 OTR Ø //

This statement says the following:

assign the value 1 to all cases in Row 1 - Col 2; assign the value 2 to all cases in Row 2 - Col 2; assign the value 3 to all cases in Row 3 - Col 2; assign the value 4 to all the rejected cases for the second column; assign the value Ø for all other cases.

Note that each element in the assignment process has the following form:

(row number)-(col number) = (value)

There is no space between either number and the dash. For rejects, the letter E is used: E-n for column rejects, n-E for row rejects, E-E for rejects that fit none of the columns or rows. The conditional connective OTR followed by a value tells the system to assign that value to all the cases not covered by the other elements in the statement--it does not include rejects, however.

The resulting variable would now consist of the following (assume one reject):

15 occurrences of the value 1 18 occurrences of the value 2

24 occurrences of the value 3

The new variable is 95 occurrences in length.

The system then asks the user to name his variable:

```
 variable name IS THE CASE INDEX  4.4
GIVE:
 ( )NAME  ( )CODE
*
```

The system tells the user what the case index variable for the display was. You are now to type in an appropriate name and the type: A, I or F (alphanumeric, integer, floating point).

The system then completes the communication by returning

```
DERIVATION COMPLETED
DISPLAY  4.0
 ( )NAME TABLE
 (L)IST CONSTRUCTED TABLES
*
```

(4) The actual case index values tabulated in any one cell in the current display can be identified. TRACE supplies a listing of these values after the user types the row and column identification numbers. For example:

```
* I 1-3 //
```

This tells TRACE to list all the case index values for the cell of row 1 by column 3. Note that there is no space between the dash and the numbers.

Rejected cases can also be displayed, for example:

```
* I E-2 3-E E-E //
```

This asks for a display of rejected cases for the second column, the third row, and the cases that pertain to no displayed column or row.

The following is a display of a typical two-dimensional table:

	1.ØØØ 2.ØØØ	5.ØØØ 7.ØØØ	TOTAL	%
Ø.ØØØ	REJECT			
Ø.ØØØ	4	7 13 1	24	16
1.ØØØ	1	3 4 Ø	8	5
2.ØØØ	48	8 15 Ø	71	47
3.ØØØ	8	16 25 Ø	49	32
REJECT	Ø	Ø Ø 2		
TOTAL	61	34 57	152	
%	4Ø	22 38	155	

ROW STAT			
MEAN =	1.954		
MEDIAN=	2.ØØØ		
STNDVN=	1.ØØ2		
COL STAT			
MEAN =	3.888		
MEDIAN=	4.ØØØ		
STNDVN=	2.413		
CHISQR=	42.2Ø67	R CORL	Ø.Ø511Ø
DEG-FRM	6	R LO-5	-Ø.15936
EX LS 1	Ø	R HI-5	Ø.159Ø4
EX LS 5	3	R SQR	Ø.ØØ261
C CNTG	Ø.466	T STAT	-9.26549
SIG .ØØ1		F STAT	5.79717

Above the table proper (i.e., above the dotted line) are shown the column variable limits: the lowest limit: 1.ØØØ, and the upper limits: 2.ØØØ, 5.ØØØ, and 7.ØØØ. The row variable limits are shown to the left of the table

(in this example, the intervals are absolute values): Ø.ØØØ, 1.ØØØ, 2.ØØØ, 3.ØØØ, and 4.ØØØ. Note that the first row variable heading is repeated above the dotted line.

Whenever a case index value meets both a row interval and a column interval, it is counted in the cell formed by the intersection of those two intervals in the table. If a case index value meets only a row interval and none of the available column intervals, it is placed in the cell at the end of that row as a rejected value--each row is terminated by a cell for rejected values. Similarly, a value meeting a column interval only is placed in the reject cell at the end of that column. A case that meets none of the available intervals at all is placed in the cell formed by the intersection of the column reject cells and the row reject cells. In our example, this cell shows the value 2--two cases met none of the intervals. To refer to these reject cells in responses, the user types E-n for row rejects (e.g., E-5), n-E for column rejects, and E-E for complete rejects.

Following the reject cells in the table are the subtotal cells: these exhibit the total cases for their respective rows and columns. Notice that these cells have no intersection.

The last sets of cells are the percentage cells. These show the percent of the associated subtotal to the grand total of cases, excluding rejects. In the sample table display, row 1 shows one reject, a subtotal of 24 valid cases, and a subtotal percent of 16%. Comparative figures for the first column are: Ø rejects, 61 cases, and 40%.

At the lower right hand corner of the table appear two grand totals: the value 152 represents the total of all valid cases displayed by the cells; the value 155 represents the total of all cases encountered in constructing the cell, including the rejects.

Below the table are printed various statistical products of the displayed table. These products can be examined in the following groupings:

(1) ROW STAT

MEAN =

MEDIAN =

STNDVN =

These are statistics for the row variable, drawn from the values that make up that derived variable. Shown are the mean of all the values (except rejects) of the row variable, the median for these values, and the standard deviation.

These statistics are produced only for numeric (i.e., integer or floating point) variables, whether the table is one-dimensional or two-dimensional.

(2) COL STAT

MEAN =

MEDIAN =

STNDVN =

This group of statistics is produced for a numeric column variable. The table may be either one- or two-dimensional.

(3) CHISQR =

DEG-FRM

EX LS 1

EX LS 5

C CNTG

SIG .ØØ1

This group is yielded with all two-dimensional tables, regardless of the nature of the row and column variables. It is not produced for one-dimensional tables. These statistics are, in order:

CHI SQR =	the chi-square value. Fisher's exact probability is computed if the table reduced to a 2 x 2 matrix has any cells with expected values less than 5.
DEG-FRM	the degrees of freedom
EX LS 1	the number of cells with expected values less than 1
EX LS 5	the number of cells with expected values less than 5
C CNTG	the contingency coefficient

(4) R CORL
R LO-5
R HI-5
R SQR
T STAT
F STAT

These statistics are presented only for those two-dimensional tables wherein both row and column variables are numeric. They are described as follows:

R CORL	Pierson's R correlation coefficient with
R LO-5	5% confidence non-symmetric interval
R HI-5	limits given
R SQR	Pierson's coefficient squared
T STAT	t statistic
F STAT	F statistic

PART 5

The Edit Subprogram

The edit subprogram is a complex of six individual functions, each called separately by the user and each being independent of the other functions. Therefore, once a function is finished with by the user, he must use a GOTO command to call another function, there being no sequence of functions, or to exit from the subprogram altogether. The six functions, together with their calling codes are given the user when he first enters the edit program:

```
EDIT 5.0
 (D)ELETE
 (L)IST
 (C)HANGE
 (F)ORM
 (R)ENAME
 (A)SSIGN BY REPLACEMENT
 (B)ASIC OPTIONS
*
```

Every response to an edit routine message must end with a double slash! This device in other subroutines is generally confined to those responses whose corresponding system messages carried within themselves a command about the double slash, namely:

```
END WITH //
```

In edit, a message will not normally advertise this need for a //. This device is to be used even though a response could not possibly travel into another line.

--The Delete Function--

DELETE 5.1
(M) ( )MEASURE NAMES
( )DATA SET NUMBER
(D) ( )DISPLAY NAMES
( )NAME BLK-BLK
DATA SET NUMBER BLK-BLK
*

This function undertakes to delete for the user various portions of data, either input or derived. You have only one choice of response during each entry into this function, however--to request another type of deletion means returning back to this message via a GOTO command.

(M) ( )MEASURE NAMES

The first choice of responses allows you to delete any number of derived variables (measures) from your set of derived data. All the values of the variables listed by you will be eliminated. You simply type the letter M followed by the variable names, for example:

```
* M IM005 AM034 STD# //
```

( )DATA SET NUMBER

The second response asks the system to delete an entire data set (all the data entered via one tape, or input run if disc or teletype). The response consists of the appropriate data set number. Only one data set number may be entered at a time. These numbers are assigned to the data sets in chronological order and can be viewed, if desired, by requesting a listing of variables at the very first message of the derivation subprogram (message 2.0). Since only one data set can be eliminated at a single time, you must call this function a second time in order to request another data set deletion.

(D) ( )DISPLAY NAMES

You may delete one or more tables that have already been constructed for display by typing in the letter D followed by the appropriate table names. Example:

```
* D TABLE1 TABLE2 TABLE3 //
```

( )NAME BLK-BLK

You can delete specific blocks of derived variable data by typing the name of the variable concerned, followed by the beginning and ending block numbers of the values in that variable you wish deleted. These values, of course, must occur consecutively in the data. More than one series of blocks may be entered for a variable, and more than one variable may be handled in one response. There can be only one series of blocks per line--if two or more lines refer to the same variable, the variable name must be repeated. Example:

```
*STD# 3 - 5
 STD# 7 - 7
 STD# 9 - 23
 IM001 4 - 4 //
```

Note: there must be a space between the dash and the block numbers. If the block numbers are not known, they can be learned by means of the listing function in this subprogram. (See message 5.2)

DATA SET NUMBER BLK-BLK

A more sweeping deletion of data blocks can be achieved by your listing them in terms of their appropriate data set numbers, rather than variable names. This type of deletion results in the elimination of <u>all</u> the variable values falling in the specified range. When you choose to do this, however, you can only refer to one string of blocks in any one response. For example:

```
* 2 83 - 99 //
```

This tells the system to delete blocks 83 through 99 of all the variable values in the second data set.

Note: no space may separate the dash from the numbers when specifying a string of consecutively appearing block numbers.

The first and third responses have alternatives: the user may ask that all the derived variables be deleted or that all the display tables be deleted. The following statements accomplish this:

```
*M ALL //                    *D ALL //
```

--The Listing Function--

```
LIST VALUES  5.2
 ( )NAMES
 (D)ATA BASE INDICES
 (FOR)  ( )NAME (=) ( )VALUE  (FIND) ( )NAMES
END WITH (//)
*
```

This function undertakes to furnish the user with listings of variable values and certain properties of those variables. It is this function that permits you to look at the original input values of your data set, or to discover the block numbers of those values. It is also the function that lists derived variable values. In some cases, then, it can be said to complement the work of the construct table-display subprogram partnership and even to duplicate it in a sense.

( )NAMES

The first response to this message is used to obtain a listing of values of a variable, either input or derived. The request for the listing will be in terms of block numbers. First of all, you are to type in the names of the variables whose values are to be listed. The system then responds with a second message:

```
TOTAL NUMBER OF BLOCKS = n
GIVE BOTH  5.21
 ( )STARTING BLOCK
 ( )NUMBER OF BLOCKS
*
```

The first line of this message informs the user of the total number of blocks involved. The message then asks you to type in both the starting block number and the total number of blocks desired for listing. For example:

```
TOTAL NUMBER OF BLOCKS = 56
GIVE BOTH  5.21
 ( )STARTING BLOCK
 ( )NUMBER OF BLOCKS
*3 47 //
```

The user has here been informed that his variable is contained in 56 blocks of data. He then asked the system to list 47 blocks starting with the third block.

If the user is interested in seeing the values of two or more input variables, he may name all of these variables in the initial response. The subsequent response, which concerns the beginning and ending block numbers, would then apply to all of these variables. On the other hand, if he wishes to see more than one derived variable listed, he shall have to work with them one at a time--the function treats only one derived variable at a time. The user may not request a listing of both a derived variable and an input variable concurrently. If the user specifies more than one input variable at a time, they should be in the same data set. If they also occur in other data sets, TRACE asks the user to select just one of those sets--only one set can be dealt with at a time. The user receives this message:

```
SELECT ONE OF THESE DATA SETS
m
n
o
p
q
*
```

where m, n. o, p, and q are set numbers. The user then types in an appropriate number.

The listing of variable values takes the following forms:

for input variables:

column 1 = block number
column 2 = variable value
.
.
.
column n = variable value

for derived variables:

column 1 = block number
column 2 = case index value
column 3 = variable value

After the listing, the system returns to the beginning of this same message, asking for the starting and total number of blocks. This is to help you in listing scattered sections of your data. If no more listing is required, you must type a GOTO command to another function or routine.

(D)ATA BASE INDICES

The second response to this function is a request to look at various properties of the current TRACE run. These are called data base indices. Having typed the letter D, the user receives the following type of listing:

DATA BASE INDICES 5.22

TYPE	NUMBER USED	MAXIMUM
DCT-VARB	52	353
DER-MEAS	3	181
TOTL-VRB	55	385
DATASETS	4	10
DISPLAYS	4	60
CLS/INTR	57	960
INCL-VAL	723	960
DTA-TRKS	20	23

Note: since the response in this case is only a letter code, not one containing names or variables, the user doesn't have to end it with a double slash. He simply types:

*D

The second column in this listing refers to the total number of items already used by the system; the third refers to the maximum set by the system--a value that cannot be controlled by the user (except for the last item--here "maximum" means a number currently allocated by the system. An option in the changing function can alter it). The items are as follows:

DCT-VARB	the total number of variables input by the user
DER-MEAS	the total number of variables (measures derived thus far)
TOTL-VRB	the total number of variables--both input and derived
DATASETS	the number of data sets input and still outstanding
DISPLAYS	the number of displays thus far
CLS/INTR	the number of classes or intervals
INCL-VAL	the total number of values included within these classes and intervals
DTA-TRKS	the total number of data tracks on disc

Note: the values presented in this listing are only the current ones. Deletions of data sets, tables, etc. will be reflected in this listing.

(FOR) ( )NAME (=) ( )VALUE (FIND) ( )NAMES

The third response to this function allows the user to look at variable values without knowing their block numbers. He does this by creating a subset of his data base with a request that resembles a restriction statement:

FOR variable = value FIND variable name //

for example:

FOR DISTRICT = 7 FIND STUDENT //

This response asks the system to list all the values for the variable STUDENT when the value of the corresponding occurrence of DISTRICT is 7. Just as in the restriction statement, the system makes a subset of the data wherein the

values for DISTRICT are each 7; this automatically makes a subset of the occurrences for STUDENT. The values for these occurrences are those that are listed. However, in edit, subsets are formed one data set at a time. The system therefore follows the user's response with another request, this one for the data set number:

```
INDICATE WHICH DATA SET  5.22
 ( ) 1 TO 10
 (N)ONE
*
```

The user then types in either a data set number if the variable whose values are to be listed is an input variable, the letter N if the variable is a derived one. After the desired listing has been presented, the system asks for another data set number:

```
COMPLETED
INDICATE WHICH DATA SET  5.22
 ( ) 1 TO 10
 (N)ONE
*
```

If you have no further work to do with this operation, you must exit by means of a GOTO command.

There are several rules that pertain to the use of this operation:

(1) The variable whose name appears to the left of the equals sign may only be an input variable, not a derived variable.

(2) The variables whose names appear to the right of the system word FIND may be input, derived, or a combination of both. The values for all these variables will appear in the same listing.

(3) The user may introduce more than one value to the right of the equals sign. In this case, the system works only with the values that actually exist in the data base.

For example, suppose we construct the following statement:

```
FOR DISTRICT = 5 7 9 - 11 FIND STUDENT ATTEND
```

If no data exist for DISTRICT equal to 5 or 7, only values for STUDENT and ATTEND corresponding to DISTRICT values of 9, 10 and 11 are listed. (There must be a space between the dash and either value when a string of consecutive values is indicated in the statement.)

--The Changing Function--

```
CHANGE  5.3
 ( )NAME = ( )VALUE ( )BLK#  (-)  ( )BLK#  EOM = //
 ( )NAME  (C/D) SET TO ( )NUMBER
 ( )NUMBER OF DATA BASE TRKS
*
```

This function undertakes to change actual values of variables for the user. It also can be used to change the number of characters in a string variable, thus converting a string variable to a non-string variable or vice versa, if desired.

( )NAME = ( )VALUE ( )BLK# (-) ( )BLK# EOM = //

The first response is used to change variable values. You must, however, first know the block numbers of the values designated for change--these, of course, can be gained from the listing function. In this response, you first type in the name of the variable, then the value that is to replace the original values, then the starting and ending block numbers for the values that are to be changed (these values must occur consecutively). If more values of the same variable or values of a different variable are to be changed, you may type them in the same format on following lines, ending only the last line with a double slash. (The word EOM means the same as END WITH.) For example:

```
STUDENT = 121 16 - 17
STUDENT = 121 19 - 19
MONTH = Ø 1 - 455 //
```

Here the user has asked that the 16th, 17th, and 19th blocks of the values for the variable called STUDENT be changed to a single value--121; he has also asked for the first 455 values of MONTH to be changed to Ø. Note: there must be at least one space between the dash and either block number in the response.

Warning: the new value must be of the same type as the old values; for example, an alphanumeric variable cannot have any of its values changed to integer values by means of this function. If the user wishes to change all values of a variable into another type, he should use the assign function (message 5.31). If he wishes to change just some of those values into another type, he must form a new derived variable from those values.

( )NAME (C/D) SET TO ( )NUMBER

The second response allows you to change the number of characters per subdivision in a string variable, that is, the number of characters for each distinct value, assuming that a string variable is defined as a variable that can consist of more than one value per occurrence. For example, the following statement:

GRADE C/D 2 //

asks the system to make each occurrence of the string variable GRADE contain two characters per subdivision; that is, each value now will be two characters long. Setting this variable to a non-string variable could be accomplisned by typing as the new C/D value, the total length of the variable per occurrence. It could also be done by setting it to zero.

( )NUMBER OF DATA BASE TRKS

The third response is used to change the number of data tracks on disc.

--The Data Set Forming Function--

FORM DATA SET 5.4

( )NAMES OF MEASURES

*

This function has only one purpose: it builds data sets from a designated group of derived variables, thus giving to the derived variables the status of input variables. The new data set formed has all the properties of an original input data set; for example, a variable previously treated as a derived variable and therefore unable to serve as a case index variable can now carry out that role. This type of conversion also serves to prevent the user's surpassing the system-determined limit for derived variables; that is, upon reaching this limit he may, instead of deleting some of his variables, convert them to data set variables, in other words, virtual input variables.

Note: all the variables designated must have the same case index variable.

Warning: if more than one variable is used for the forming of a new data set, they must all have exactly the same number of values. If not, you are expected to delete the overflow values, or else convert them into a different variable.

--The Variable Renaming Function--

RENAME 5.5

( )NAME (TO) ( )NAME

*

This function simply allows the user to rename any variable, either input or derived. You type the original name of the variable followed by the word TO followed by the desired new name, for example:

* STUDENT TO PUPIL //

All the values previously belonging to STUDENT will now be known under the name PUPIL.

You are allowed to specify more than one renaming in the same response. Simply type in a new renaming statement on a second line. The final line will end with a double slash.

If the variable name being used is found to be in more than data set, the system asks for a clarification, sending the following message:

FOR OLD (variable name) SELECT WHICH OR (A)LL OF THE FOLLOWING DATA SETS:

m

n

o

p

q

*

where m, n, o, p, and q are data set numbers. The user selects the desired data set and the system fulfills the request. It then returns the user to the beginning of the routine. If he wishes to rename every occurrence of the variable, he typed A. (The variable name typed by the system in the first line of this message is the original name of the variable.)

Notice how this function serves to complement the restriction statement in a derivation. By renaming a variable for one or more particular data sets, the user in effect restricts any derivation to just those data sets. After a derivation, the user may restore the original variable name, if he wishes.

--The Value Assigning Function--

This function is somewhat similar in effect to one of the options of the changing function: it allows the user to change actual values of a variable. However, there is one major difference. The changing function operated only to change designated individual blocks, or strings of blocks; this function permits the user to specify a value change that will affect every block or value of a particular variable. Like that function, however, the value change may not be from one type to another, for example, integer to alphanumeric.

The messages and responses for this operation have the following sequence:

```
ASSIGN VALUES  5.31
 ( ) NAME OF VARIABLE
*
```

Here you type in the name of the appropriate variable.

```
GIVE ASSIGNMENTS  END WITH //
 ( )OLD VALUE (=)  ( )NEW VALUE
*
```

You now proceed to type in the value that you wish changed followed by the value that will replace it. If you wish to make several value changes, each assignment must occupy a line by itself. For example, suppose the variable is SCHOOL, a non-string variable. By typing:

```
* 454 = 45
  457 = 45 //
```

you will have changed all values for SCHOOL that previously were either 454 or 457 into values of 45. On the other hand, if your variable were GRADE, a string variable, you would be permitted to change only one character per line in the response. TRACE allows only one character of a string variable to be changed at a time. To illustrate: suppose GRADE, a string variable

with two characters per subdivision, consisted of values such as A+, A-, B:, D+, and so on. To change these into numeric values such as 98, 92, 85, 68, etc., you would have to type lines such as:

```
*A = 9
+ = 8
B = 8
- = 2
D = 6
: = 5
```

and so on.

Another route lies open for the user who wishes to convert values in a string variable. Suppose that this user wishes to work with a string variable having three characters per subdivision, the possible characters being A, B, and C. Suppose next that he wanted to convert two possible occurrences to a different set of characters: the occurrences of ABB and CBA to XYZ and PQR. This conversion is impossible with the character by character method. First of all, that method involves <u>all</u> values in a variable, not just specific ones. Second, this conversion requires identical characters to be changed to different ones depending upon their original context. The user may solve this dilemma by converting the variable to a non-string variable by means of the option in the changing function of the edit program that does this conversion, changing the two occurrences in question three characters at a time, and then returning the status of the variable to that of a string variable.

Note: when the variable is alphanumeric, whether string or not, only <u>eight</u> characters are permitted to be changed at a time.

PART 6

The Output Subprogram

The output subprogram has one function: to allow the user to output a group of input and derived variable values in punched card format. It therefore is the program to call when variable values are to be output. There is one major set of restrictions on the variables that may be combined to produce one card image: derived variables must have the same case index variable; input variables must be of the same data set. If mixed, the input variables must have appeared in the same data set as the case index variable of the derived variables.

The first message in the output subprogram:

```
OUTPUT  6.0
GIVE CARD FORMAT
USE 0-9 AND A-Z
SEPARATE BY BLANKS
USE . FOR FLOATING POINT
LINE MUST END WITH A BLANK
*
```

This message asks the user to specify the number of columns to be devoted to each variable value in the card image by typing in a series of characters to represent the length of each variable. Each unique character will stand for a distinct variable and the number of those characters entered in a series denotes the length of the variable. Example:

```
* 55555555 7777 AAAAAA MM
```

(There must be a blank between the last character and the carriage return.) The character 5 stands for some one variable, as does 7, A, and M. There are eight 5's, so there will be eight columns allotted for that variable. Similarly, there will be four columns allotted for the variable represented by 7, six for A, and two for M. In case the user wishes to introduce spacing between variables in the card image, he need only specify more characters per variable than the actual length of the values in that variable. All values are right justified in the card image. So, if the user wished two spaces between the first two variables, he would represent the second variable by two more characters than would actually be necessary.

```
GIVE NAME FOR FORMAT SYMBOL  6.1
 ( )NAME (S-E)
END WITH (//)
CHR IS n
*
```

Having received the card image from the user, the system now endeavors to learn the meaning of each symbol used. It types out the symbols used (the n) one at a time and asks the user to provide it with the appropriate variable name and, if the variable is a string variable, to add both the number of the beginning subdivision and the ending subdivision. For example, a response could look like this:

```
* SCHOOL //
```

where SCHOOL is a non-string variable, or like this:

```
* GRADE 3-4 //
```

so that only the third and fourth subdivisions in each occurrence of GRADE will be output. (Note that there is no space between dash and subdivision number.)

For our card image example, the communication for this one message could proceed as follows:

```
GIVE NAME FOR FORMAT SYMBOL  6.1
 ( )NAME (S-E)
END WITH (//)
CHR IS 5
* ALPHA //
CHR IS 7
* MENGRADE //
CHR IS A
* TEACHER
CHR IS M
* GRADE 3-4
```

After the system has asked the user for identification of the final character, it sends this message:

```
GIVE NEXT CARD FORMAT OR INDICATE CARDS
 ( )FORMAT
 (N)O
*
```

Here you either respond with another card image or else type N to indicate that no more images are forthcoming. This message is issued to allow you to output more variables than could be set into one card image.

```
GIVE NAME OF ID-VARIABLE  6.2
 ( )NAME
*
```

The user types in the name of the case index variable or, if none is necessary as in the case of an output of data set variables, a variable that can serve as a case index variable.

The subprogram now sends out to the user a list of facts concern'ng his output, for example:

```
#DATA SETS = 1
#ID-VALUES = 640
#CARDS/ID-VALUE = 1
FIRST VALUE = 11211
LAST = 93345
TOTAL # OF CARDS = 640
INDICATE:
 ( ) STARTING BLOCK
 ( )NUMBER OF BLOCKS
END WITH (//)
*
```

(cont'd)
This information, in order of appearance, means:

the number of data sets involved

the number of case index values

the number of cards per case index value

the first value of the case index

the last value of the case index

the total number of cards that will be produced

The user is now permitted to choose either a production of all the possible cards or that of a segment--the segment must consist of a series of contiguous blocks. Although the response ends with a double slash, he can only enter one set of blocks here. For example:

* 455 4

Here he has asked the system to prepare output for blocks 455, 456, 457, and 458.

The next message in the cycle of preparing output is:

```
OUTPUT ON:
 (TT)Y
 (TA)PE
 (B)OTH
*
```

The user specifies the mode or modes of output that he wants. The teletype output of response B is used for monitoring the output on tape. The system's communication with the user is now dependent upon the response to the previous message. There are three paths which the system can take:

(1) the response is TT

No further messages are sent to the user; the next communication to the user is that of the output listing.

(2) the response is TA

The train of messages after this response is as follows:

ENTER REEL NO FILE TAPE = *

Here you supply the reel number of the tape you wish to use for holding the output. Notice that the asterisk is on the same line as the message--therefore, your response will also fall on the same line. After you have entered a number, the system delivers the message:

WAIT

This tells you that the computer operator is retrieving the tape requested and is mounting it on a tape drive. The system then produces the output, followed by the message:

END OF FILE WRITTEN
(M)ORE OUTPUT ON THIS TAPE
(N)O
*

If you wish to develop more output for the same tape, you enter the first response. Having done this, you would then return to the beginning of the routine and form another set of output data. This time, however, since the reel number is the same, the system will not ask you for a reel number. If, this second time, your response to the message:

OUTPUT ON:

is TA, you will immediately receive your output on tape, followed by the message stating that an end of file has been written.

If, on the other hand, you respond with N to the message:

END OF FILE WRITTEN

you are in effect telling the system that you are either finished with the routine or that you will prepare more output--but for another tape reel. If the case is the latter, you will again be asked to provide a reel number on the next time through the routine. If you are finished with the routine, you will exit at the next message.

(3) the response is B

The train of messages after this response begins with the initial message of that following the response of TA, that is:

```
ENTER REEL NO FILE TAPE = *
```

and

```
WAIT
```

The first message peculiar to this response is:

```
FOR TTY OUTPUT:
 ( )START BLOCK
 ( )NUMB OF BLOCKS
*
```

When comparing tape output with teletype output, the user is permitted to specify distinct groups of blocks for the teletype. (Remember that when the output is on teletype alone, the user has no opportunity to look at separate groups of blocks; he is limited to one set of consecutively appearing blocks.) The response to this message is then the first set of blocks to be looked at, for example:

```
*126 45
```

The user has asked to have printed on the teletype the output for the 45 blocks starting at block 126 and ending at block 170. The system then proceeds to prepare the specified output. The next message is:

```
LAST CARD REQUESTED
 ( )START ( )END FOR MORE TTY
 (S)TOP TTY
 (T)ERMINATE OUTPUT
*
```

Here the user may specify another set of blocks to be put on the teletype. (Remember that the total number of blocks for tape output has already been fixed--by the response to the message that asked for the starting block number and the total number of blocks immediately following the statement of the number of blocks available. This message only concerns teletype output.)

A response of S tells the system that outputting is to continue on tape but not on teletype. A response of T means that the outputting is finished for both tape and teletype. A response of a starting block number and a number of blocks means that the system is to continue outputting with this new interval for teletype output. This information does not affect tape output. This message is sent to the user after every interval of teletype output.

After the outputting is completed, the system sends its final message:

OUTPUT COMPLETE

and then repeats the first message of the routine, allowing the user to specify another card image. He exits from the routine by typing a GOTO command.

Warning: even though there may be more than one value of a variable being output per distinct case index value, you are allowed to output only one of them. TRACE warns you of this situation during the actual outputting itself, sending the following message at each encounter of two or more values of an output variable for a particular case index value:

```
FOR ID = value DS-VRB variable name HAS n VALUES
 (L)IST VALUES  (A)LL OR (#-#)
 ( )# TO USE
*
```

or, as an example:

```
FOR ID = 24 DS-VRB DISTRICT HAS 3 VALUES
 (L)IST VALUES  (A)LL OR (#-#)
 ( )# TO USE
* 2-3
```

This sample message has told the user that for the case index value (ID-variable value) of 24, the variable DISTRICT has three values. It then asks you to choose from three possible courses of action:

(1) to have all the values listed

(2) to have a particular segment of those values listed. Note that the segment must consist of consecutively occurring values and that no space should separate the numbers from the dash in your response.

(3) to select one of the values as the value to be output

In the sample response, the user has asked to have the second and third values listed. Following the requested listing, TRACE again issues this very same message, allowing you to select one of the values listed (or not listed) as the value to be output.

Note: since this message is issued <u>every</u> time a multi-value case is met, the user is in danger of having his output operation prolonged a considerable amount of time. If the situation is that many such cases are to be found in any one operation, the user should use the edit subprogram to view the values of concern and then probably derive a new variable consisting of the desired values.

PART 7

The Statistics Subprogram

For the TRACE user, this routine performs one specialized service: it evaluates differences between two groups or classes within a derived variable by running either a Mann-Whitney U Test or a Chi-square Test on that variable against one or more variables in the data base.

Suppose, for example, that we want to explain the phenomenon of underachievement in school. We want to know how underachievers differ from others in the student population. In our data base then, we would have entered a variable for students (STUDENT) and several other variables representing characteristics associated with this variable, like acuity of hearing (HEARING), family stability (FAMILY), and relative age of the student within his class level (RELAGE). Next we must have a variable that separates these students into levels of achievement in school studies, for example, three levels: a value of U for underachievers, O for overachievers, and N for normal achievers. This variable, which we can call ACHIEVE, could have been derived by comparing average grades with intelligence quotient; or, it could originally have been an input variable. One convenient method of producing this variable is by means of the ASSIGN VALUES TO CELLS work option in the display subprogram. For example, a display could have been prepared showing the correlation between actual student performance and predicted performance (perhaps on the basis of intelligence). The user could have then assigned values of U to those cells that showed a lower than expected correlation, values of O to those cells that showed overachievement, N to the rest. This new variable (ACHIEVE) would have as case index values the values of STUDENT.

So now suppose that we wanted to know whether any of the variables among HEARING, FAMILY, and RELAGE might have had any effect in separating the students into their achievement levels. In other words, we wish to understand how underachievers differ from other students in terms of family stability, hearing ability, and relative age if they actually do.

This then is the function of the statistics subprogram: to perform tests showing the relationship of one group to another by investigating their differences with respect to variables selected for their potential explanatory character. These variables are called "target variables." The variable containing the two groups is called the "criterion variable" or "criterion measure." In our example, ACHIEVE is the criterion variable, HEARING, FAMILY, and RELAGE the target variables.

Having entered the statistics routine, the user receives the following introductory message from the system:

```
STATISTICS  7.0
 (T)WO GROUP SAMPLE TEST
 (P)RESS DATA BASE GENERATOR
*
```

The user here responds with the letter T. The second response is related to an operation beyond the scope of this document. (It creates an interface between TRACE and IDEA: A Conversational, Heuristic Program for Inductive Data Exploration and Analysis--see SP-2638/000/01.)

```
2-GROUP SAMPLE TEST
GIVE CRITERION MEASURE FOLLOWED BY TARGET VARIABLES
 ( )NAMES
END WITH (//)
*
```

This message asks for the name of the criterion measure and the target variables. Only one criterion measure can be named--it must be a derived variable. Any number of target variables may be named, either derived or input or both. However, all the variables named must have the same case index or, in the case of input variables, must be in the same data set as the common case index variable. Alphanumeric variables used as either the criterion measure or a target variable must be no more than one word in length.

Example:

```
*ACHIEVE HEARING FAMILY RELAGE //
```

```
     GIVE VALUES FOR GROUP-1
     *
```

Here the user is being asked to select the value or values to form the first group within the criterion measure. For example:

```
     *U //
```

(Note that this response must end with a double slash.) This response tells us that the user is planning to relate underachievers to either overachievers or the rest of the students all together. In case the values are not alphanumeric but integer or floating point, the user may indicate a series, for example:

```
     *2 4 5 6 8 11 13 15 16 //
```

There is no provision for entering a series of consecutively appearing values as a range, for example: 14-16.

```
     GIVE VALUES FOR GROUP-2
     *
```

The user here responds with the values for the second group, for example:

```
     * O N //
```

In our example, the user has asked to relate underachievers to both overachievers and normal achievers.

TRACE now proceeds to issue a statistical analysis comparing the two groups within the criterion variable: one analysis for each target variable. Each of these analyses are dependent upon the nature of the target variable: an alphanumeric variable yields one type; a floating point or integer variable another. The following analysis is generated by a target variable that is alphanumeric.

```
FOR FAMILY
CHISQR:                          the Chi-square values are to follow:
   0.000 NSIG                    confidence level
   3.908                         obtained Chi-square
   9.236  .10                    confidence level
C =   0.1374                     contingency coefficient
DF =    5                        degrees of freedom
N1 =   89                        number of values in the first group
N2 = 2114                        number of values in the second group
CELL EXP LS5   Ø                 cell expectancy less than five
COMPLETED
 (C)ONTINUE
 (B)ASIC OPTIONS
*
```

(page 168 blank)

The following is a typical analysis for a target variable which is either integer or floating point:

```
FOR HEARING
MANN-WHITNEY TEST:
2.33  .02                    confidence level
 3.000                       obtained result (U)
3.08  .002                   confidence level
N1 =  89                     number of values in first group
N2 = 2114                    number of values in second group
COMPLETED
 (C)ONTINUE
 (B)ASIC OPTIONS
*
```

A response of C for CONTINUE will cause the system to return to the first message of the routine.

(page 170 blank)

A P P E N D I C E S

APPENDIX 1

RULES FOR TELETYPE RESPONSES

(1) An asterisk typed by the system is the signal for a user response. The response must begin on the same line as the asterisk. There need not be a space between the asterisk and the response; however, one or more intervening spaces are permissible.

(2) The forms that a response may take are always included in the message; these command one or more whole lines of type, each line representing a possible response. Each of these lines contains one or more sets of parentheses--sometimes enclosing an alphanumeric string, sometimes empty. If empty, the user is expected to type in a value (described by the words immediately following the empty set of parentheses). If the parentheses contain some entity, e.g., a string of characters, this must be typed by the user. Often, a response includes both empty and full parentheses.

(3) Whenever spaces are to be used in a response, they may occur either singly or in strings; the system treats single and multiple spaces alike.

(4) A space is the only means of separating entities on a teletype line.

(5) A system message that contains the request: END WITH // makes the user responsible for ending his response with a double slash. This convention is required by the system whenever there is a possibility that the response will extend beyond a single line; it must be used even though the response does not reach beyond the initial line.

(6) All responses are sent to the system by means of a carriage return.

(7) In case of a typing or other error, a response can be deleted (providing that it hasn't already been sent to the system via a carriage return) by typing in a quotation mark. This causes a carriage return allowing the user to begin his response anew. The system will not print another asterisk, however.

APPENDIX 2

RULES FOR CREATING NAMES

(1) A name must not exceed eight characters in length. (A data base name may be only six characters long.)

(2) It must include at least one character that is not a digit.

(3) The character string must not spell one of the system words or components, e.g.: EQ AM + AND EACH COR

(4) It must not include the following characters:

$ ' ! // *) ( ] [

(A slash occurring singly is permitted, for example: ALPHA/X)

(5) It may not hold an imbedded blank.

These rules apply to all names provided by the user, including input variable names, derived variable names and table names.

APPENDIX 3

PREPARATION OF DATA CARDS

The data deck consists primarily of a data dictionary and a set of data cards. There must also be at least five control cards. These members of the entire data deck appear in the following order (see Figure 1 at the end of this appendix):

(1) START DICT (control card--these two words may start in any column.)

(2) The dictionary--each variable is represented by a card--each card has the following information (the elements in this information do not appear in specific fields, e.g., the variable name might begin at column 18; however, the elements must appear in the correct order, separated from each other by one or more blanks):

- -1- variable name
- -2- type of variable: A for alphanumeric, F for floating point, I for integer
- -3- starting column for the variable for its particular card in each block
- -4- total number of columns (e.g., if a string variable has a total of five two-column values per occurrence, the answer is 10).
- -5- number per subdivision: this is zero for non-string variables. It means the number of columns for each value in the string.
- -6- skips between subdivisions: the number of columns intervening between appearances of separate values in a string variable.
- -7- beginning card: the card in the block on which the value for the variable starts; for example, if values for a particular variable start in column 68 of the first card of a two-card block and end in column 23 of the second card, the answer here would be 1.
- -8- ending card: the answer for the previous example would be 2.

Note: nothing may appear in column 80 of any dictionary card.

Note: there should be not more than 150 cards in the dictionary.

(3) END CASE *variable name*--this is a control card starting with the words END CASE followed by the name of a variable selected as a general case index for the data. If there is one variable that has a unique value in every block, e.g., STUDENT in our data set of Section 1, this should be named. Otherwise, any variable can legitimately serve--this is the variable that will be recognized as the case index in any derivation if the user chooses not to name one in the derivation subprogram.

(4) START DATA (control card--start in any column).

(5) the data cards--these must follow the specifications laid down by the dictionary. Note: no data may appear in column 80 of any data card. The cards in the data section do not have to follow any order from block to block. That is, within each block of multiple cards, the individual cards must be in the correct order; however, the blocks themselves can be arranged in any order.

(6) END DATA (control card--start in any column).

(7) END INPUT (control card--start in any column). This is a special control card (also called the TSS EOF card) prepared as follows:

col. 1 punch 7 8 9

col. 2 punch 7 8

col. 3 punch Ø 3

Punch the letters ENDINPUT starting in column 15.

There is also one partially optional type of information that can be entered onto one of the control cards. An estimate of data set size can be appended to information already on the END CASE card. This size is in terms of blocks of data for the entire set; it need not be the exact number of blocks but it should not be an underestimate. This figure appears on the control card as follows:

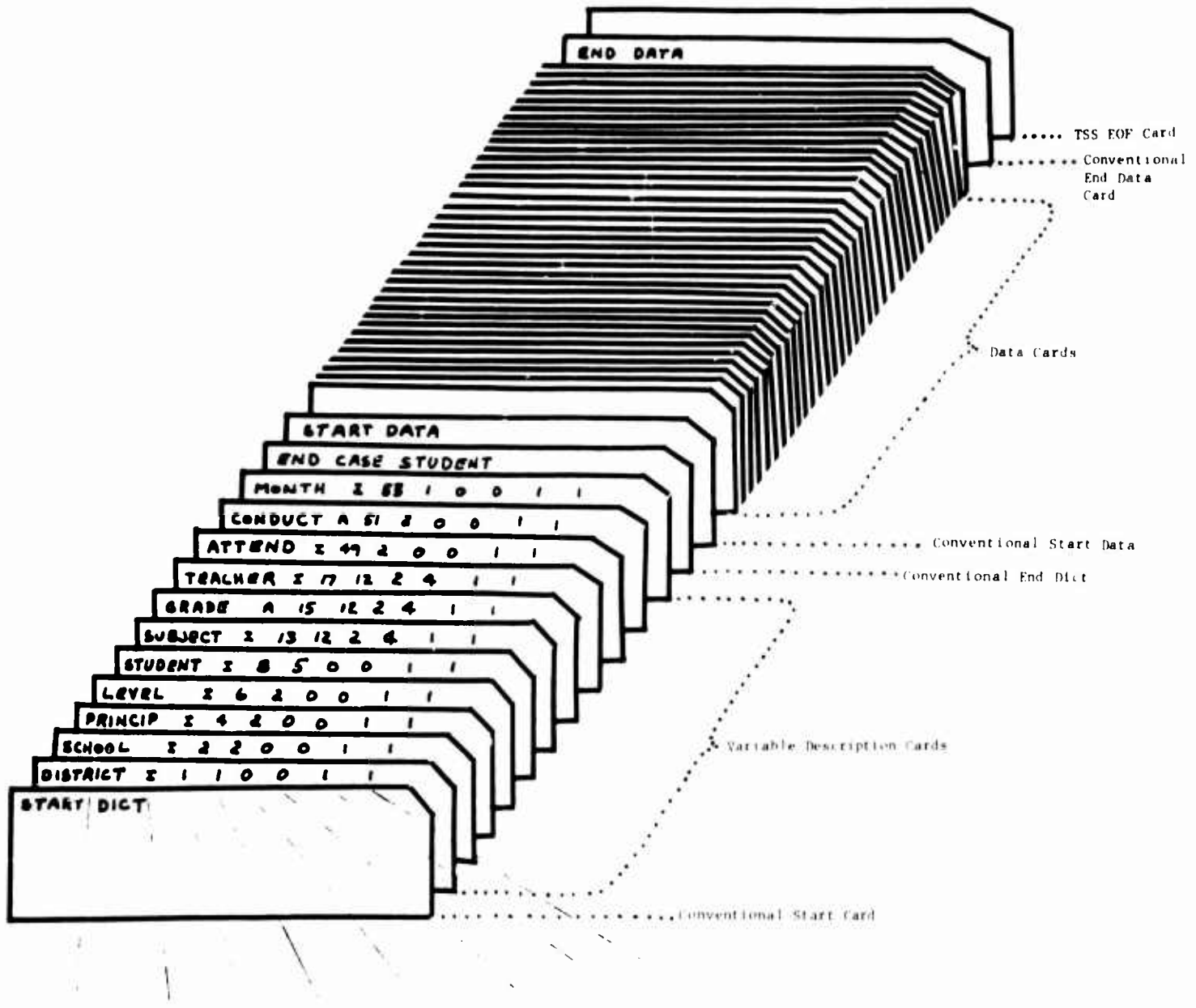
END CASE STUDENT 1999

If no such estimate is entered into the data deck, the system will ask you for it during the inputting stage. (See references to the input subprogram.) Note, however, that ordinarily this estimate should appear whenever the data set is on disc.

There must be a new dictionary for every data set. Each data set may be completely independent of any other data set; that is, it may contain variables not part of the other sets. However, if the same variable appears in two or more sets, it must be of the same type (alphanumeric, integer, floating point), have the same character length and subdivisions. It need not occupy the same card columns in the different sets, nor the same starting and ending cards within a block.

Note: although units of alphanumeric data may be as long as 120 characters long, TRACE, in many cases, can only work with eight characters at most, at one time. If you wish to escape this restriction, you must convert your data so that it can be worked with. This can be done during the punching of the data or it may be left for the edit subprogram to handle.

(page 178 blank)



VARIABLE NAME	TYPE OF VARIABLE	STARTING COLUMNS	NUMBER OF COLUMNS	NUMBER PER SUBDIVISIONS	SKIPS BETWEEN SUBDIVISIONS	BEGINNING CARD	ENDING CARD
DISTRICT	I	1	1	0	0	1	1
SCHOOL	I	2	2	0	0	1	1
PRINCIP	I	4	2	0	0	1	1
LEVEL	I	6	2	0	0	1	1
STUDENT	I	8	5	0	0	1	1
SUBJECT	I	13	12	2	4	1	1
GRADE	A	15	12	2	4	1	1
TEACHER	I	17	12	2	4	1	1
ATTEND	I	49	2	0	0	1	1
CONDUCT	A	51	2	0	0	1	1
MONTH	I	53	1	0	0	1	1

Figure 1

APPENDIX 4

THE Q-32 TIME-SHARING SYSTEM

TRACE II can function only within the Q-32 time-sharing system. Thus, before the user can begin his work on the TRACE program, he must first bring that program into the system. This means, naturally, that he must himself become one of the time-sharing users. Once the user and the TRACE program have been accepted in the system, interaction between them proceeds without visible recourse to the system.

All interaction between the user and the TRACE program or between the user and the time-sharing system is by means of the teletype--essentially an electric typewriter. The first action by the user on this teletype is the typing of:

!STATUS (followed by a carriage return--all responses are communicated to the system by a carriage return)

You then wait for a response from the system that says the following:

$NO LOGIN

This system response tells you that it is alright for you to enter the system. If a response other than this is forthcoming, it means that the teletype is already in use by someone else.

You "log in" with the following type of message:

!LOGIN 7688 94000

where the number 7688 is the man number of the user and the 94000 the work order number. Both of these numbers must be known and acceptable to the system.

Note: messages from the user directly to the time-sharing system must begin with a !. Responses from the system begin with a $.

If this logging-in message is honored by the system, it responds with a message of the type:

$OK LOG ON 13 12:36.1 10/13/67

where the number 13 is the channel (teletype) to the system that is to be used,

12:36.1 is the time of day, and 10/13/67 is the day of the year. If the request to log in is not honored, the system will issue the reason for the denial.

You are now expected to work with one of the system's library programs. Our first project is that of getting the TRACE program into the system. Because of its size, TRACE is not a permanent member of the disc library--it exists only on magnetic tape. The project specifically involves bringing the TRACE program from tape onto so-called drum storage, where all operating programs must reside, although this residence can only be temporary. However, no direct transfer can be made from tape to drum--transfers are possible from tape to disc and from disc to drum. Our project is therefore expanded: we must first transfer TRACE from tape to disc, and then from disc to drum. The first step can be accomplished by means of the library program LIBRY, which serves to move files from tape to disc or from disc to tape. The document describing the operation of this program is TM-2708/203/01. The process of using LIBRY to bring TRACE onto disc is as follows (user input in blocks):

!LOAD LIBRY	
$LOAD 13	
GO	(now that you are no longer communicating directly with the system but with one of its library programs, the response signals ! and $ are not applicable)
$MSG IN	
ENTER OPTION DESIRED	
TAPE TO DISC	
INNAME FORM (B,H,C,A,P,T) LIBRARY REEL	
TRACE P 1478	(the name of the program, the letter P for program, and the reel number of the tape holding the program--currently the TRACE program resides on reel #1478)
$ WAIT	

```
$FILE FL0001  DRIVE 14  REEL 24Ø3

ANOTHER FILE?  Y/N

Y                                   (The first file brought in the TRACE
                                    executive program--now you must enter
                                    the TRACE subprograms.)

INNAME FORM (B,H,C,A,P,T)  LIBRARY REEL

TRACES B

ANOTHER FILE?  Y/N

N

STANDBY

THE FOLLOWING TAPE REELS ARE FILED

2403

ENTER OPTION DESIRED
```

At this point, TRACE (the executive program plus all its subprograms) has been placed on disc and you are now ready to begin the transfer from disc to drum. LIBRY does not affect this type of transfer--it is one of the functions of the time-sharing system's executive. Therefore, you call on the system again. (Remember to use the signal ! again.)

```
!LOAD TRACE
```

The system can honor this request only if there is enough space on the drums to accommodate the TRACE executive program, which consists of 46,456 words. (The subprograms remain on disc--to be called by the executive whenever necessary.) If the drum space is not sufficient, the system will issue the message:

```
$ NO LOAD DRUMS FULL
```

This instructs the user to wait until another user or two leave the system, thereby freeing enough drum space for TRACE. If drum space is immediately available, the system returns the favorable message:

```
$LOAD 13                            (channel 13, of course, appears here
                                    only as an example)
```

The user replies with:

GO

which elicits the first message of the TRACE program:

$MSG IN.
YOU ARE STARTING TRACE 0.1
NAME YOUR DATA BASE
( )NAME
*

Suppose that the system tells the user that no drum space is available. How will you know when enough drum space for TRACE becomes available? There are two ways of learning this:

(1) Continue trying to load TRACE--when enough space becomes available, the request will be fulfilled.

(2) Ask the system immediately and from time to time for the amount of drum space available. This communication may run as follows:

!DRUMS

$1208

This signifies a long wait. You would have to wait for more than 45,000 words to open up.

On the other hand, a system response of:

$ 42236

would tell you that sufficient space should certainly be available upon the withdrawal of the very next user from the sytem.

The user enters the time-sharing system briefly upon the close of his interaction with the TRACE program. To indicate to the system and thereby to the other prospective users of the system (with your more than 45,000 words of drum space, you may be preventing several potential users from entering the system) that you are finished and can relinquish your drum space, you type the message:

!QUIT

This message typed in at any point in any program such as TRACE, immediately terminates your interaction with that program--and also with the time-sharing system.

A Summary of TRACE Residences

- In operation, the TRACE executive always resides on drums.
- In operation, the TRACE subprograms stay on disc, but are copied onto drums whenever needed--one at a time as they are called.
- The data base resides on disc; portions that are needed are brought into core (drums). The first track on disc is a bookkeeping track, outlining the structure of the data base.
- All additions to the original data base are placed on disc and treated subsequently like any other portion of the data base.

APPENDIX 5

TROUBLE IN TRACE --
and how to get out of it

Like any partnership between mind and matter, the latter sometimes seems to have a mind of its own and the partnership comes to a breaking point. So it can be with the user's interaction with TRACE--sometimes things can go awry. For example:

- the time-sharing system itself may fail
- the TRACE program may fail
- the user may be at fault, doing something illegal (e.g., trying to divide by zero)

Whenever there is a TRACE failure, or a user-instigated malfunction occcurs, TSS (the time-sharing system) assumes control, sends a message to the user informing him of the situation, and waits for a remedy. A typical message asking for rescue is:

```
$ PRGM STOPPED     DISX     DISPATCHER CALL ERROR
```

The best cure for this is a reloading of the TRACE program. Simply type:

```
!LOAD TRACE
```

and retrace the loading procedure. Remember that TRACE is still safely secured in disc storage--you need not worry about calling LIBRY to bring in TRACE from tape. And your data base is also safely on disc--you do not have to input your data again.

A failure of TSS itself may require the user to begin all his operations over again, starting from the business of logging in. The system may inform the user of a regaining of control if he remains at the teletype; on the other hand, he can initiate a call to the control center if that is convenient (extension 221).

There may be occasions when the user has doubts about the program's operation (unusually long delays, unexpected responses, for example). In this case, he may ask TSS about his program's status. He simply types:

```
!STATUS
```

The system then answers with an appropriate message, for example:

```
$ WAITING FOR TTY
```

which means that the system is waiting for a response from the user (perhaps he thought he responded but, for some reason, the response never reached the program). Notice that he was able to talk directly with the system merely by prefacing his message with an !. If he now wishes to continue interaction with his program, he must signal this intention with a ". For example:

```
"GOTO DE
```

which will take him out of the system and back into his program at the specified point.

Generally, you will do better to stay with the TRACE program unless you have strong evidence that something has gone wrong.

APPENDIX 6

SAVING THE DATA BASE FOR FUTURE USE

As the user interacts with the TRACE program, he is actually building up his original data base, adding to it various derived variables and tables constructed for display. All this is stored on disc. However, this storage is really only temporary; it is reserved solely for current time-sharing users. It may be that because of certain coincidences in the way succeeding data bases from other users are stored, a particular user will find that his own data base has remained intact over a period of several hours. This chance should not be counted upon--there is no way of automatically guaranteeing the retention of any information on disc.

If you do wish to preserve your data base for future use, there is only one convenient means of doing it--by transferring it from that temporary disc storage to magnetic tape. This can be accomplished by the system utility program LIBRY.

LIBRY may be called whenever you feel ready to store your data base. This could be at the end of a problem run when you have just displayed your results, or it could even be in the middle of some operation, like setting up the derivation or constructing a table. TRACE, however, does not transfer the results of any operation from core to disc until the user exits from the subprogram containing that operation or at least reaches a message that states that the operation is complete. Stopping in the middle of an operation means then losing whatever has been gained during that particular operation. To avoid this loss, the user should type in a GOTO command to the beginning of a subprogram, either the current one or a different one. This step will preserve the results of the responses entered during the aborted operation.

The call to LIBRY is simply:

```
!LOAD LIBRY
```

followed by the usual carriage return. The ! is used to leave the TRACE program and enter the time-sharing executive system; the command asks the system to load the LIBRY program.

For a description of the LIBRY program, see the document: TM-2708/203/01.

The dialogue between you and the LIBRY program might proceed as follows (user input in blocks):

```
! LOAD LIBRY

$ LOAD 17

GO

$ MSG IN

ENTER OPTION DESIRED

DISC TO TAPE

INNAME FORM (B,H,C,A,P,T) MAN NO. LIBRARY REEL INDEX ON TAPE?  Y/N

EDUC B 1545 1263 N

ANOTHER FILE?

NO

STANDBY

THE FOLLOWING TAPE REELS ARE FILED

12631

ENTER OPTION DESIRED

! QUIT

$ MSG IN
```

This process can be reversed--LIBRY can be used to read onto disc a file from a tape. You simply call LIBRY and, at the message requesting an option, enter the option:

```
TAPE TO DISC
```

TM-2708/203/01 describes both processes in detail.

APPENDIX 7

ERROR MESSAGES

Each subprogram has an individual complement of error messages--messages that are sent by the system to the user whenever he has entered some invalid information or when the system for some reason cannot fulfill his request. Each message ends, like any message within a subprogram, with a line containing an asterisk which the user is expected to follow with a response. The error messages of TRACE are grouped according to their corresponding subprograms.

PART 1

The Input Subprogram

```
ERROR IN SELECTING VARIABLES
INDICATE THE VARIABLES TO BE READ  1.12
 ( )NAMES
ALL
*
```

This message is issued if the user has inadvertantly asked for one or more variables that either do not exist in the input data or else have already been read by the system. A simple typographical error, of course, would also trigger this error message. It is issued immediately after the user has entered his response to message 1.12; the system then repeats this message 1.12, giving the user the opportunity to revise his response.

```
     RAN OUT OF DISC
      (K)EEP TRYING
      (A)BORT PROCESS
     *
```

The system here is telling the user that all available space on disc has been used up and that all the variables listed by him in his response to message 1.12 cannot be read in. You then have the option of either waiting for enough space to open up on disc so that your data can be read in (the first response) or telling the system to halt the process of trying to input your specified variables (the second response). This message is issued after your response to message 1.12 and after the system has discovered that the entire set of variables cannot be accommodated. At this point, the tape or disc for input has been positioned for reading but no data at all has been read, even if there is room enough on disc for all but one word of the input.

Asking the system to "keep trying" means waiting indefinitely with the input file positioned for reading until the required space becomes available. Asking for an "abort"causes the system to return tc the first message of the input subprogram, thereby allowing you to repeat the attempt, this time perhaps requesting in your response to message 1.12 only those variables that are necessary for the first derivation.

```
     CASE INDEX NAME IN ERROR
      ( )NAME
     *
```

This error message stems only from a user error in specifying the case index variable in response to message 1.30 in the teletype input cycle. The variable name typed in may have no counterpart in the data or it may simply contain a typographical error. At any rate, the system asks you to retype the name of the case index variable.

PART 2

The Derivation Subprogram

```
      NO DICT FOR THE CASE INDEX
      DERIVATION  2.0
       (S)TART
       (L)IST AVAILABLE VARIABLES
      *
```

This error message, issued if necessary after his response to message 2.1, tells the user that the variable he has designated as the case index is not in the dictionary. This may be, for example, because he has chosen a derived variable as a case index variable (only input variables can be used) or has typed in an invalid name. The system returns him to the beginning message of the derivation routine.

```
      NULL SET
      DERIVATION  2.0
       (S)TART
       (L)IST AVAILABLE VARIABLES
      *
```

This message, issued after you have entered your restriction statement (response to message 2.11), states that the subset that you have requested for your derivation is empty; that is, you have restricted your original data base so much that no case index values remain for use in deriving. The system sends you back to the beginning of the subprogram.

```
      RAN OUT OF DISC
       (K)EEP TRYING
       (A)BORT PROCESS
      *
```

Similar to one of the error messages in the input subprogram, this tells you that there will not be enough room to place all the derived values on disc and that you may either wait for space to become available, or else stop the process of deriving altogether. If you choose the latter response, the subprogram will return you to its first message and you have the opportunity to set up a different derivation.

```
      ERROR IN REQUEST
      DERIVATION  2.0
       (S)TART
       (L)IST AVAILABLE VARIABLES
      *
```

This error message is produced after the derivation statement--it states that some mistake has been made in setting up that statement, for example, following the connective OTR with a conditional clause or typing two consecutive IF clauses without an intervening conjunction. The user is asked to return to the very beginning of the subprogram.

```
      NO COMMON DICT FOUND FOR VARB AND CASE INDEX
      DERIVATION  2.0
       (S)TART
       (L)IST AVAILABLE VARIABLES
      *
```

Also issued after the derivation statement has been entered, this message declares that the hoped-for derived variable does not have any correspondence with the current case index--that the case index variable and the variables used in the derivation statement probably belong to different data sets. The user is requested to begin the subprogram anew.

(page 194 blank)

```
INTERNAL TABLES EXCEEDED
DERIVATION  2.0
 (S)TART
 (L)IST AVAILABLE VARIABLES
*
```

The user is told here that his derivation statement demands more data than the system's internal tables can handle. There are two of these tables that can be affected by too much data:

(1) the table that collects the case index values appropriate to the derivation and that associates these with the derived values--this table can accommodate a maximum of 2800 case index values

(2) the table that collects the appropriate case index values and annexes to them the corresponding values of the other variables used in the derivation--this table can hold 10,000 values all together

An overflow of either of these tables results in the issuing of the above error message. The user then has only one alternative if he wishes to complete his original derivation: he must institute artificial restrictions on his case index values so that not all values would be recognized at once. For example, he may set up a restriction statement that effectively divides his case index values in half. He then runs two derivations, one with each half. This message, which in effect is telling you that there is not enough restriction on your derivation, also sends you back to the beginning of the subprogram.

(page 196 blank)

PART 3

The Construct Table Subprogram

```
      NO SUCH NAME
      CONSTRUCT TABLE  3.0
       (S)TEP BY STEP
       ( )SHORTHAND
       (L)IST SET UP TABLES
      *
```

This error message is sent whenever in the construct table routine the user has made a mistake in typing in a variable or table name, for example, misspelling a name. He is returned to the beginning of the routine.

```
      ERROR IN CASE INDEX
      CONSTRUCT TABLE  3.0
       (S)TEP BY STEP
       ( )SHORTHAND
       (L)IST SET UP TABLES
      *
```

This message is sent after the user's response to message 3.4. It states that the two variables named in setting up a two-dimensional table do not have the same case index--a mandatory condition for that type of table. He is returned to the beginning of the routine.

If you have used the shorthand method, these two messages would be sent after you have entered that response.

PART 4

The Display Subprogram

```
      NO SUCH NAME
      DISPLAY  4.0
       ( )NAME TABLE
       (L)IST CONSTRUCTED TABLES
      *
```

This message tells the user that a name used in the previous response does not exist. He is then returned to the beginning of the routine. The message may be issued after a response to message 4.0, when the invalid name was to have been the name of the table, or after a response to message 4.2, when the error message refers to an improper name for the display, the row variable, or the column variable.

```
      NO DISC
       (K)EEP TRYING
       (N)O
      *
```

This message, sent in place of the desired table display after your response to message 4.1, informs you that not enough disc space is available for the production of the display. The display routine at this point uses some tracks on disc for building intermediate tables. A response of K tells the system to wait until enough disc space has been released by current time-sharing users; a response of N tells the system to return to the beginning of the subprogram.

```
TABLE REDUCED TO LESS THAN A 2 X 2
DISPLAY  4.0
 ( )NAME TABLE
 (L)IST CONSTRUCTED TABLES
*
```

The system is telling you that, even though you have constructed a table of two-dimensional proportions, the resulting table ready for display contains only one dimension (that is, either a single row or a single column) because so many prospective intervals and therefore potential cells in the display do not hold any values. TRACE does not display such tables unless they have been designated as one-dimensional tables during the construct table subprogram.

Notice that the system has presented you with the first message of the display program. If you, however, wish to have that invalid table legitimized, you should type in a GOTO command to the construct table subprogram instead of accepting either of the preferred options. There you may begin to reconstruct the table, probably using the shorthand method, in a one-dimensional form. First attempt to reconstruct the table on the basis of two variables alone; if the same error message appears later in the display program, then construct the table as a single row or column.

This error message is issued after the user's response to message 4.1 of the display program.

```
ONLY ONE VARIABLE IN THIS DISPLAY
DISPLAY  4.0
 ( )NAME TABLE
 (L)IST CONSTRUCTED TABLES
*
```

This message is produced after the user's response to message 4.3 where he has asked to have a display of a table using the expected frequencies rather than the observed frequencies--and the table is only a one-dimensional table. The system is therefore reminding him that he has only one variable to work with and that there can therefore be no expected frequencies, in fact, no frequencies at all. He is brought back to the beginning of the display routine.

PART 5

The Edit Subprogram

```
      NO SUCH NAME
      EDIT  5.0
       (D)ELETE
       (L)IST
        .
        .
        .
       (B)ASIC OPTIONS
      *
```

This message appears whenever in the edit subprogram a response contains an invalid variable name, for example, because of a typographical error. The user is returned to the beginning of the routine.

```
      ALL VARIABLES ARE NOT IN ONE DATA SET
      EDIT  5.0
       (D)ELETE
       (L)IST
        .
        .
        .
       (B)ASIC OPTIONS
      *
```

This error message is triggered only by an improper response to message 5.2 in the edit program where the user has asked for a listing of the values of certain variables. If these variables do not all belong to the same data set, it is an error condition. The user is returned to the beginning of the edit routine.

```
      TOO MANY PRINT POSITIONS REQUIRED FOR VALUES
      EDIT  5.0
       (D)ELETE
       (L)IST
       .
       .
       .
       (B)ASIC OPTIONS
      *
```

This message is incurred when the user, in asking for a listing of the values of designated variables in response to message 5.2, has nominated too many varisbles--so that the values if printed would overflow a print line. He is brought back to the beginning of the edit program where he can again call for the listing function and this time request fewer variables.

```
      NO OTHER NAMES FOUND
      INDICATE WHICH DATA SET  5.22
       ( ) 1 TO 10
       (N)ONE
      *
```

This message is issued after a response to message 5.22 in the list function doesn't give the proper set number for the variables designated. The user then must either type in the correct set number or else enter a GOTO command to another routine.

PART 6

The Output Subprogram

```
      NO ID-NAME FOUND
      GIVE NAME OF ID-VARIABLE
       ( )NAME
      *
```

This error message, issued after the user has entered his response to message 6.2, tells him that he has made a mistake in naming his ID-variable. Either there is a typographical error or else the variable no longer exists. He is asked to give the name of a legitimate variable.

```
      THIS VARIABLE FAILED PRE-CHECK variable name
       (C)ONTINUE
       (N)O
      *
```

The first line of this message names a variable which the user has typed as one of the output variables. The system informs the user that this variable has no values for the case index (ID-variable) values being used for output. He is then allowed to either continue with the output, which means turning out blanks for the values of the variable in question, or else abort the process and return to the beginning of the subprogram where he may again try to form a card image, this time avoiding the variable that has no values. This error message is issued after output message 6.2.

```
      SET = NULL
      OUTPUT  6.0
      GIVE CARD FORMAT
      .
      .
      .
      LINE MUST END WITH A BLANK
      *
```

This message signals that there are no values for the case index variable (ID-variable) selected; therefore, there is no possibility of getting any data at all for output. This message, sent after message 6.2 in the output routine, then carries the user back to the beginning of the routine.

```
      SET = ONE
      OUTPUT  6.0
      GIVE CARD FORMAT
      .
      .
      .
      LINE MUST END WITH A BLANK
      *
```

This message states that only one value exists for the ID-variable selected. Issued after message 6.2, this error message then repeats the first message of the output routine.

PART 7

The Statistics Subprogram

```
      target variable name HAS NO CASE INDEX OR DATA SET IN COMMON WITH
      criterion variable name
     2 - GROUP SAMPLE TEST
     GIVE CRITERION MEASURE FOLLOWED BY TARGET VARIABLES
      ( )NAMES
     END WITH (//)
     *
```

This error message is issued whenever the user has designated as a target variable an input variable that is in a data set not containing the case index variable of the criterion measure. The user is then returned to the first message of the 2-group sample test cycle where he may enter new target and criterion variables.

```
     N1 OR N2 IS LESS THAN 2
     2 - GROUP SAMPLE TEST
     GIVE CRITERION MEASURE FOLLOWED BY TARGET VARIABLES
      ( )NAMES
     END WITH (//)
     *
```

This message tells the user that there are fewer than two values for one of the two groups of values that make up the criterion measure. He is asked to repeat the request using a different grouping.

```
 target variable name HAS DIFFERENT CASE INDEX THAN  criterion variable name
2 - GROUP SAMPLE TEST
GIVE CRITERION MEASURE FOLLOWED BY TARGET VARIABLES
 ( )NAMES
END WITH (//)
*
```

This message is issued whenever the user has named as a target variable a derived variable that has a case index different from that of the criterion measure. The user is returned to the first message of the routine.

```
variable name HAS MORE THAN 1 WORD/VALUE
2 - GROUP SAMPLE TEST
GIVE CRITERION MEASURE FOLLOWED BY TARGET VARIABLES
 ( )NAMES
END WITH (//)
*
```

This message tells the user that the variable value surpasses a word in length. Remember that an alphanumeric variable as either the criterion measure or a target variable must be no longer than one word.

APPENDIX 8

CORE AND COMPONENT RESTRICTIONS

The following is a list of restrictions imposed upon the user by the limitations of core space and the structure of the TRACE program.

- A maximum of 10,000 words* (Q-32 computer) may be transferred at any one time from the data base on disc into the working area in core.
- The case index table can accommodate 2800 occurrences of a case index variable.
- A maximum of 353 variables may be input by the user for the entire data base.
- There is a limit of 150 variables per data set; a dictionary may therefore hold up to 150 cards.
- A total of 181 variables may be derived and saved.
- The total of input and derived variables outstanding at any one time must not exceed 385 variables.
- A maximum of 10 data sets may be input by the user.
- A total of 60 tables for display may be carried by the program.
- The number of intervals in these tables and the total number of values within these intervals must not exceed 960.
- A maximum of 120 characters may be declared in any one alphanumeric variable.
- A restriction statement may contain no more than 10 variable names nor more than 50 restricting values.
- A derivation statement is restricted to a total of 50 components. Within this restriction is a limit of 20 unique variable names and 20 occurrences of the conjunctions AND and OR.

*A computer word can accommodate one integer value, eight characters of an alphanumeric value, or one floating point value. No packing is done, so that a one-character value still occupies a whole word.

APPENDIX 9

STATISTICAL OPERATIONS OF TRACE

- Mean
- Median
- Standard deviation
- Chi-square or Fisher's exact probability
- Pearson's R correlation coefficient
  with 5% confidence and symmetric interval limits
- t statistic
- F statistic
- Tabular frequency distribution
  with equal number, width, all or specified interval limits
- Expected value matrix
- Number of cells with expected value less than 1
- Number of cells with expected value less than 5
- Degrees of freedom
- Significance
- Contingency coefficient
- Mann-Whitney U test

All of these operations are conducted within the display subprogram with the exception of the Mann-Whitney U test, which is found in the statistics subprogram. The statistics program also runs the Chi-square test with its associated products.

Besides these specialized statistical services, TRACE provides the following operations:

addition	sine	square root
subtraction	cosine	absolute value
multiplication	tangent	exponentiation
division	arcsine	logarithms (to any base)
	arccosine	
	arctangent	

INDEX

This is an index primarily of topics whose complexity has earned them mention and explanation beyond the reference section. Except for the various components of the derivation process, there is little duplication of the references shown in the table of contents for Section 2.

INDEX (cont'd)

Unclassified
Security Classification

DOCUMENT CONTROL DATA · R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
System Development Corporation
Santa Monica, California

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE
TRACE -- Model II User's Guide, Timeshared Routines for Analysis, Classification and Evaluation

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*
Richard P. Esada

6. REPORT DATE
9 October 1967

7a. TOTAL NO. OF PAGES
208

7b. NO. OF REFS
-

8a. CONTRACT OR GRANT NO.
DAHC15-67-C-0277

b. PROJECT NO.
Bargaining and Negotiation Behavior for ARPA

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
TM-2621/003/00 (DRAFT)

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT
Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

This document presents a user's description of the TRACE system, which provides an on-line technique for scanning data and deriving variables. It is divided into two main sections: the first a tutorial guide introducing the user to the basic principles of the system, and the second a reference guide to the entire body of the TRACE program. The user is shown how to initiate an interaction with the time-sharing system, how to employ every capability of TRACE, what errors may be expected in operation, and what statistical products may be derived through use of the program. A complete index allows the user to refer readily to any portion of the document.

DD FORM 1 NOV 68 1473

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
TRACE system Time-shared Routine for Analysis, Classification and Evaluation On-line technique JOVIAL AN/FSQ-32						