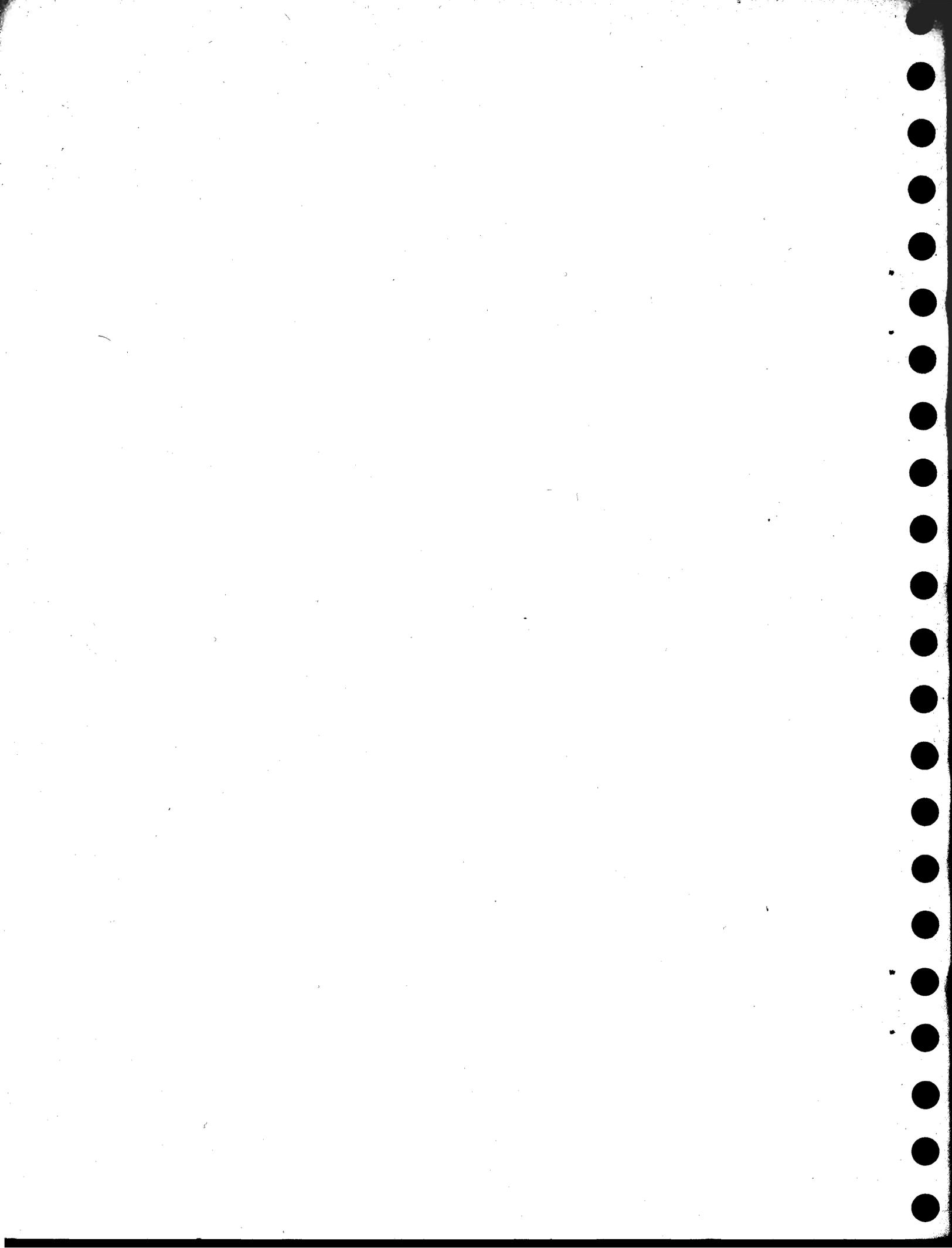


IBM

Customer Engineering
Manual of Instruction

FORTRAN
I, II, AND 709



IBM[®]

Customer Engineering

Manual of Instruction

FORTRAN
I, II, AND 709

Issued to: _____

Department or Telephone
Branch Office _____ Number _____

Address _____ City _____ State _____

Home Address _____ City _____ State _____

If this manual is mislaid, please notify the above address.

© 1959 by
International Business Machines Corporation

Form R23-9518-0
Printed in U. S. A.

FOREWORD

THE PURPOSE of this manual is to familiarize the IBM customer engineer with the language and data processing methods found in the Fortran automatic coding systems. Some service hints are also included where available.

The reader is expected to have at least a cursory knowledge of the Fortran Primer and Fortran Reference manuals. With this background, the material in this manual can be of great value; without it many points will be obscure. Wherever possible, references are made to the two manuals just mentioned.

Much thought has been given to what should be included in this manual. A survey of customer engineering opinion indicates that the most difficult problems are experienced in the data transmission area. Therefore, this manual concentrates most heavily on input-output, and mentions arithmetic processing only briefly.

A word should be said concerning the interrelationships of the various Fortran systems. At this time the Fortran I and II systems have been stabilized; that is, there will be no more major changes in the organization of their executive programs. The Fortran 709 system is at present in a state of major change. In general, much will be common in all three systems. For instance, the function of all tape units in the three Fortran modes will be the same. Where the drum was used in Fortran I and II, upper core storage will be used by Fortran 709. More diagnostic routines are written into Fortran II and Fortran 709 than in Fortran I. All manual operation features of the three systems are alike, except for the use of sense switch 6 to control batch compiling in Fortran II and Fortran 709.

This manual supersedes the Fortran Translator Customer Engineering Manual of Instruction, Form 29-9437-0. It supplements the following published IBM manuals:

Title	Form
Programmer's Primer for Fortran	32-0306
Programmer's Reference Manual Fortran	32-7026
IBM Reference Manual, Fortran II (S)	C28-6000
IBM Reference Manual, Fortran II (S)	C28-6001
IBM Reference Manual, Fortran Automatic Coding System	C28-6003
IBM Fortran General Information Manual	F28-6019



CONTENTS

1.00.00 FORTRAN EXECUTIVE ROUTINE	7	2.00.00 GENERAL FORTRAN RECORD STRUCTURE	33
1.01.00 Introduction	7	2.01.00 CALLFN Record	33
1.02.00 General Organization of Translator	13	2.02.00 CLOSUB Record	33
1.02.01 Section 1	13	2.03.00 Compiled Instruction Tables (CIT)	33
1.02.02 Section 1'	14	2.04.00 Common Record	34
1.02.03 Section 2	14	2.05.00 DIM Record	35
1.02.04 Section 3	14	2.06.00 DOTAG B Format	36
1.02.05 Section 4	14	2.07.00 END Record	37
1.02.06 Section 5 and 5'	15	2.08.00 EQUIT Record	38
1.02.07 Section 6	15	2.09.00 FIXCON Record	38
1.03.00 Number of Instructions in Each Section	15	2.10.00 FLOCON Record	39
1.04.00 Fortran Systems Tape	15	2.11.00 FORMAT Record	39
1.04.01 General Organization	15	2.12.00 FORSUB Record	39
1.04.02 Fortran I Ordinary System Records	16	2.13.00 FORTAG Record	40
1.04.03 1-CS (Tape 1 to Core Storage Program)	18	2.14.00 FORVAL and FORVAR Records	40
1.04.04 The Tape Record Monitor	18	2.15.00 FRET Table	41
1.04.05 Listing of 1-CS on Tape	19	2.16.00 HOLARG Record	42
1.04.06 Listing of 1-CS in Storage	21	2.17.00 LAMBDA, ALPHA, and BETA Tables	42
1.04.07 Control Words of 1-CS Monitor	22	2.18.00 NONEXEC Record	43
1.05.00 Updating the Edit Deck	22	2.19.00 SIGMA and TAU Tables	43
1.06.00 Fortran Tape Assignment	24	2.20.00 SIZE Record	45
1.07.00 Relating Tape Assignment Fortran I & II to Fortran 709	24	2.21.00 SUBDEF Record	45
1.08.00 Fortran Source Program Card Format	26	2.22.00 TDO Record	46
1.09.00 Using the Fortan System Tape	27	2.23.00 TEIFNO Record	46
1.10.00 Running the Object Program	27	2.24.00 TIFGO Record	47
1.11.00 Description of Tape and Drum Usage During a Fortran I Run	28	2.25.00 TRAD Record	49
1.12.00 Service Aids	29	2.26.00 TSTOP Record	50
1.13.00 Fortran II Refinements	30	3.00.00 FORTRAN AUTOMATIC CODING SYSTEM.	51
1.14.00 Answers to Common General Questions Involving Fortran.	30	-----	
		TABLE I	62
		TABLE II	64



1.00.00 FORTRAN EXECUTIVE ROUTINE

1.01.00 INTRODUCTION

The problems involved in man's communication with the complex computer are in many respects similar to those problems involved with his communication with another man who speaks an unfamiliar language. There must be language translation in either situation in order that the communication be fruitful. In considering current problems of communication with the computer we will consider here only the medium of "written" language, although there are research teams seriously engaged in communicating with the machine by the spoken word.

The Frenchman who must learn English has a great advantage over the Oriental who is faced with the same task. The former need not learn a new set of symbols (Roman and Arabic) that is the first task of the latter. The position of man and machine is more akin to the Oriental's problem of learning English than that of the Frenchman. Man must first learn the symbols that the machine uses to build its words.

The machine's symbolism (speaking now of the 701, 704, 709, and 7090) is binary in nature. The need for recognition of two symbols makes the internal circuitry of the machine relatively simple. In terms of machine hardware, the binary symbol takes the form of a component conducting or not conducting, a pulse available or not available, a core saturated in one direction or the opposite direction, or a changing magnetic flux on tape or the lack of that change. The internal evidence of this binary symbolism is brought to the surface of the machine in the readable form of light patterns. The individual lights are binary, in that they are either on or off. Needless to say, the normal man has little difficulty learning the binary symbolism of the computer.

The Oriental should not be considered an English scholar, simply because he has learned the 36 symbols that express all our literature. Similarly, the man who wants to communicate with the machine should not feel satisfied that he knows machine language by virtue of his recognition of the on or off condition of a light. He must learn the limitations and rules involved in grouping the symbols together into intelligible patterns to form words, and those words into complete thoughts.

In considering words of the English language, we know that the words can be categorized into one of seven types (nouns, verbs, adjectives, and so on) depending primarily upon their use in a sentence. When considering binary language, the types of words recognizable by the machine are clearly defined by their use in a program and fall into the following categories.

Operation Code. Command to the machine to perform a given sequence of events (E.g., the binary bit pattern 000101000000 informs the computer: "Remove a number from core storage and place that number into a central processing unit register called the accumulator. If another number is already in the accumulator, destroy it.")

Address. Action in the computer cannot take place unless the location of an activity is defined. The address defines the location of activity during the execution of an instruction. In the example given in the definition of an instruction, the core storage cell from which data is withdrawn has a location called an address (e.g., 01010101). Similarly, if an operation code informs a tape unit to write, an associated address defines which specific tape unit is to perform this action.

Data. Arithmetic processing involves the combination of numbers. Usable numbers must have magnitude and sign and should be expressible in either fixed point (1.64,268) or exponential form (3×10^6 , 1.6×10^{-4}). The internal circuitry of the computer expresses all these factors in binary form.

Address Modifiers. It is possible to perform the same series of operations many times on different areas of the machine, without repeating the writing of the instructions, through modification of the addresses associated with the instructions. The address modifiers alter the addresses between successive executions of the instructions. This is comparable to the use of business routing forms where the originator fills in the blanks, indicating destinations of the information. For example:

Form number _____ is forwarded to _____, _____, and _____ via _____ on the following dates: _____ and _____.

The blanks represent areas of activity; the preset wording represents the commands that are unchanging. The selection of words to "fill in the blanks" in a computer is controlled by address modification words. Generally, the selection is sequential in nature.

What can man gain by grouping the computer's words together? Consider first the grouping of operation codes. The uninitiated "Chinaman" will be delighted to learn that there are only 200 or less combinations of words that can be called operation codes. He says to himself, "If I could express these A, B, C combinations in Chinese, which is familiar to me, I will have little difficulty remembering all the operation codes."

This system has been adopted by the users of computers. IBM and the SHARE organization have established a standard set of symbols to take the place of the operation code bit-no-bit pattern. This symbolism is in the form of abbreviations of the function of each code. These symbols are called mnemonic because they are reminders of the function. The following chart illustrates the point.

Machine Language	Function	Mnemonic Code
000100000000	ADD to the current value in the accumulator	ADD
000010000000	MULTIPLY the present value of the given register by a given number	MPY
000000010000	TRANSFER to a new location when the next instruction is required	TRA
000110000001	STORE the contents of the MQ register in core storage	STQ

Consider again the Oriental's problem of learning English. In the light of the previous paragraph, he has learned that groups of symbols are called words, and there seems to be no end to the combination of symbols. In order to systematize his approach to the language learning problem, he knows that he can group the words in categories depending on their usage. This grouping brings out common factors from which general conclusions can be drawn. Once the general conclusions are accurately drawn, the learning of the new language is reduced to expansion of well established facts. Unfortunately for the Oriental, English cannot be resolved into a collection of precise, arithmetic formulae. Therefore, he is always faced with modification of his general conclusions.

The man communicating with the machine, however, does not have to contend with this "grey area" of learning. After all, the machine is just a machine. Provided it is operating in the manner for which it was designed, the message delivered by a given command will be identical on the first and millionth pass. The mechanism is designed to respond in as predictable a series of events as those which occur when a simple lever is operated.

Notice that the language of the computer is highly precise in form, as is the mnemonic code. There is no need for confusion in translating from the mnemonic to binary--and yet the translation must be accomplished.

The next question is, which agent in the transaction should learn the foreign language? (It is only necessary for the Oriental to learn English in order to communicate with an Englishman. If the Englishman were also to learn the other language, a redundant effort would be made.) Based on its speed and accuracy and its untiring effort, the computer was selected as the translator.

Fortran is a program which controls the machine to accept our familiar notation and pass on the binary translation to the machine circuits. Man writes his operation codes in familiar mnemonic and the machine translates the mnemonic to binary. This translation is one of the functions of Fortran, SCAT, and other assembly routines. An important difference in Fortran is that the machine does not start with mnemonic language but rather with a special, concise Fortran language of mathematical symbols from which it generates its mnemonic and binary operation codes. It is to be emphasized here that the prime function of the Fortran system was to permit the mathematician to use his symbols rather than SHARE mnemonic codes in

controlling the machine. The word Fortran is a contraction of Formula Translation. The following diagram shows how the Fortran language relates to mnemonic.

Fortran Language	Mnemonic	Function	Machine Language
+	ADD	Add to the current value in the accumulator	000100000000
GO TO	TRA	Transfer to a new location when the next instruction is required	000000010000
X=A + B	*STO	Put the current value of A + B in storage cell X	000110000001

*The STO represents the mnemonic translation of the equal sign only.

The ultimate goal in translating the Fortran language to usable form is a binary object program. In writing the binary program, Fortran also writes its own series of mnemonic instructions. A listing of the mnemonic program is available to the programmer under sense switch control, in addition to a binary punched object program.

In review, we have considered only one part of the machine language, operation codes. Through our analysis we learned that the number of operation codes was limited. The machine assumes the burden of translation. Also, since the machine is concise in its language, we must be concise in the language we use in communicating with it. Fortran language is a form of mathematical language which meets the qualifications of precision. Fortran mnemonic instruction coding is a secondary output of the prime function of converting from the mathematical source program to the binary object program.

Let us focus our attention on the second category of machine words -- addresses. We have stated that addressing defines the area of activity of an operation code. How can these addresses be best grouped and then subjected to translating from binary to familiar language? Grouping can best be accomplished by considering core storage as opposed to non-core-storage locations.

Core storage addresses each have well defined binary "cell numbers" similar to the house numbers on a given street. The number of numbers depends on how long the street is or how large core storage is. (From the Fortran point of view, there are different programs written for 4k and 8k systems.)

Again, difficulty is experienced in communicating in the binary symbolism of the machine when referring to core storage addresses. Man has grouped his addresses into octal notation by considering groups of three binary lights to produce eight possible light patterns. By inspection, in other words, man has simply translated binary to octal notation for ease of communication (from one man to another). The combination of a mnemonic operation code and address is called an instruction; e. g. , CLA

0700, clear and add the number which is at storage location 0700. With the mnemonic notation of operation code and octal addressing, man is prepared to program the computer. He can specify precisely what is to happen (operation code) and where it is to happen (address). However, in using the octal address notation already described, the programmer is limited in his programming flexibility. Programming of the type just described (CLA 0700) is called absolute addressing because the address (0700) is inflexibly 0700. If for some unpredictable reason the data at 0700 was moved, the CLA instruction must be altered to meet the requirements of the new data location and, for that matter, any absolute address in the program may have to be altered to meet unpredictable requirements.

In order to take into consideration future needs of a given program, absolute addressing was replaced by symbolic addressing. Essentially, symbolic addressing put the bookkeeping job of assigning data and instructions to storage squarely on the shoulders of the computer. Using the preceding example, the CLA 0700 instruction in symbolic form might appear as CLA R, where R is now an arbitrary symbolic address. The machine has the capability of doling out the storage area to a given program and eventually assigning an absolute value to R once all the needs of the program have been satisfied. The assignment of core storage locations to instructions and data is called assembling. Assembly is one of the functions of UASAP 1 and 2, UASAP 3 and 4, SCAT, and Fortran. Fortran differs from the other executive routines listed because the programmer has no control over the symbols used as symbolic addresses by the Fortran program. This is a great advantage in that the programmer using Fortran need not be concerned with the details of keeping track of symbols. (Because the programmer using SCAT must make constant reference to his symbols, it is best that he have his choice of symbols for his addresses.) The symbols that the machine has manufactured for its core storage addresses will appear in the printed symbolic listing of the Fortran assembled program.

In summary, although core storage addresses represent one of the major categories of machine "words," the programmer using Fortran need not concern himself with these addresses because the machine assigns both symbolic and binary addresses to the object program.

Let us next consider the non-core storage addresses. Input-output devices all have addresses. Unlike core-storage, these devices provide the programmer with his program results. If results are to appear on tape, for example, the programmer must have control over which tape is to receive this information. The Fortran program cannot assign arbitrary symbolic and absolute binary addresses to input-output media; this assignment is the responsibility of the programmer. A family of Fortran statements is provided to specify use of tapes, card machines, drum, sense lights, and sense switches during the run of the object program. The language used is specific and closely resembles the standard SHARE coding. For example:

SHARE	Fortran Statement
RDS 302	READ DRUM 2*, 1, AB
WRS 232	WRITE TAPE 2*, AB

*The 2's in these statements are drum and tape unit addresses.

Consider next the data. Like the operation code and the address, the internal circuits of the machine respond only to the data in binary notation. However, Fortran controls the machine to accept decimal notation and to pass on the binary equivalent to the machine circuits. In addition, when required, Fortran control informs the machine to translate the binary back to understandable decimal for the printed report.

Using Fortran, data coming into the machine during the run of the object program can be alphabetic, decimal, fixed point, floating point, on tape, or on cards. No standard form is required of the data because the author of the source program describes the form with Fortran statements, called Format statements. Format statements describe the data cards according to field length, floating versus fixed point, sign, and placing of the decimal point.

Address modifiers like adverbs are difficult to describe in general terms. Their use is associated only with the modification of core storage addresses. They add finesse to the programmer's technique by relieving him of the monotony of repeating instructions used in a repetitive manner. Address modification is defined in terms of operation codes, addresses, and a new term, index registers. Index registers are central processing unit devices. Index registers enter into the discussion because they contain the amount that an address is changed during a given operation. For example: TRA 6000, 1 is an "indexed" instruction written in SHARE symbolic. In everyday terms it says "take the next instruction from location 6000 less the amount in the number one index register." Therefore, if the index register contained 1000, the transfer would take place to location 5000 instead of 6000. Address modifier words describe the contents (i. e. , 1000) of index registers and the "name" of the index register (i. e. , 1). There are three index registers in the 704 and 709, identified in SHARE symbols as 1, 2, and 4. Whenever an index register is called for in an instruction such as CLA, TRA, or STQ, the contents of the index register are subtracted from the address of that instruction. Other instructions (e. g. , TIX, TXI, AXT) alter the contents of the index registers. Simply by using the notation already introduced in operation coding and addressing, a system of address modification can be built into a UASAP 1 and 2, UASAP 3 and 4, or SCAT program. How does Fortran handle address modifiers? Easily understandable Fortran statements are translated to address modifiers. Consider the Fortran "DO" statement. This statement permits the programmer to repeat a series of commands. For example:

External Formula Number

1	DO 25 I = 1, 10 A(I) = B+C R(I) = A*D
25	PRINT 1 . A(I), R(I) STOP

The first statement (1) indicates the address modifiers. It says "the object program should execute the following statements up to and including the one labeled 25. It should repeat this process from the time that I has an initial value of 1 up to and including the time that I equals 10, each time through the program increasing I by a factor of 1." The contents of the index register would initially be 10, and indexing instructions would be built into the object program to reduce these contents by 1,

each pass through the program. The selection of which index register (1, 2, or 4) is to be used in the object program is the choice of the Fortran program. A series of SCAT coded instructions to accomplish the same result would require several dozen instructions.

In conclusion, the Fortran system contains a concise language, mathematical in background, employing familiar symbols which are readily translated to machine binary coding. The programmer need not learn anything more about the machine than the limitations placed on his mathematics by the physical size of the machine and some statements governing the input-output transmission. In the run of the Fortran executive routine, Fortran language is translated to SHARE mnemonic and relocatable binary cards. The SHARE symbolic operation code listing is only secondary to the machine control. Addressing of core storage is symbolic and is entirely under control of the machine. Input-output addressing is under option of the programmer. Data can be of any decimal form, as is described by Format statements in the program. Address modification is employed by the Fortran program but is given to the programmer in easily understood language. The selection of index registers and indexing instructions is made by the computer under control of the Fortran translator program.

1.02.00 GENERAL ORGANIZATION OF TRANSLATOR

In all three Fortran modes (I, II, and 709), the translator is described in terms of six sections. Each section is distinct in purpose. Section 3.00.00 of this manual describes, in programmer's terms, the functions of each section of the translator. A digest of Section 3.00.00 follows in Section 1.02.

1.02.01 Section I

In this translator section, source cards, punched in the Fortran language, or BCD tape containing the source information, are read into the system. If card input is used, the information on the cards is transmitted to tape 2 in BCD. If tape input is used, tape 2 is originally set up with the source program in BCD.

With tape 2 containing the source program, the translator proceeds to code all the statements in the source program. Every statement receives a code number, called the "internal formula number" (IFN). These numbers are assigned in sequence starting with 1. All internal references to the original statement are made using IFN as identification. The IFN is not to be confused with the "external formula number" (EFN) punched in the source program cards. Input-output statements receive more than one IFN.

The scanning of the BCD file on tape 2 occurs only once. All information contained in this file must be coded as it is being read. The information will take one of two forms: compiled instruction tables (CIT) or non-CIT. (The CIT format is described in detail in Section 2.03.00. It is a standard form, and the final form of all Fortran statements and data before they are compiled in Section 6). The CIT

information is stored temporarily on tape 3 in Section 1. This information is erased during Section 1' and is stored on tape 2, file 2. The CIT contains the results of analysis of all arithmetic statements. This analysis is completed in Section 1. It is the most important part of the Fortran program in that it accomplishes the translation between source arithmetic statements and machine instructions.

All arithmetic instructions written in this section are written in CIT form and placed in a record called the COMPAIL file (Complete Arithmetic, Input-output, Logical).

The non-CIT information is stored in buffer areas temporarily in core storage. When the buffers are full, their information is transferred to tape 4. The buffer units are ten words long and located in lower storage.

1.02.02 Section 1'

Section 1' places the CIT information of tape 3 on the second file of tape 2, arranges the tape 4 tables in order, and stores the information in tape 2, files 3, 4, and 5.

1.02.03 Section 2

Section 2 compiles the instructions associated with indexing that result from DO statements and the occurrence of subscripted variables. These instructions are placed in the COMPDO file in CIT form.

In this section the program assumes that there are many index registers in the machine. Indexing instructions are going to be generated through the use of: Arithmetic statements. For example, $I = N + 1$ where N is a subscript.

The contents of an index register will be increased by one.

DO statements. For example, DO 10 I = 1, 5. Here, statements up to and including 10 will be repeated for all values of I from 1 to 5. To the reader familiar with programming, this statement implies the use of indexing instructions to accomplish the given result. The handling of indexing becomes more complex where DO statements fall within DO statements to create "DO nests."

Combination of arithmetic and DO statements.

1.02.04 Section 3

Section 3 merges the COMPAIL and COMPDO files into a single file, meanwhile completing the compilation of non-arithmetic statements begun in Section 1. At this point the object program is complete but it has been assumed that the 709 is a machine with an unlimited number of index registers.

1.02.05 Section 4

Section 4 carries out an analysis of the flow of the object program to be used by Section 5. The running of the object program is simulated in order to do this. The object program may be run several hundred times in this section.

1.02.06 Section 5 and 5'

Section 5 manipulates the symbolic tag information so as to write the object program with the three actual index registers of the 704-9. The processing is very complex; however, certain tabled information of interest is generated during the run. All major decisions concerning index registers are recorded in the predecessor (PRED) table, and a table of all tagged instructions (STAG) is recorded. These are one-word entry tables which remain in core storage.

Section 5' is generally considered a bookkeeping section. Up to this point the constants involved in the object program have been stored in tables. Since all information from the translation must get to the object program in compiled instruction tables (CIT), a departure is made from the normal sense of CIT's in that the constants are also transmitted in this form. In the place of an instruction code, the SAP mnemonic for octal data (OCT) is entered into the decrement field of the second word of the CIT entry. In this form the information from the following tables is recorded in the object program: FIXCON, FLOCON, ASSIGNED CONSTANTS, FORMAT and UNIVERSAL CONSTANTS.

1.02.07 Section 6

Section 6 assembles the object program, producing a machine language program on cards or tape ready for running. The object program can also be produced in Share symbolic language, if desired.

1.03.00 NUMBER OF INSTRUCTIONS IN EACH SECTION

There are approximately 24,000 instructions in the Fortran executive program. The number of instructions in each section of Fortran I are as follows:

Section No.	No. of Instructions
1	5,500
2	6,000
3	2,500
4	3,000
5	5,000
6	2,000

1.04.00 FORTRAN SYSTEM TAPE

1.04.01 General Organization

The complete Fortran translator program is written on the Fortran system tape. The program is written and executed in sections, each section consisting of a number of variable-length records. An individual record is called into core storage from the system tape when it is needed.

Versions of the Fortran translator differ according to size of the 704 used to run the program. The most common version is designated 4-1-4-1. This program requires a single 737 (4k words), a single 733 Drum (8k words), and four 727 tape units. There is also an 8-1-4-1 version almost identical to 4-1-4-1 except that it requires a

704 with two 737's providing 8k storage. For large programs it will run considerably faster than the 4-1-4-1 version. There is another modified Fortran version for use on 704's with the 738 (32k words and no drum).

The system tape has three files. The first two files are the executive routine or the system proper. The third file is the library (Figure 1). File 1 contains a special first record called 1-CS and 7 other ordinary system records (Figure 2.) The end-of-file mark is not considered to be an ordinary system record. File 2 of the Fortran system tape consists of ordinary records numbered 8 through 67 (Figure 3). These records are not called into core storage and executed in sequence, but are executed in the order shown in Figure 4. First of all, the special program 1-CS, the first record (number 0) in file 1, is loaded. Then 1-CS reads in record 1. This is the first executable record of Section 1. It is called the "card-to-tape simulator." It reads the source program from the card reader. This source program, consisting of Fortran statements, is converted to BCD and written on tape 2. The EOF signal from the card reader causes the skip to the beginning of file 2. Records 8 through 67 are then executed in sequence. The following list specifies the records of file 2 and the section of which each is a part, for Fortran I.

Section No.	Records Included
1	8-20
2	21-36
3	37-41
4	42-48
5	49-54
5	55-67
6	2-7

1.04.02 Fortran I Ordinary System Records

The first word of each system record is a check sum for that record; it is placed in location 2 by the 1-CS program (Figure 5). The second word of each record is a control word which is placed in location 17_g. The address field of the control word gives the first storage location into which the first step of the program is to be stored. This location is called the "load address." The remaining words of the program will be stored in consecutive locations above the first location or load address.

The decrement of the control word contains the address to which control is to be transferred after the record has been completely read (Figure 6). This address is referred to as the "TRA address."

The information stored in location 17 can be displayed to find out the last TRA and LOAD address handled by the 1-CS program. This information could be helpful if the machine "hangs up" somewhere in the running of the Fortran program or comes to a halt at some storage location not in an error stop listing. Table I shows the TRA and LOAD addresses for the various records of the Fortran program, providing a reference for determining what record or what area of the program is failing. By displaying location 17_g and using the table, the particular record and section where the stop is encountered can be identified. This does not necessarily mean that the error is in this particular record, since the trouble could be introduced earlier and not indicated at that time.

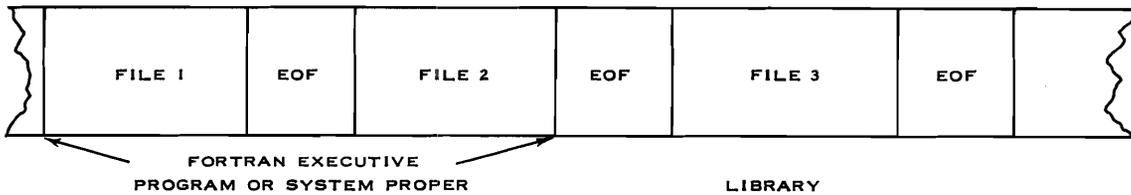


Figure 1. Fortran System Tape

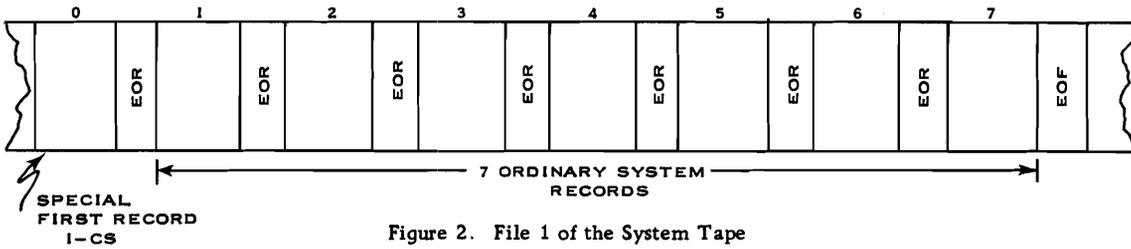


Figure 2. File 1 of the System Tape

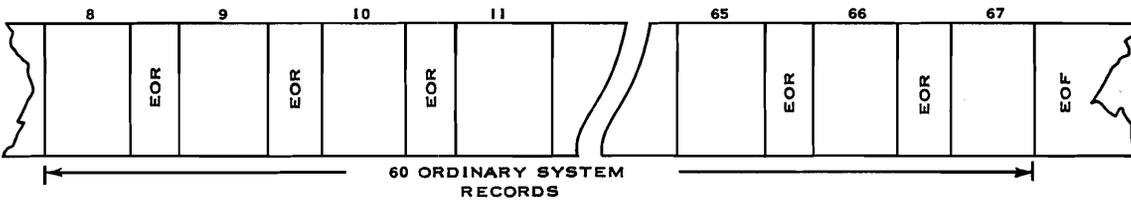


Figure 3. File 2 of the System Tape

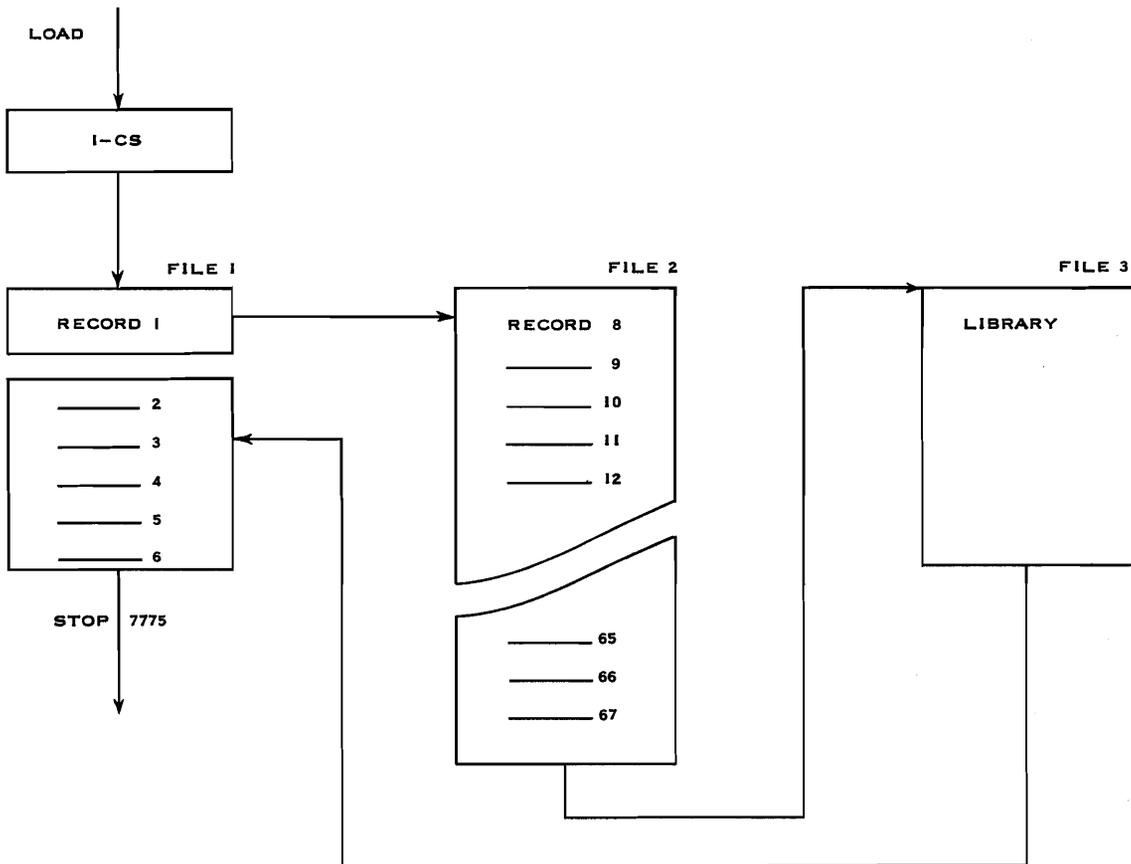


Figure 4. Execution Order of the Records

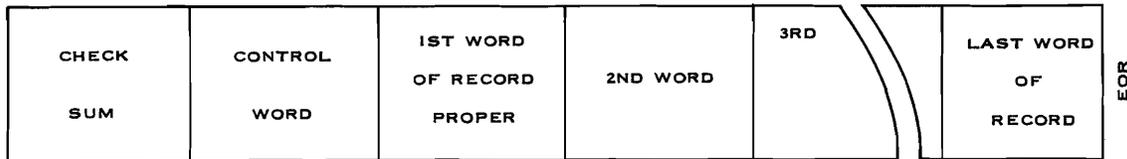


Figure 5. Ordinary Fortran Record

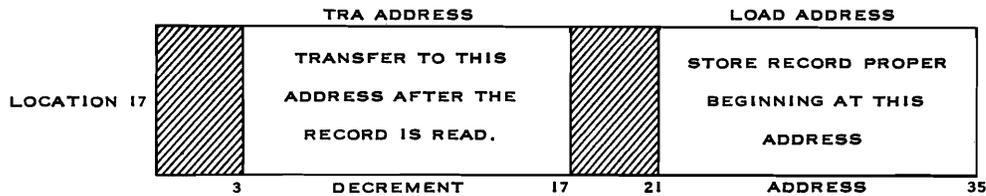


Figure 6. Record Control Word

After record 67 is loaded, a search through the library, file 3, follows to incorporate any library functions into the compilation. When this search is completed, the system tape is rewound and the first two records of file 1 are skipped. Records 2 through 7 are loaded and executed. These six records comprise Section 6 of the Fortran program. At this point the Fortran run is complete. This arrangement of the program into records on tape requires a program within the machine that allows the next record to be read when it is signaled that the previous record is completed. This program is called 1-CS and is described in detail in the following section.

1.04.03 1-CS (Tape 1 to Core Storage Program)

It is important from a service standpoint for the customer engineer to understand the 1-CS self loading program. This short program is relatively simple but it is the key program in monitoring the progression of the Fortran translator. With a knowledge of the loader the customer engineer can better determine which record of the translator is failing. A listing or a map of the translator will have to be used by the customer engineer to isolate the subroutine of the record that is giving trouble.

Figure 7 is a block diagram representing the loading of 1-CS into storage. After the load-button sequence and bootstrap, the tape record monitor is loaded in descending sequence from locations 27 to 4. The COPY instruction at location 1 and the TXI at location 2 accomplish this loading. The copy loop stores COPY 3 at 3 and LTM at 2. The LTM is performed and the COPY 3 stores BST at location 3. The program control enters the monitor proper at location 4. Figure 8 shows the operation of the monitor in block diagram form.

1.04.04 The Tape Record Monitor

The program in storage from 4 through 27 reads in, sequentially, all the records of the systems tape (tape 1). The records are check sum and redundancy checked. If no errors exist, the program proceeds into the record that has read in. The check sum, which is the first word of each tape record, is stored at location 2. The record control word, which is the second word on tape, is stored at location 17. The record control word's address is the address used to store the first instruction (third word) of the record currently being read in. The record control word's decrement is the location of the first instruction to be executed after the current record is read in. These are not relative addresses but absolute locations. Refer to Figure 8.

1.04.05 Listing of 1-CS on Tape

The following listing of 1-CS shows the instruction sequence on tape and the locations where the instructions are stored.

Sequence On Tape	Storage Locations	Operation	Address, Tag, Decrement
0	0	LXA	0, 1
1	1 (originally)	CPY	2, 1
2	2 (originally)	TXI	1, 1, 1
3	1	CPY	31, 1
4	27	HTR	3
5	26	TZE	0
6	25	COM	6
7	24	ACL	2
10	23	COM	6
11	22	TRA	27
12	21	RTT	12
13	20	WRS	333
14	17	HTR	0
15	16	TXI	15, 1, 77777
16	15	CAD	0, 1
17	14	CAL	17
20	13	STA	26
21	12	ARS	22
22	11	STA	15
23	10	CAL	17
24	7	CPY	17
25	6	CPY	2
26	5	RDS	221
27	4	LXD	27, 1
30	3 (originally)	CPY	3
31	2	LTM	7
32	3	BST	221

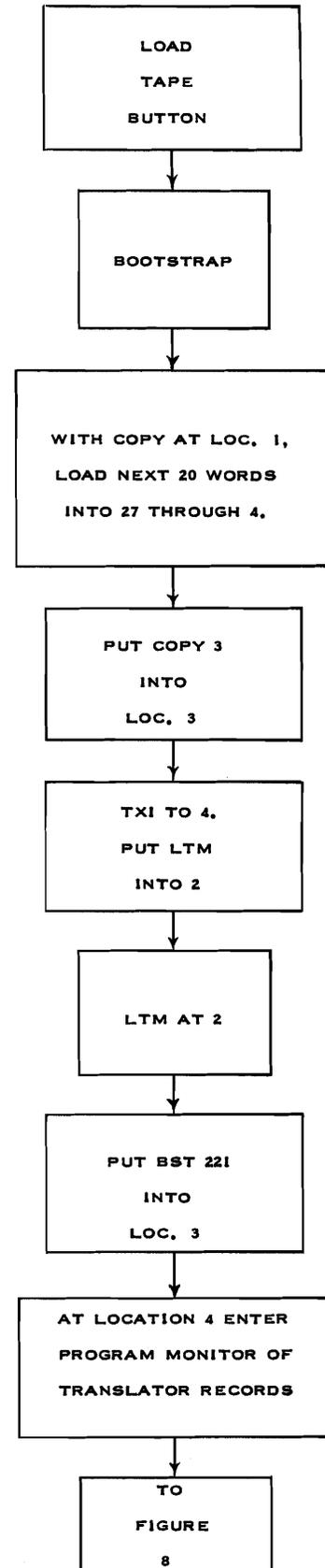


Figure 7.

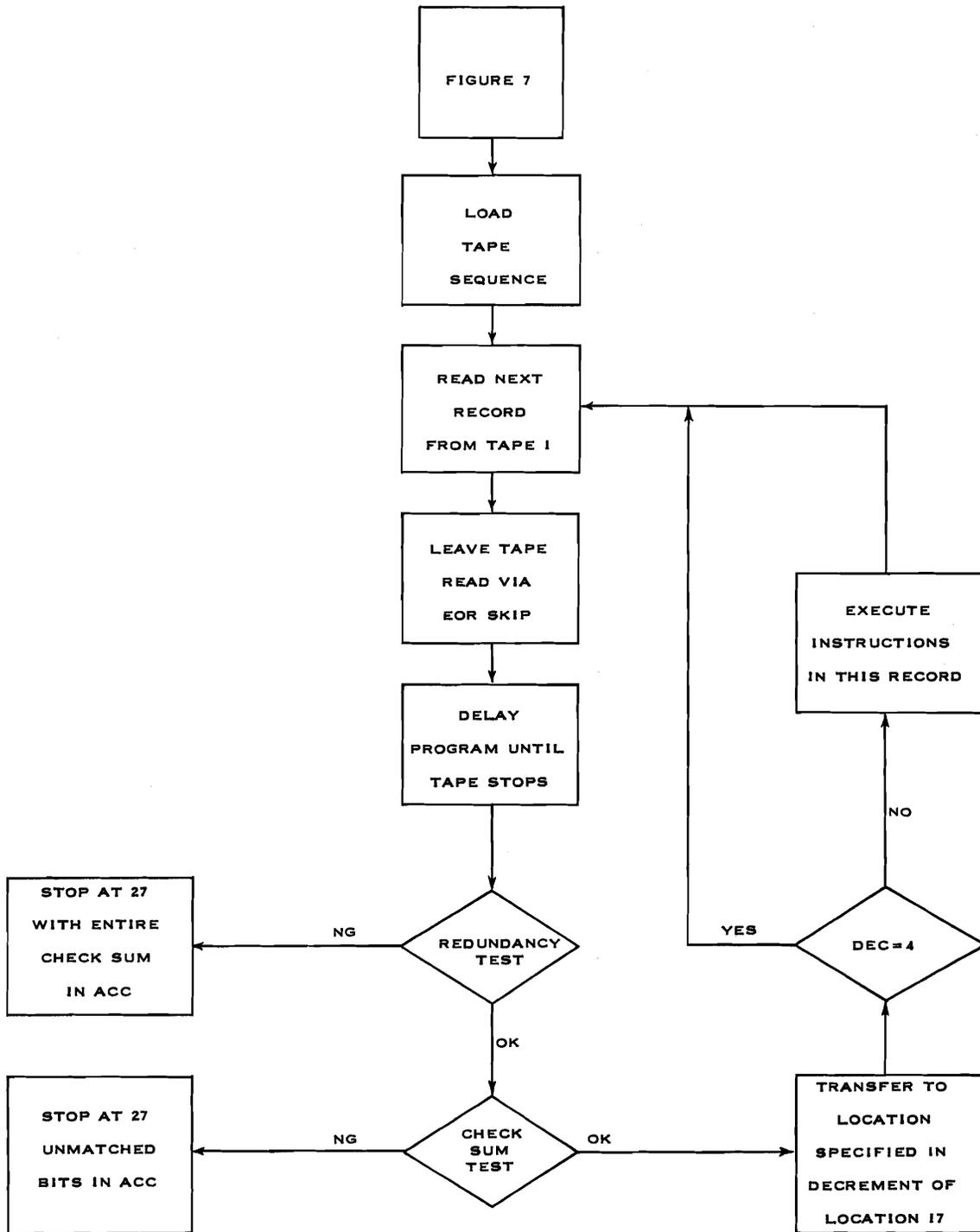


Figure 8. 1-CS, Tape Record Monitor

1.04.06 Listing of 1-CS in Storage

The original boot-strap loading loop in locations 1,2,3 is wiped out by the self-loading 1-CS. After loading, the program is located in core storage as follows:

LOCATION	OPERATION	ADDRESS TAG, DECREMENT	COMMENT
0	LXA	0, 1	Zero to XRA initially
1	CPY	31, 1	Loads most of 1-CS
2	LTM	7	
3	BST	221	Backspace to reread record
4	LXD	27, 1	Clear XRA before reading next record of Fortran
5	RDS	221	Read next record
6	CPY	2	Check sum of record ---- 2
7	CPY	17	Control word -----17
10	CAL	17	Control word -----Acc
11	STA	15	Where record is to begin in storage
12	ARS	22	DECRE _{Acc} ----ADR _{Acc}
13	STA	26	Transfer address -----26
14	CAL	17	Again control word -----Acc to start computing check sum
15	CAD	0, 1	Copy record & compute check sum
16	TXI	15, 1, 77777	Repeat & store record in con- secutive ascending addresses
17	HTR		Control word location
20	WRS	333	Write delay
21	RTT	12	Test for redundancy error
22	TRA	27	Recognize redundancy error
23	COM	6	Check sum in Acc
24	ACL	2	Bring in transmitted check sum to be compared
25	COM	6	
26	TZE		Check sum OK, go to TRA address
27	HTR	3	Read error

1.04.07 Control Words of 1-CS Monitor

Table I shows a listing of all the control words used by the monitor. An understanding of this listing will give you access to the area of Fortran that is causing an error stop. The control word for the record currently in process is stored in location 17. Suppose, for example, that the machine stopped at an unlisted stop during the Fortran run. A display of location 17 showed a decrement of 335 and an address of 110. In Table I you will find that all the control words are unique and that the one in this example indicates the stop occurred during the run of Section 1, while record 1 of the first file on the tape was being run. Also indicated in the listing is the first instruction stored in this record (REW...222). Notice that this is not the first instruction executed in this record. The first instruction executed in this record is located at the address indicated by the decrement of storage location 17. In the present example this is an RDS...321 instruction located at 335. With a dump or listing of the Fortran tape you should be able to follow the sequence of instructions from that starting point. The last column in the list indicates the high order storage location used in the storing of the current record. Notice that the load address and last address of the records indicate that during the assembly many core storage locations are rewritten many times. For instance, records 11 and 12 of Section 1 are both stored starting at 3440 and ending above the 50000 area. This may make tracing difficult where only portions of a previous record are erased in order to store the shorter current record.

On a listing of the Fortran system tape you may have difficulty locating the first executed record if you do not know the organization of the list. Most lists of tape records indicate the order of full 704 words in the record starting with zero. Recall that the "zeroth" word in the record is the record check sum and the "first" word is the control word that is stored in location 17. The first ordinary word of the record proper is the "second" word. Since the order of instructions is relative to a starting point of zero, how do you find in the listing, the first executed instruction whose absolute location is 335? To find the relative address in the list use the following formula:

$$\text{Relative Address} = \text{decrement of 17} - \text{address of 17} + 2$$

In the example of the previous paragraph, the relative address in the listing of the RDS 321 instruction is:

$$\text{R.A.} = 335 - 110 + 2 = 227$$

The corresponding listing of records on the Fortran II Systems tape is given in Table II. It follows the same general organization as the information in Table I for Fortran I.

1.05.00 UPDATING THE EDIT DECK

Additions, deletions, and changes in the list of library functions can be made by means of the Fortran librarian, FNLIB 1. Each time the librarian is used it rewrites completely the list of functions; hence it should be followed by all the routines which the system is to contain.

Each routine consists of one or more control cards, followed by the routine proper on relocatable binary cards. The routine proper must meet the specifications given on page 40 of Form 32-7026.

The control cards are punched as if for loading by NYBL1. The loading address (9L address) must be zero, and the check sum must be given. The first control card has in its 8L address the number of locations occupied by the subroutine, and in its 8R address the 2's complement of n, where n is the length of the common storage region used by the routine. Succeeding rows have in the left word a function name (without the terminal F) followed if there is room by a blank character and zeros in internal 704 BCD with the significant characters packed to the left, and in the address of the right word the corresponding entry point into the routine, relative to zero. For example, the control card for the UASC--1 routine, which can calculate either cosine or sine by entering at relocatable 0 or 1, has COSb00 and 0 in its 7's row and SINb00 and 1 in its 6's row. If there are too many function names to fit on a single control card, they may be continued on additional control cards. On these additional cards do not repeat the information given in the 8's row of the first control card.

The entry point which will cause the specifications for a library routine to be met can be given a function name (or several names if desired). Such names can be distinguished as primary or secondary names by not prefixing, or prefixing, the entry point with a minus sign (punch in column 37 of the appropriate row of the control card). The meaning of primary and secondary arises from the following rule of precedence which is used by the Fortran system in compiling library routines into the object program.

RULE. When a function is mentioned in a source program, the routine which will be used is the first routine on the system tape which meets either of the following conditions: (1) the name mentioned is a primary name of the routine; or (2) the name mentioned is a secondary name of the routine, and at least one of the primary names of the routine is also mentioned. (If no such routine exists, the universal empty routine HTR 1, 4 is compiled).

If the system tape is arranged with the routines which have many secondary names preceding the routines with few or none, this rule will prevent unnecessary duplication of routines in the object program. Suppose, for example, that the system tape contains an arc sine routine which also has an entry point which will compute a square root, and that this routine is given two names, ASINF (primary) and SQRTF (secondary). Suppose also that later on the tape is an ordinary square root routine with the single name SQRTF (primary). Then a source program which asks for both ASINF and SQRTF will cause compilation of the former program only.

In addition to the updating of the edit deck, your listing in this manual of the records on the master tape should be maintained up to date. In order to do this properly you must have three documents:

1. The memorandum describing the change.
2. The deck of cards which produce the correction.
3. Either a dump or SAP listing of FORTRAN.

Referring to Table I, the following items are required:

Required Information	Where Found
Record number	First three digits of *000 correction card
Description of record	Usually not changed
Transfer address	8L decrement of 000 correction card
Load address	8L address of 000 correction card
Last address	8R address of 000 correction card
Contents of transfer word	SAP listing or dump of Fortran
Contents of load word	SAP listing or dump of Fortran
Section No.	Memorandum

*The 000 correction card is the first card in the correction deck; the three zeros are the last three digits of the card number.

1.06.00 FORTRAN TAPE ASSIGNMENT

Figures 9 and 10 represent the contents of the tapes in Fortran processing. They indicate the contents of all four tapes at the end of each of the six sections. With the exception of the first four records of file 5, tape 2, the assignment of tapes is identical in all Fortran modes. (Fortran I processing does not develop these four records.)

1.07.00 RELATING TAPE ASSIGNMENT IN FORTRAN I AND II TO FORTRAN 709

Fortran I and Fortran II utilize the first four tapes identically. Fortran II, however, uses additional tape units 5, 6 and 7. They have the following function during batch compiling:

- Tape 5: The programs to be batch compiled are recorded on tape, separated by END statements. If the input is card on-line, all card information is transmitted to tape 5. If the input is off-line, tape 5 is used as the off-line input.
- Tape 6: This becomes the output tape for batch compiling. All source programs in Fortran language, storage maps and SAP mnemonic outputs appear on this tape. This tape receives the output information for all object programs that previously were consigned to tape 2.
- Tape 7: This tape recovers the binary object programs of the batch-compiled source programs. This is, in Fortran I, the responsibility of tape 3.

Sect. No.	File 1	File 2	File 3	File 4	File 5	File 6	File 7	File 8	File 9	File 10
1'	SOURCE PROGRAM (BCD) All others in Binary EOF	COMPAIL EOF	COMPAIL RECORD COUNT and FORSUB(1) EOF	FLOCON FORMAT SIZE EOF	10 END 11 SUBDEF (1) 12 COMMON 13 HOLARG 0 TEIFNO 2 TIFGO 3 TRAD 1 TDO 6 FORVAL 5 FORVAR 4 FORTAG 7 FRET 8 EQUIT 9 CLOSUB EOF (3)	Not in Fortran I				
2							DOTAG B Rec. Count EOF	DO FILE C	DO FILE C RECORD COUNT	
3								ASSIGN CONSTANT EOF	FIXCON EOF	
4										
5										ASSIGN CONSTANT EOF
5'										
6										

Notes: (1) If no entries exist, no identification exists either. (2) Also includes new FUNCTION statement. (3) Identification label number of the tape tables.

FIGURE 9. FORTRAN TAPE 2

End of Section	Fortran Tape 3				Fortran Tape 4	
	File 1	File 2	File 3	File 4	File 1	File 2
1					Tape Table Entries	
1'	1. Non Exc. 2. T Stops					
2					TRALEN	COMPDO
3					CIT's 100 WDS/REC	Closed Subroutines
4	CIT's from Sec. 3 (erased)	EMPTY	Tag List ? 5 WR/REC	BB List	CIT's	Closed Subroutines
5	CIT's					
5'	CIT's					
Pre-6					CIT's	
6	Binary Output					

FIGURE 10. FORTRAN 709

1.09.00 USING THE FORTRAN SYSTEM TAPE

Set the system tape to logical 1, and set two machine tapes to logical 3 and 4. If operating with off-line input, set the input tape (bearing the source program as the first file) to logical 2; otherwise set a machine tape to logical 2.

At the 704 card reader, load the one-card Fortran system caller FN_{SC}1, followed (if the input is on-line) by the source program deck. Do not use extra blank cards.

(If tape 1 is known to be rewound, FN_{SC}1 is not necessary. With off-line input, simply press LOAD TAPE and, when the card reader is selected, press START on the card reader. With on-line input, ready the source program in the card reader and press LOAD TAPE.)

Place the SHARE printer board 2 in the 704 printer. Set the sense switches as follows:

- Switch 1. UP to obtain the object program as a binary tape (tape 4) and as a deck of binary cards.
 DOWN to obtain binary tape (tape 4) only.

- Switch 2. UP to produce on tape 2 two files containing the source program and a map of object program storage.
 DOWN to add a third file to tape 2, containing the object program in the language of the forthcoming modified SHARE symbolic assembly program.

- Switch 3. DOWN to list on-line the first two or three files of tape 2, depending upon whether switch 2 is up or down.

- Switch 4. Up or DOWN to cause on-line listing to be single or double-spaced.

The program ends by executing a load button sequence to the card reader. If the card reader is not ready, the machine will hang up at location 77775₈; if it is ready but empty the machine will stop at 77777₈.

1.10.00 RUNNING THE OBJECT PROGRAM

The binary deck that is produced when switch 1 is up consists of the object program in relocatable binary, together with the four-card Fortran relocating loader UA CSB3 and appropriate control card and transfer card. The binary deck is thus ready for immediate loading and execution. For further details see the forthcoming SHARE write-up for UA CSB3.

Details about using the binary tape form of the object program will be announced later.

The printer board to be used with Fortran object program is SHARE 2.

There are nine standard error stops in object level input-output routines. They are to be recognized not by looking at the instruction counter but by looking at

the HPR instruction itself in the storage register.

HPR	0,0	End of file in reading binary tape. Press START to resume reading next file.
HPR	0,1	End of file in reading cards or BCD tape. Press START to resume reading next file.
HPR	1,1	Inappropriate character encountered in a data field in reading cards or BCD tape. Pressing Start causes that character to be treated as a zero.
HPR	2,1	
HPR	3,1	
HPR	4,1	
HPR	0,2	Non-Hollerith character encountered in reading card. Correct card, ready in card reader, and press START.
HPR	0,3	Redundancy check in reading BCD tape. Press START to accept information read.
HPR	0,4	Echo check in printing. Press START to continue. Press RESET and START to repeat line, and continue.

1. 11. 00 DESCRIPTION OF TAPE AND DRUM USAGE DURING A FORTRAN I RUN

With sytem tape rewound on tape drive 1 and the source program ready in the card reader, hit the load tape button. The special first record on the system tape is stored in 0-27. The second record on the first file of the system tape is a card-to-tape simulator that puts the source program on tape 2 in BCD. If the card reader is empty, this program is not executed since the source program is assumed to be on tape 2 already. In either case the EOF signal from the card reader causes the Fortran system tape to skip to the beginning of the second file. The records in file 2 are executed sequentially. Each record is a program or part of a program. In some cases these records are copied to drum and used over again.

During the first section, tape 2 is read, using programs copied from tape and some stored temporarily on drum. All four logical drums are used. Tape 3 is also written during this section. Tables of information are written on drums and tape 4. At the end of this section, the tables from the drums and tape 4 are written as additional files on tape 2. If the programmer has made errors which the Fortran diagnostic can catch, these errors are printed on line at the end of Section 1.

The second section processes table information from the drums and tape 2 (additional files) and writes a new tape 4; it also writes tables on drums.

The third section merges the tapes 3 and 4 using another file on tape 22 intermediately, and finally prepares a tape 3.

The fourth section examines this tape 3 and writes tables on drums and in core.

The fifth section writes a tape 4, and tables on drums, then uses the tables from section 4 and tape 4 to write a new tape 3.

The last section takes the tables from the additional files on tape 2 and writes them on the drums, then, using these tables, writes the first part of the output storage map as a new second file on tape 2. Next the first pass of a special assembly program is made over tape 3. Then a library search is made. The library is the third file on the Fortran tape. When this is finished the Fortran tape rewinds, spaces over the first two records on the first file, and makes the second pass of the assembly. The binary output from this assembly is written on tape 4. More of the storage map is written on tape 2.

If sense switch 1 is up, contents of tape 4 are punched on line; if not, no punching takes place.

If sense switch 2 is down, a third file is written on tape 2 (obtained by translating tape 3). This is the symbolic version of the "object" program.

If sense switch 3 is down, tape 2 is rewound and printed on line.

Finally all tapes (1, 2, 3, 4) are rewound and the load button sequence is executed. The words "end of diagnostic" are printed, tapes are rewound, and the 704 stops at 17777.

1.12.00 SERVICE AIDS

To successfully run the Fortran translator the 704 must be in prime working order. The tape system in particular, and the drum are given a good work-out during the execution of the program. The following items are listed to assist you in analyzing troubles.

1. Mechanical adjustment of 727's is very critical as far as tape creep goes. Creep can be recognized by observing a mark on tape (e.g., load point during a multiple write-backspace-write operation. The tape should tend to creep forward (i.e., lengthening the inter-record gap). Care must be taken on 704 systems to keep this forward creep to a point where 20 to 25 passes are possible before the gap becomes too large. The customer is allowed to program up to ten passes.

2. When using Model I 727's the belt tension for capstans must be correct. Refer to page 55 of 727 Customer Engineering Manual of Instruction (Form 223-6681).

3. Tape operation may be improved by installing the capstan drive shaft change, E.C. 242797, B/M 561995, on a Model II 727.

4. The recovery time of the "write load point delay SS-753" has given trouble. Refer to CEM 803 for correction.

5. If you have installed B/M 562270, note:

1. Item 4 of CEM 787.
2. CEM 815.

These modifications eliminate the necking down of the start envelope that appeared about 12 ms after go.

6. Noisy pre-amps have caused record trouble on Fortran and the SHARE assembly program. Refer to CEM 234; recommended frequency is two weeks if trouble has been encountered.

7. A 100 uu capacitor must be connected from the output of AND circuit E to ground on Systems 5.11.05 to prevent errors on "read printer." E.C. 243552 adds this capacitor (704).

8. E.C. 242469 must be installed to prevent drum LDA trouble on Fortran. During the installation of E.C. 242469, wires may have been left out from MF 1-617-1 to MF 1-G29-1 and from MF 1-F30-2 to MF 1-G29-2. Check your machine.

Keep in mind that some of the stops recorded in operator's manuals are misleading in their explanations. It is possible that a drum check sum error may be indicated by the listing but the actual trouble may be due to tape creep. Some stops are ambiguous in this respect. Make sure you have the latest listing of error stops to obtain the greatest assistance in analyzing machine operation and stops.

1. 13. 00 FORTRAN II REFINEMENTS

From a user's standpoint, there are generally two areas where the present Fortran system needs improvement. First of all, the system requires improved facilities for debugging source programs. Secondly, it needs better facilities for creating and using subroutines. An improved version of the system called Fortran II will contain improvements in these areas. This version is to be available in the not too distant future.

The main improvement contemplated as far as debugging is concerned, is the incorporation of a general and expandable diagnostic procedure in the translator. This procedure is designed to find and print a description of every detectable error in a source program. Thus, stops in translation will be eliminated or reduced to a minimum. The description of the failure will be documented in such a way as to eliminate ambiguity. The print-outs will include useful information in locating trouble and will be a definite aid to the customer engineer as well as the customer.

A variety of routines that are helpful in debugging object programs during execution can be added to the library of Fortran. The routines can be included in the object program during translation by making use of the improved subroutine facilities added to Fortran II. These new facilities are provided by six new statements that are added to the Fortran language.

1. 14. 00 ANSWERS TO COMMON GENERAL QUESTIONS INVOLVING FORTRAN

1. Why is Fortran used and what are its advantages over the SHARE assembly program?

Fortran allows a programmer to write in relatively familiar and simple language the steps of a procedure to be carried out by the 704. The programmer need not know 704 language, and is relieved of clerical work; human error is minimized. The programmer writes in symbolic machine language in SHARE. Fortran translates, compiles, and assembles, whereas a SHARE assembly program essentially just assembles, although subroutines can be compiled from the library tape of SHARE.

2. Can Fortran be used for logical problems as well as mathematical problems?

Fortran is essentially designed for mathematical problems. It can be used for logical problems. However, it wouldn't be as efficient in input/output as a manually coded program.

3. What are the minimum machine requirements for using Fortran? Can the library routines used during the translation be contained on more than one tape?

Fortran will run on the minimum 704 defined by the SHARE organization: Single 737, 733, four tape units, floating point, and the Copy Add and Carry Logical word instruction. There are variations of Fortran for 704's with larger core storage with or without a drum. As Fortran now stands, any library routines to be used during translation must be included in file 3 of the Fortran system tape.

4. What size of program will it write and how can it be determined whether the object program is too large for the particular 704 system?

There is no definite limit as to the size of an object program that can be written by the Fortran system. It is possible to exceed the capacity of a 32k core storage. It is not too difficult, therefore, to write a program that will exceed 4k storage. This is an aspect of Fortran that is bothersome. The size of the object program can be estimated by a method described on page 60 of Form 32-0306. Other limits regarding the size of the source program are given on page 44 of Form 32-7026.

5. What does the 704 system do during each of the six phases or sections? Which tapes are used?

The general description of the six sections of the Fortran translator are included in Section 1.11.00 of this manual. An explanation of the machine operation during the six different phases is summarized in Section 3. There may be an exception but generally all four tapes are used in every section.

6. How are the Fortran statements translated into 704 language? Does each statement set up a standardized loop?

A brief description of the Fortran translation is contained in Section 1.02.00. In general, each subroutine of the output program has been constructed, instruction by instruction, so as to produce an efficient program. In addition each subroutine is constructed to fit efficiently with adjacent subroutines and the over-all program. Fortran written programs seldom contain sequences of even three instructions whose operation parts alone could be considered a precoded "skeleton."

7. How can the program be restarted after an unlisted stop?

There is no convenient restart point for running Fortran after an unlisted stop. Essentially, it means starting over at the beginning. If all the source program has been read from cards successfully, then the cards do not have to be reread. The source program will already be on tape 2 in BCD. The starting procedure

is the same as if the source program were put on tape 2 after being prepared on off-line equipment.

8. Can the 704 be put into an error loop so that the cause of machine trouble can be located?

No, there are no facilities in Fortran that permit looping on an error condition. It is possible, by altering the program manually, to accomplish this, but it would be time consuming. It may be necessary in some instances.

9. What do each of the 32 Fortran statements accomplish and how are they used?

The Fortran language is completely covered in Form 32-7026. It is recommended that the beginner start with Form 32-0306 which provides a very sound introduction to the Fortran language.

10. How would an operator go about running the object program?

The complete details regarding tapes, sense switches, cards, and so on, related to running the Fortran program or the object program, are in Section 1.09.00 of this manual.

2.00.00 GENERAL FORTRAN-RECORD STRUCTURE

THE TWO PRINCIPAL forms that the object-source program takes in its examination by the translator are CIT's (compiled instruction tables) and tables. Both forms are described in detail in the Section 3.00.00.

There is common information which should help in integrating the tables. For instance, file 5 of tape 2 contains many tables. The first word of all these tables is a label number which is used in Section 1' to merge together all common information. The second word in the records of this file is a count of the number of words in the record.

Fortran I does not have any of the first four records of file 5, tape 2 of Fortran II and Fortran 709. Therefore, the record number of a record in Fortran II will be four more than the corresponding number in Fortran I.

Any reference to storage on drums implies the use of either Fortran I or II. If 709 Fortran is under consideration, upper core storage will be used in place of the drum.

2.01.00 CALLFN RECORD

The CALLFN record is a table of IFN's presented in CALL statements. (See page 16-18 of Form C28-6000 for a full explanation of the CALL statement). Each entry into the table requires only one full word. The decrement of the word contains the IFN of the first variable in the CALL statement; the address contains the IFN of the last variable in the CALL statement.

2.02.00 CLOSUB RECORD

The CLOSUB record is a table of closed routines called for in the source program. The entry is made for each library subroutine called for in the source program. The entry is the BCD representation of the names of these closed subroutines and demands only one word of storage because six BCD characters are sufficient to hold any of the names of the closed subroutines. (See page 8 of Form 32-0306.)

The table entry is SQRTF for the following statement:

$$\text{ROOT} = (-B \text{ SQRTF } (B^{**2} - 4. * A * C)) / (2 * A)$$

The CLOSUB table is stored during the run of Section 1' on tape 2, file 5, record 14 in Fortran II and 709 or record 10 of the same tape in Fortran 1. The first word of the first record is the label number, 9. The second word is a count of the number of words in the CLOSUB table.

2.03.00 COMPILED INSTRUCTION TABLES

By the end of Section 3, the object program is completely compiled in symbolic form (with the exception of library subroutines). The compiled instructions, and later all constants, must be placed in a table which is translatable by the compiler that is the major portion of Section 6. Ultimately most source information must appear in a compiled instruction table (CIT).

Notice that several CIT's are stored on tape during the run of the executive routine. There is only one standard four-word format for CIT's:

	Decrement	Address
Word 1	(IFN) Internal formula No.	Instruction number within formula number
Word 2	Operation code of Instr. in SAP mnemonic	Decrement of Type A instruction
Word 3	Symbolic address (BCD) assigned by executive routine	
Word 4	*	Symbolic tag

* Decrement of Word 4:

Fortran I - The subscript, if any, of the symbolic address.

Fortran II and 709 - Relative absolute part of address of the instruction; for example, in

CLA N + 3

The + 3 would be entered into this field.

The decrement of word 1 contains the IFN of the statement from which the instruction was generated. The address of word 1 indicates the number of the instruction generated as a result of the statement. It would contain four if the table entry were the fourth instruction necessary to carry out the original Fortran statement.

The decrement of word 2 contains a BCD mnemonic representation of the instruction for which this entry is made (e.g., CLA, OCT). From this entry the reader can appreciate the sophistication of the Fortran translator; the executive program has written a symbolic instruction which will be subsequently assembled by an assembly program similar to USAP 1 and 2. The address of word 2 contains, if any, the decrement of a type A instruction.

Word 3 contains a BCD representation of the symbolic address assigned to the instruction by the executive routine. As in coding by hand, the executive routine uses symbolic addressing in writing its instructions in symbolic form. It is interesting to note that the symbols used by the machine have no mnemonic value to the human reader but of course a one-bit difference in configuration is accurate enough discrimination for the machine. A typical symbolic address is) 84!

The decrement of word 4 is explained in the footnote under the preceding diagram. The address of word 4 contains the symbolic tag of the FORTAG table for this instruction.

The COMPAIL file, the second file on tape 2, is a typical CIT table.

2.04.00 COMMON RECORD

Normally data and instructions are compiled adjacent to each other in order to preserve high-order storage cells.

The COMMON statement in FORTRAN II permits the programmer to assign specific core storage areas to the storage of data. The COMMON statement is of the following form: COMMON X, ANGLE, MATA, MATB

The items listed after COMMON statements will be assigned to core storage starting at location 774628. Entire arrays may be shifted to high-order storage through the use of the COMMON statement. (See pages 20-22 of Form C28-6000.)

The COMMON record is a compilation of all COMMON statements and is recorded on 2, file 5, record 3 during the run of Fortran II. It is generated during the run of Section 1'. The first word of this record is the label number, twelve. The second word is a count of the number of words in the COMMON record.

Each entry into the table demands as many words as there are items following the word "common". For example, the COMMON statement

COMMON X, ANGLE, MATA, MATB

requires the use of four full words and is recorded in the following format in BCD:

Word 1					X
Word 2	A	N	G	L	E
Word 3		M	A	T	A
Word 4		M	A	T	B

2.05.00 DIM RECORD

The DIM record is generated during the arithmetic processing in Section 1, as a result of encountering DIMENSION statements, and is recorded on drums. Recall that the DIMENSION statement consists of a list of variables with an integer in parentheses following the variables. Integer represents the greatest number of elements in an array. (See pages 28 and 47 of Form 32-03061.) During Section 1' the DIM table is converted to the SIZE table.

The DIMENSION statement is not executed (no instructions will appear in the object program for this statement) but will preserve blocks of storage for subscripted variables. The entry into the DIM table will occupy three words for one or two dimensions and four words for three dimensions. The entries are made according to the following format:

One-dimensional array: (Example: DIM A (7))

	Decrement	Address	
Word 1	A		Subscripted variable Dimension
Word 2		7	
Word 3	A	+7	Check sum of entry

Two-dimensional array (Example: DIM A (7, 12))

	Decrement	Address	
Word 1	A		Subscripted variable Dimensions 1 & 2
Word 2	7	12	
Word 3	A + 7	+12	Check sum of entry

Three-dimensional array: Example: DIM A (7, 12, 6)

	Decrement	Address	
Word 2	A		Subscripted variable
Word 2	7	12	Dimension 1 & 2
Word 3		6	Dimension 3
Word 4	A + 7	+ 12 + 6	Check sum of entry

2.06.00 DOTAG B FORMAT

This is the first record of file 6 on tape 2, with all Fortran modes. The DOTAG B table is the result of an analysis of priority of interlocking DO statements (nests). In this analysis an entry is made into the DOTAG table for every entry of the TDO table. The DOTAG B table is a collection of nine-word records. The first five records are identical to the corresponding entry in the TDO table. The last four records are a result of the analysis of the nests of DO statements. The last four records take on the following format: (for description of n_1 , n_2 , n_3 , see Section 2.22.00).

- Word 6: Decrement Level number (described in Section 3.00.00).
 Address $\left\{ \frac{n_2 - n_1 + n_3}{n_3} \right\} n_3$ (Only the integral part of the term in brackets is multiplied by n_3).
- Word 7: Decrement An integer representing level ≥ 0 .
 Address Level of definition of n_1 .
- Word 8: Decrement The bits in the field indicate to which level control will pass from this DO statement. (I.e., a bit in 7 indicates that control will pass to the seventh level from the current DO.) Control will always pass to a lower level from current level.
 Address Level of definition of n_2 .
- Word 9: 1 - 5 Test table number: A quantity to be entered into the indexing routine which optimized XR's.
 6 - 19 Symbolic tag to which reference is made with the test table number.
 20 - 35 Level of definition of n_2 .

For example, given the same DO statements as described in the TDO table format (5 DO 8 I = 1, 26, 2):

	Decrement	Address
Word 6	1 (first DO in nest)	26
Word 7	4	Level of definition of n_1
Word 8		Level of definition of n_2
Word 9	Test Table No.	Symbolic Tag Level of definition of n_3

$$n_1 = 1; n_2 = 26; n_3 = 2$$

$$\frac{n_2 - n_1 + n_3}{n_3} = \frac{26 - 1 + 2}{2} = \frac{27}{2} = 13 - 1/2$$

Use only integral part of previous calculation (13). Address of word 6 =
 $13 \times n_3 = 13 \times 2 = 26$

2.07.00 THE END RECORD

The END statement permits batch compiling. Several programs can be compiled with one pass of the Fortran translator provided an END statement separates the various symbolic programs. If only one program is being compiled, there is no need to include an END statement, although an END record is generated with five "2" entries as described below.

The END statement permits the programmer to use separate sense switch control for each of his programs. The specifications for use of the END statement are on page 22 of Form C28-6000. Briefly stated, the 0, 1, or 2 designation following the END indicate:

- 0 = Ignore the sense switch and assume it is up.
- 1 = Ignore the sense switch and assume it is down.
- 2 = Interrogate the sense switch.

For example: END (2, 2, 2, 0, 1). This statement indicates:
 Interrogate SS 1, 2, and 3.
 Ignore SS 4 and assume it is up.
 Ignore SS 5 and assume it is down.

The END statement takes precedence over the sense switch settings if the 0 and 1 codes are used.

In addition to sense switch control, the END statement simulates an end-of-file condition on card reader or tape and permits passage of I-O control from one program to the next.

The END record is always generated, whether an END card is used or not. During 1' the END record is recorded on tape 2, file 5, record 1. (Of course, there is no such entry in Fortran I, because it does not contain the END facility.) The END record on file 5, is the only record in this file not to have a label number.

The END entry is always a five-word record. The 0, 1, or 2 designation is stored in the addresses of these five words (in binary).

For example, END (2, 2, 2, 0, 1) appears in the END record as:

Word 1	000 010
Word 2	000 010
Word 3	000 010
Word 4	000 000
Word 5	000 001

2.08.00 EQUIT RECORD

The EQUIVALENCE statement permits the programmer to equate the names of several different quantities, or to assign the same storage location to several different variables. (See page 36, Form C28-6000.)

For example: EQUIVALENCE (A,B,C (5))

This statement says that location A, location B and the fourth location after C (the fifth location including C) are identical. In general A (P) is defined for $P \geq 1$ and means the (P-1)th location after A or the beginning of the A array, that is the Pth location in the array.

The EQUIT record is a table containing all the information included in EQUIVALENCE statements. Each item in the pairs of parentheses demands two full 704-9 words for storage. A minus sign in the last entry indicates the end of a series of equivalent storage locations.

For example, the following entry would be made for the foregoing EQUIVALENCE statement:

EQUIVALENCE (A, B (1), C (5))

The Equit Entry

Word 1	A	
Word 2		1
Word 3	B	
Word 4		1
Word 5	C	
Word 6	-	5

note minus sign

The "1" entry in word 2 is entered automatically and is associated with A. The "1" entry in word 4 is the 1 in parenthesis after the B. The "5" entry in the word 6 is the 5 following the C.

The EQUIT record is generated during the run of Section 1' and is recorded on tape 2, file 5, record 13 during Fortran II and 709 or record 9 during Fortran I. The first word in the first record is the table number, 8. The second word is a count of the number of entries into this table.

2.09.00 FIXCON RECORD

The FIXCON record is a table of fixed point constants specified by the program. These constants are entered in fixed point form as data or are subsequently computed from other fixed point constants. These numbers, entered without decimal points during READ statements and defined according to some FORMAT statement as fixed point constants, are one of the types entered into the FIXCON table. Numbers appearing as constants in statements of the form $A = 3 + B$ are entries in the FIXCON table; in this example "3" is an entry.

The FIXCON table is generated during Section 1 and is stored on drum in a two-word format for each entry. It seems peculiar but it is necessary to the program control, that each entry consists of (1) the fixed point constant in binary and (2) the

check sum of that word. An entry for the fixed point constant 5 would appear as:

Word 1	000 0101
Word 2	000 0101

Word 2 is the check sum of word 1.

In Section 3 the FIXCON table becomes the only record of tape 2, file 9.

2.10.00 FLOCON RECORD

The FLOCON record is a table of floating point constants occurring in the source program. They may be entered from an input source such as cards or tape, computed from combinations of floating point constants, or appear as coefficients with decimal points in Fortran source statements.

The FLOCON table is developed during Section I and is stored on the drum, in Fortran I, in the same format as the FIXCON table. That is, there are two words required for each entry, the first containing the floating point constant and the second the check sum of this one word. (In Fortran II and 709, this table is stored in high-order core storage.)

The FLOCON table is stored on tape 2, file 4, the first record, and also on the drum during the processing that occurs in Section 1.

2.11.00 FORMAT RECORD

The FORMAT record is a table of arguments presented in FORMAT statements. The arguments are stored in BCD form in sequential storage locations. Since the length of arguments is a variable, the number of words required to store all the argument must be variable. Each entry into the table is separated from succeeding entries by a word filled with bits. See Section 2.16.00. The format of the FORMAT entry is identical to that of the HOLARG entry.

The Format record is generated during the processing of Section 1'. It is stored on tape 2, file 4, the second record.

2.12.00 FORSUB RECORD

The FORSUB record is a table of the subroutines described in the source program. This table has only one word per entry. An entry is made for each Fortran II statement headed by the subroutine statement. The entry itself is a BCD image of the function described in the subroutine statement. For example (page 31, Fortran Reference Manual):

1. Subroutine MATMPY (A, N, M, B, L, C)

The BCD image of MATMPY would be entered into the FORSUB table.

This table is generated in Fortran II and 709 during the run of Section 1'. It is stored in the second record of file 3, on tape 2. Of course, Fortran I does not have subroutine calling facility.

2.13.00 FORTAG RECORD

The FORTAG record is a table that represents an index to the TAU table. It has a one-word entry of the following format:

IFN		* XR INFO	** INDEX TO TAU TABLE
1	17	24 26 27	35

* XR INFO - This field indicates whether or not the FORTAG entry uses an absolute or symbolic index register. If there are no entries, a symbolic XR is inferred. If there is an entry the field is treated like the tag field of an instruction (e.g., 24 = XRA, 25 = XRB, 26 = XRC).

** Index to TAU table - The bit configuration in this field indicates which TAU table entry has the associated IFN.

This table is generated during Section 1 and appears as a table in storage as a buffer. From the buffer area the table is written on tape 4 temporarily. Then during Section 1' the FORTAG table becomes the eleventh record of file 5 on tape 2 in Fortran II and 709. It is the seventh record on the same tape of Fortran I.

2.14.00 FORVAL AND FORVAR RECORDS

The FORVAL and FORVAR records are tables of the fixed point non-subscripted variable, appearing to the left of (FORVAL), and the right of (FORVAR), of the equality sign in a statement. A fixed point non-subscripted variable must satisfy the following conditions:

1. Must be six or less than six characters.
2. The first character must be alphabetic.
3. If an integer, it must start with I, J, K, L, M, or N.
4. Must not read like a function name.
5. Must not have a left parenthesis following it.
6. Must be entered as data in fixed point form.

For example, if A and B are fixed-point form, the statement, "ARG = BRAND + 6" contains "ARG" as an entry in the FORVAL table and "BRAND" as an entry in the FORVAR table.

The tables are generated during Section 1' and are written on tape 2, file 5, record 9 and 10 in Fortran II and 709 or record 5 and 6 in Fortran I. The label number of the FORVAL table is 6. The label number of the FORVAR table is 5. Both label numbers are the first words of the first records, in the respective tables. The second words in each record are the counts of the entries into these tables. Only one word is necessary to hold each entry in BCD form.

For example the statement ARG = BRAND +6 would be written:

FORVAL TABLE (BCD)

A	R	G	
---	---	---	--

FORVAR TABLE (BCD)

B	R	A	N	D
---	---	---	---	---

2.15.00 FRET TABLE

The FRET table is a table generated from the FREQUENCY statements given in the source program. (See page 37, Form 32-7026.) This is a variable length entry table; that is, each entry occupies an indeterminate number of words, dependent on the number of branch points described by frequency statements. Each FREQUENCY statement permits the programmer to specify the number of times a particular branching point will be utilized by the source program. For instance, a particular IF statement may appear in a program as:

38 IF N (10,20,30)

The programmer can best use index registers in the program by informing the program that branch 10 will be used five times, branch 20 will be used three times and branch 30 will be used six times, by entering the following frequency statement:

FREQUENCY 38 (5,3,6)

The general form is

FREQUENCY N (i, j, k...)

Where N = EFM of branch point

i, j, k = frequency of each branch

Entries into the FRET table are made according to the following format:

	Decrement	Address
Word 1		38
Word 2		5
Word 3		3
Word 4		6

The length of each entry will be determined by the number of branches.

The FRET table is generated during the run of Section 1. It appears as record 8 on file 5, of tape 2, during Fortran I. It appears as the record 14 of the same file during Fortran II and 709. Notice that the first word in each entry is flagged with a minus sign.

Before the table is recorded on tape 2 in Section 1' all the EFN's are changed to their corresponding IFN's. This means that 38 in the previous example would be replaced by its corresponding IFN.

2.16.00 HOLARG RECORD

The Holarg record is a collection of Hollerith arguments of CALL statements. In a CALL statement, the Hollerith argument is not describing argument of some subroutine but is itself the data to be operated upon. (See page 16, of Fortran II, Form C28-6000.) An example of this kind of CALL statement is:

```
265 CALL 56H * * * * * This data indicates the Orbital entry point
* * * * *
```

The number 265 is the external formula number. The 56 specifies the number of Hollerith characters (including blanks) that are in the Hollerith argument. H indicates that this is a Hollerith argument. The rest of the statement is data or commentary which will be later used in a print-out during the run of the object program under control of a PRINT 265 statement.

The HOLARG record is generated during the run of Section 1'. It is on tape 2, file 5, record 4 of Fortran II compilation. The first word in the record is the label number, 13. The second word is a count of the number of words in this record. After this a variable number of words are required to store each entry. Since the storage consists of BCD entries, the number of full words required will be equal to the number of BCD characters divided by six, allowing another full word for any fractional part. A word consisting of 36 binary bits is stored immediately after the last word. The preceding example will be entered into the HOLARG record in BCD as:

Word 1	*	*	*	*	*	*
Word 2	Blank*	T	H	I	S	Blank
Word 3	D	A	T	A	Blank	I
Word 4	N	D	I	C	A	T
Word 5	E	S	Blank	T	H	E
Word 6	Blank	O	R	B	I	T
Word 7	A	L	Blank	E	N	T
Word 8	R	Y	Blank	P	O	I
Word 9	N	T	Blank	*	*	*
Word 10	*	*	*			
Word 11	7	7	7	7	7	7

OCTAL

*Blank - - 110 000

2.17.00 LAMBDA, ALPHA, and BETA TABLES

The LAMBDA and ALPHA tables are generated during the arithmetic processing of Section 1. They represent the output of the executive program in its analysis of arithmetic statements. Pages 6 and 7 of Form 32-0306 describe the order of operation of arithmetic statements. In terms of the object program arithmetic statements must generate arithmetic and storing instructions with appropriate checking. These instructions must occur in proper order according to the placement of the operation symbols: + - * / (**). The LAMBDA table is an internal record of the order of operations necessary to accomplish the required object program. It contains a symbol-by-symbol analysis of the right-hand side of arithmetic statements. Each operation symbol receives a level number during this analysis which indicates the priority that the operation has in the processing.

The ALPHA table is used in Section 1 as a tally of the level of the operation symbol currently under investigation. Because the level number changes repeatedly through the analysis, the ALPHA table must be similarly altered. This alteration is much too detailed for this manual.

The LAMBDA table requires three words for each entry. Each entry in turn is the result of an operation symbol in a Fortran statement. Therefore, it is common for the LAMBDA table resulting from one Fortran arithmetic statement to extend over 100 core storage locations. The general configuration of the three-word entry is:

Word 1	Symbolic tag info.	Current level number
Word 2	Operation code: + * ** / -	
Word 3	Level number or BCD used in the source statement.	

The BETA table is closely associated with the LAMBDA and ALPHA in that it contains information on the control of arithmetic processing. This table requires only one word per entry. Its control is beyond the scope of this manual.

2.18.00 NONEXEC RECORD

The NONEXEC record is a table of IFN's and associated EFN's for non-executable Fortran statements. The following statements are non-executable:

PAUSE, FORMAT, DIMENSION, EQUIVALENCE, FREQUENCY

Each entry into the NONEXEC table requires only one word. The decrement of this entry contains the IFN, and the address contains the EFN of the non-executable instruction.

The NONEXEC table is generated during the run of Section 1 and is stored during Section 1' on tape 2, file 5, record 15 in Fortran II and 709 or as record 11 in Fortran I. The first word in the record is the label number; the second number is a count of the number of words in the table.

2.19.00 SIGMA AND TAU TABLES

The SIGMA and TAU tables are collections of the subscript information used by the source program. An entry into the SIGMA table requires two or three full words of storage, depending on the dimensions of the subscripts. An entry into the TAU table requires three, five, or seven full 704-9 words, again dependent upon the dimensions of the subscripts. Both tables are stored on drum during the run of Section 1. They are recorded according to the following format:

One-Dimension Subscripts. (Example: B (5 * I + 6)

The symbol in the parentheses is the subscript of B. It is the most complex type of subscript, chosen intentionally to indicate its entry into the SIGMA and TAU tables. It will be entered in a two word storage space of the SIGMA table according to the following format:

Sigma Table		
Decrement	Address	
Word 1	+6	Addend
Word 2	+6	Check sum

TAU table		
Decrement	Address	
Word 1	5	Multiplier
Word 2		Variable
Word 3	5 + 1	Check Sum

Two-Dimensional Subscripts. Example: B (5 * I + 6, 7 * J + 8)

The example shown will enter the SIGMA and TAU tables in the following format.

Sigma Table		
Decrement	Address	
Word 1	6	Addends 1 & 2
Word 2	6	Check sum

Tau Table		
Decrement	Address	
Word 1	5	Multipliers 1 & 2
Word 2		Variable 1
Word 3		Dimension of I
Word 4	d. See Section 2.05.00	in DIM statement
Word 5	5 + D	Check sum

Three-Dimensional Subscripts. Example: B (5 * I + 6, 7 * J + 8, 4 * I + 2)

Sigma Table		
Decrement	Address	
Word 1	6	Addends 1 & 2
Word 2	2	Addend 3
Word 3	6 + 2	Check sum

Tau Table		
Decrement	Address	
Word 1	5	Multipliers 1 & 2
Word 2	4	Multiplier 3
Word 3		Variable 1
Word 4		Variable 2
Word 5		Variable 3
Word 6	d ₁ =dimension of I	d ₂ =dimension of J
Word 7	5 4 d ₁	7 I J K d ₂

2.20.00 SIZE RECORD

The **SIZE** record is a table of the variables and maximum dimensions of arrays described by dimension statements. It is closely associated with the **TAU** table. (See Section 2.19.00)

The **SIZE** record requires two full words for each entry. (See page 35 of Form 32-7026 for a detailed description of the **DIMENSION** statement.) The entry is of the following format:

Word 1	BCD image of the variable
Word 2	Total size of array in binary

For example, given the **DIMENSION** statement:

DIMENSION C (3, 4, 5)

The table entry would appear as:

Word 1	C (BCD)
Word 2	60 (i.e. 3 x 4 x 5)

The **SIZE** table is generated during Section 1 and is stored in Section 1' on tape 2, file 4, record 3.

2.21.00 SUBDEF RECORD

Fortran II can also call in subroutines described by the programmer in the source program. For example, the subroutine introduced by the statement **SUBROUTINE MATMPY (A, N, M, B, L, C)** could be called into the main program by the statement:
CALL MATMPY (X, 5, 10, 4, 7, Z).

Essentially, what happens is that the previously described **MATMPY** subroutine is brought into the compilation with the arguments of the **SUBROUTINE** statement. Naturally the arguments of the **SUBROUTINE** statement should correspond in mode, number, and order to those of the original **MATMPY** subroutine. (See pages 16 and 17 of Form C28-6000.)

The **SUBDEF** record is generated during the run of Section 1'. It is recorded on tape 2, file 5, in record 2. The first word of this record is the label number, 11. The second word of this record is a count of the number of words in the record.

Each entry into the **SUBDEF** record requires one full word for the name of the subroutine (e.g., **MATMPY**) and one full word for each of the arguments included in the parenthesis. For example, the subroutine statement **SUBROUTINE MATMPY (A, N, M, B, L, C)** is recorded as:

	Decrement	Address	
Word 1		MATMPY	(BCD)
Word 2		A	
Word 3		N	
Word 4		M	
Word 5		B	
Word 6		L	
Word 7		C	

2.22.00 TDO RECORD

The TDO record is a table which results from DO statements in the symbolic program. Each entry requires five full words. The five words are written according to the following format.

Word 1:	Decrement	External formula number (EFN) of the DO statement (a)
	Address	The EFN of the last statement executed under control of the DO statement (b)
Word 2:	Decrement	The BCD symbol for the integer variable of the DO statement (I, J, K, L, M, ORN)
Word 3:	Address	First value of variable (n_1)
Word 4:	Address	Final value of variable (n_2)
Word 5:	Address	Increment of the variable (n_3)

NOTE: The symbols in parentheses are those used by the authors of Fortran. (See pages 24, 45 and 46 of Form 32-0306.)

The following DO statement would result in the table entry shown:

5 DO 8 I = 1, 25, 2

	Decrement	Address
Word 1	5	8
Word 2	I	
Word 3		1
Word 4		25
Word 5		2

The TDO record is written during the run of Section 1', on tape 2, file 5, on record 4 of Fortran I or record 8 of Fortran II and 709. The first word of the first record is the label number of the TDO record, 1. The second word of the first record is a count of the number of words in the TDO table.

Before tape 2 receives the TDO record in Section 1', all the EFN's are replaced by IFN's. (For example, the address of the first word becomes an IFN.)

2.23.00 TEIFNO RECORD

Two reference numbers are associated with Fortran statements, the internal IFN and external EFN formula numbers. All statements in the source program have internal formula numbers (IFN). These numbers are assigned to the statement sequentially starting with 1. The external formula number (EFN), on the other hand, is an arbitrary integer assigned to the statement by the programmer. It is entered into the location field of the source program card by the programmer, generally to permit reference to the particular statement by the source program. There is no need to assign an external formula number to any statement to which reference is never made. Therefore, all statements have IFN and some have both IFN and EFN.

The EFN's and their corresponding IFN's are stored in the TEIFNO record by the translator during the run of Section 1'. Each statement requires the use of one

full 704-9 word for storage. The entry is made as follows:

Decrement	Address
IFN	EFN

For example, if the following statement is the 28th statement in the program, the indicated table entry is made.

STATEMENT

15 DO 6 I = 1, 8

TEIFNO Entry

28	15
----	----

The TEIFNO (Table of External and Internal Formula Numbers) record is generated during the run of Section 1' as previously stated. This table is stored on tape 2, file 5, record 1 of Fortran I or the fifth record of the same tape and file on Fortran II or 709. The first word of the first record is the label number of the record, 0. The second record is a count of the number of words in the table.

2.24.00 TIFGO RECORD

The TIFGO record is a table of the IF, ASSIGN, and GO TO statements in the source program. Each statement in the program demands the use of two full 704 words for storage. This section describes entries that result from each type of statement. The first word of the first record in the TIFGO table is the label number. For this table the label number is two. The second word of the first record is a count of the number of words in the TIFGO table.

IF Statement Entry. Example: 16 IF (E) n_1, n_2, n_3

The entry for this statement would be as shown below, provided the IF statement given was the 31st statement in the source program. (See page 16 of Form 32-0306.)

	Decrement	Address
Word 1	(IFN) 31	n_1
Word 2	n_2	n_3

Unconditional GO TO Entry. Example: 21 GO TO n

The entry for this statement would be as shown below, provided the given statement was the third statement in the source program. (See page 14 of Form 32-0306.)

	Decrement	Address
Word 1	(IFN) 3	0
Word 2	0	n

Assigned Go To Entry

In this type of statement the GO TO destination is determined by a previous ASSIGN statement. (See pages 48 and 49 of Form 32-0306.) The list of alternatives following in parenthesis are merely a list of all the possible GO TO destinations.

For example, consider the following statement:

21 GO TO N (B₁, B₂, B₃, B₄ B_N)

The GO TO destination will be the Ith statement. The TIFGO table entry for this statement would be: (Assume an IFN of 6)

	Decrement	Address
Word 1	IFN (6)	2
Word 2	*CTRAD ₁	**CTRAD _N

*CTRAD₁ - The number of the entry in the TRAD record corresponding to the first possible transfer address given in the GO TO argument.

**CTRAD_N - The number of the last possible transfer address.

(See Section 2.25.00)

Computed GO TO Statement. Example: 26 GO TO (B₁, B₂, B₃ B_N) I.

In this type of statement a transfer will take place in the object program dependent on the current value of I. I is a variable which is assigned some computed integer by the source program. The transfer takes place to the Ith term of the GO TO list of B's. For example, if the value of I is computed as 3, then the program will transfer to the third location in the list of locations which follow GO TO (B₃ in the example). The entry for the computed GO TO takes the following form:

	Decrement	Address
Word 1	EFN 26	2*
Word 2	CTRAD ₁	CTRAN _N

*Unconditional.

Assign Statement Entry. Example: 29 ASSIGN 30 to N

This statement is used in conjunction with a GO TO statement, as described under the computed GO TO Statement.

The table entry of the above example takes the following form:

	Decrement	Address
Word 1	EFN 29	6*
Word 2	BLANK	Assigned value 30

*Unconditional.

Indicator-Controlled IF Statement. Example: 16 IF (sense light N) 30, 40

This statement is used in conjunction with:

1. Sense switches
2. Sense lights
3. Divide check indicator
4. Accumulator overflow light
5. Quotient overflow light

If the corresponding N light is on or switch is down, transfer of the program proceeds to the statement specified by the first number following the parenthesis. If the corresponding N light is off or switch is up, transfer of the program proceeds to the statement specified by the second number following the parenthesis.

The table entry takes the following format, for the example given:

	Decrement	Address
Word 1	EFN, 16	3, 4, or 5*
Word 2	30	40

*3 = Sense Switch or Sense Light
 4 = Divide Check
 5 = ACC or MQ overflow
 (Refer to pages 18-19 of Form 32-7026.)

The TIFGO record is generated in Section 1. The entries indicated in this section contain the external formula numbers (EFN) specified by the programmer. However, before the TIFGO table is written on tape 2 in Section 1', all EFN's are replaced by their corresponding IFN's.

2.25.00 TRAD RECORD

The TRAD record is a table of all possible transfer addresses listed in assigned and computed GO TO statements. See Section 2.24.00.

As many words are used for each entry (each assigned GO TO) as there are possible transfer addresses in the GO TO statement. The transfer address is entered in binary form into the address field of consecutive words on tape in the TRAD record.

Recall that the TIFGO record uses a standard two-word format, which did not have enough storage space for the transfer addresses of the assigned GO TO's. In the TIFGO record, the references made to the TRAD table were:

1. In the decrement field of the second word, the number of the word in the TRAD table which contains the first possible transfer address, reading the GO TO argument left to right.
2. In the address field of the second word, the number of the word in the TRAD table which contains the last possible transfer address.

For example, if the following two assigned GO TO statements are given first in the source program:

26 GO TO (3, 6, 9, 4, 28) I ; IFN = 61

28 GO TO (11, 13, 14, 15) I ; IFN = 62

The following table entries are made in binary, in Section 1':

	Decrement	Address	
Word 1		2	Label number Number of entries
Word 2		N	
Word 3	61 (IFN)	2	
Word 4	3 (1st TRA adr)	7 (last TRA adr)	
Word 5	62 (IFN)	2	
Word 6	8 (1st TRA adr)	1 (last TRA adr)	

TRAD Table		
Word 1	3	Label number
Word 2	X	Number of entries
Word 3	3	} First GO TO
Word 4	6	
Word 5	9	
Word 6	4	
Word 7	28	
Word 8	11	} Second GO TO
Word 9	13	
Word 10	14	
Word 11	15	

The TRAD record is generated during the processing in Section 1. It is entered on tape 2, file 5, as record 7 in Fortran II and 709 and record 3 in Fortran I. The first word of the first record is the label number three. The second word of the first record is a binary count of the number of words in the TRAD record.

The previous examples of table entries indicate the format in Section 1. In 1', however, where the tables are recorded on tape 2, all the EFN's are replaced with their corresponding IFN's.

2.26.00 TSTOP RECORD

The TSTOP record is a table of IFN's associated with STOP statements in the source program. Each entry into the table requires only one word. The decrement of the word contains the IFN and the address contains the EFN of the STOP statements.

The TSTOP table is generated during Section 1' on tape 2, file 5, record 16 in Fortran II and 709 or as record 12 in Fortran I. The first word in the record is the label number, the second word is a count of the number of words in the table.

3.00.00 FORTRAN AUTOMATIC CODING SYSTEM

This section reproduces, with permission of the Institute of Radio Engineers, the paper which appeared in the Proceedings of the Western Joint Computer Conference, February 26-28, 1957.

The FORTRAN Automatic Coding System

J. W. BACKUS†, R. J. BEEBER†, S. BEST‡, R. GOLDBERG†, L. M. HAIBT†,
H. L. HERRICK†, R. A. NELSON†, D. SAYRE†, P. B. SHERIDAN†,
H. STERN†, I. ZILLER†, R. A. HUGHES§, AND R. NUTT||

INTRODUCTION

THE FORTRAN project was begun in the summer of 1954. Its purpose was to reduce by a large factor the task of preparing scientific problems for IBM's next large computer, the 704. If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. For it was known that about two-thirds of the cost of solving most scientific and engineering problems on large computers was that of problem preparation. Furthermore, more than 90 per cent of the elapsed time for a problem was usually devoted to planning, writing, and debugging the program. In many cases the development of a general plan for solving a problem was a small job in comparison to the task of devising and coding machine procedures to carry out the plan. The goal of the FORTRAN project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and obtain automatically from this specification an efficient 704 program to carry out the procedure. It was expected that such a system would reduce the coding and debugging task to less than one-fifth of the job it had been.

Two and one-half years and 18 man years have elapsed since the beginning of the project. The FORTRAN

system is now complete. It has two components: the FORTRAN language, in which programs are written, and the translator or executive routine for the 704 which effects the translation of FORTRAN language programs into 704 programs. Descriptions of the FORTRAN language and the translator form the principal sections of this paper.

The experience of the FORTRAN group in using the system has confirmed the original expectations concerning reduction of the task of problem preparation and the efficiency of output programs. A brief case history of one job done with a system seldom gives a good measure of its usefulness, particularly when the selection is made by the authors of the system. Nevertheless, here are the facts about a rather simple but sizable job. The programmer attended a one-day course on FORTRAN and spent some more time referring to the manual. He then programmed the job in four hours, using 47 FORTRAN statements. These were compiled by the 704 in six minutes, producing about 1000 instructions. He ran the program and found the output incorrect. He studied the output (no tracing or memory dumps were used) and was able to localize his error in a FORTRAN statement he had written. He rewrote the offending statement, recompiled, and found that the resulting program was correct. He estimated that it might have taken three days to code this job by hand, plus an unknown time to debug it, and that no appreciable increase in speed of execution would have been achieved thereby.

† Internat'l Business Machines Corp., New York, N. Y.
‡ Mass. Inst. Tech., Computation Lab., Cambridge, Mass.
§ Radiation Lab., Univ. of California, Livermore, Calif.
|| United Aircraft Corp., East Hartford, Conn.

THE FORTRAN LANGUAGE

The FORTRAN language is most easily described by reviewing some examples.

Arithmetic Statements

Example 1: Compute:

$$\text{root} = \frac{-(B/2) + \sqrt{(B/2)^2 - AC}}{A}$$

FORTRAN Program:

```
ROOT
  = (-(B/2.0) + SQRTF((B/2.0) ** 2 - A * C))/A.
```

Notice that the desired program is a single FORTRAN statement, an arithmetic formula. Its meaning is: "Evaluate the expression on the right of the = sign and make this the value of the variable on the left." The symbol * denotes multiplication and ** denotes exponentiation (i.e., A ** B means A^B). The program which is generated from this statement effects the computation in floating point arithmetic, avoids computing (B/2.0) twice and computes (B/2.0) ** 2 by a multiplication rather than by an exponentiation routine. [Had (B/2.0) ** 2.01 appeared instead, an exponentiation routine would necessarily be used, requiring more time than the multiplication.]

The programmer can refer to quantities in both floating point and integer form. Integer quantities are somewhat restricted in their use and serve primarily as subscripts or exponents. Integer constants are written without a decimal point. Example: 2 (integer form) vs 2.0 (floating point form). Integer variables begin with I, J, K, L, M, or N. Any meaningful arithmetic expression may appear on the right-hand side of an arithmetic statement, provided the following restriction is observed: an integer quantity can appear in a floating-point expression only as a subscript or as an exponent or as the argument of certain functions. The functions which the programmer may refer to are limited only by those available on the library tape at the time, such as SQRTF, plus those simple functions which he has defined for the given problem by means of function statements. An example will serve to describe the latter.

Function Statements

Example 2: Define a function of three variables to be used throughout a given problem, as follows:

```
ROOTF(A, B, C)
  = (-(B/2.0) + SQRTF((B/2.0) ** 2 - A * C))/A.
```

Function statements must precede the rest of the program. They are composed of the desired function name (ending in F) followed by any desired arguments which appear in the arithmetic expression on the right of the = sign. The definition of a function may employ any

previously defined functions. Having defined ROOTF as above, the programmer may apply it to any set of arguments in any subsequent arithmetic statements. For example, a later arithmetic statement might be

```
THETA = 1.0 + GAMMA * ROOTF(PI, 3.2 * Y
                             + 14.0, 7.63).
```

DO Statements, DIMENSION Statements, and Subscripted Variables

Example 3: Set Q_{max} equal to the largest quantity P(a_i+b_i)/P(a_i-b_i) for some i between 1 and 1000 where P(x) = c₀ + c₁x + c₂x² + c₃x³.

FORTRAN Program:

- 1) POLYF(X) = C0 + X * (C1 + X * (C2 + X * C3)).
- 2) DIMENSION A(1000), B(1000).
- 3) QMAX = -1.0 E20.
- 4) DO 5 I = 1, 1000.
- 5) QMAX = MAXF(QMAX, POLYF(A(I) + B(I))/POLYF(A(I) - B(I))).
- 6) STOP.

The program above is complete except for input and output statements which will be described later. The first statement is not executed; it defines the desired polynomial (in factored form for efficient output program). Similarly, the second statement merely informs the executive routine that the vectors A and B each have 1000 elements. Statement 3 assigns a large negative initial value to QMAX, -1.0 × 10²⁰, using a special concise form for writing floating-point constants. Statement 4 says "DO the following sequence of statements down to and including the statement numbered 5 for successive values of I from 1 to 1000." In this case there is only one statement 5 to be repeated. It is executed 1000 times; the first time reference is made to A(1) and B(1), the second time to A(2) and B(2), etc. After the 1000th execution of statement 5, statement 6—STOP—is finally encountered. In statement 5, the function MAXF appears. MAXF may have two or more arguments and its value, by definition, is the value of its largest argument. Thus on each repetition of statement 5 the old value of QMAX is replaced by itself or by the value of POLYF(A(I) + B(I))/POLYF(A(I) - B(I)), whichever is larger. The value of QMAX after the 1000th repetition is therefore the desired maximum.

Example 4: Multiply the n × n matrix a_{ij} (n ≤ 20) by its transpose, obtaining the product elements on or below the main diagonal by the relation

$$c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot a_{j,k} \quad (\text{for } j \leq i)$$

and the remaining elements by the relation

$$c_{j,i} = c_{i,j}$$

FORTRAN Program:

```

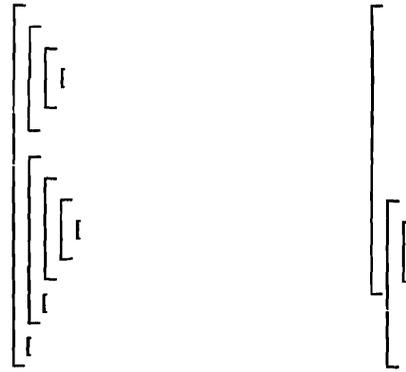
    DIMENSION A(20, 20), C(20, 20)
    DO 2 I = 1, N
        DO 2 J = 1, I
            C(I, J) = 0.0
            DO 1 K = 1, N
                C(I, J) = C(I, J) + A(I, K) * A(J, K)
            C(J, I) = C(I, J)
        STOP
    
```

As in the preceding example, the DIMENSION statement says that there are two matrices of maximum size 20×20 named A and C. For explanatory purposes only, the three boxes around the program show the sequence of statements controlled by each DO statement. The first DO statement says that procedure P, i.e., the following statements down to statement 2 (outer box) is to be carried out for $I=1$ then for $I=2$ and so on up to $I=N$. The first statement of procedure P (DO 2 J=1, I) directs that procedure Q be done for $J=1$ to $J=I$. And of course each execution of procedure Q involves N executions of procedure R for $K=1, 2, \dots, N$.

Consider procedure Q. Each time its last statement is completed the "index" J of its controlling DO statement is increased by 1 and control goes to the first statement of Q, until finally its last statement is reached and $J=I$. Since this is also the last statement of P and P has not been repeated until $I=N$, I will be increased and control will then pass to the first statement of P. This statement (DO 2 J=1, I) causes the repetition of Q to begin again. Finally, the last statement of Q and P (statement 2) will be reached with $J=I$ and $I=N$, meaning that both Q and P have been repeated the required number of times. Control will then go to the next statement, STOP. Each time R is executed a new term is added to a product element. Each time Q is executed a new product element and its mate are obtained. Each time P is executed a product row (over to the diagonal) and the corresponding column (down to the diagonal) are obtained.

The last example contains a "nest" of DO statements, meaning that the sequence of statements controlled by one DO statement contains other DO statements. Another example of such a nest is shown in the next column, on the left. Nests of the type shown on the right are not permitted, since they would usually be meaningless.

Although not illustrated in the examples given, the programmer may also employ subscripted variables having three independent subscripts.



READ, PRINT, FORMAT, IF and GO TO Statements

Example 5: For each case, read from cards two vectors, ALPHA and RHO, and the number ARG. ALPHA and RHO each have 25 elements and $ALPHA(I) \leq ALPHA(I+1)$, $I=1$ to 24. Find the SUM of all the elements of ALPHA from the beginning to the last one which is less than or equal to ARG [assume $ALPHA(1) \leq ARG < ALPHA(25)$]. If this last element is the Nth, set $VALUE = 3.14159 * RHO(N)$. Print a line for each case with ARG, SUM, and VALUE.

FORTRAN Program:

```

    DIMENSION ALPHA(25), RHO(25)
    1) FORMAT(5F12.4)
    2) READ 1, ALPHA, RHO, ARG
        SUM=0.0
        DO 3 I=1, 25
            IF (ARG-ALPHA(I)) 4, 3, 3.
    3) SUM=SUM+ALPHA(I)
    4) VALUE=3.14159 * RHO(I-1)
        PRINT 1, ARG, SUM, VALUE
        GO TO 2.
    
```

The FORMAT statement says that numbers are to be found (or printed) 5 per card (or line), that each number is in fixed point form, that each number occupies a field 12 columns wide and that the decimal point is located 4 digits from the right. The FORMAT statement is not executed; it is referred to by the READ and PRINT statements to describe the desired arrangement of data in the external medium.

The READ statement says "READ cards in the card reader which are arranged according to FORMAT statement 1 and assign the successive numbers obtained as values of ALPHA(I) $I=1, 25$ and RHO(I) $I=1, 25$ and ARG." Thus "ALPHA, RHO, ARG" is a description of a list of 51 quantities (the size of ALPHA and RHO being obtained from the DIMENSION statement). Reading of cards proceeds until these 51 quantities have been obtained, each card having five numbers, as per the FORMAT description, except the last which has the value of ARG only. Since ARG terminated the list, the remaining four fields on the last card are not read. The PRINT statement is similar to READ except that it specifies a list of only three quantities. Thus

each execution of PRINT causes a single line to be printed with ARG, SUM, VALUE printed in the first three of the five fields described by FORMAT statement 1.

The IF statement says "If ARG-ALPHA(I) is negative go to statement 4, if it is zero go to statement 3, and if it is positive go to 3." Thus the repetition of the two statements controlled by the DO consists normally of computing ARG-ALPHA(I), finding it zero or positive, and going to statement 3 followed by the next repetition. However, when I has been increased to the extent that the first ALPHA exceeding ARG is encountered, control will pass to statement 4. Note that this statement does not belong to the sequence controlled by the DO. In such cases, the repetition specified by the DO is terminated and the value of the index (in this case I) is preserved. Thus if the first ALPHA exceeding ARG were ALPHA (20), then RHO (19) would be obtained in statement 4.

The GO TO statement, of course, passes control to statement 2, which initiates reading the 11 cards for the next case. The process will continue until there are no more cards in the reader. The above program is entirely complete. When punched in cards as shown, and compiled, the translator will produce a ready-to-run 704 program which will perform the job specified.

Other Types of FORTRAN Statements

In the above examples the following types of FORTRAN statements have been exhibited.

- Arithmetic statements
- Function statements
- DO statements
- IF statements
- GO TO statements
- READ statements
- PRINT statements
- STOP statements
- DIMENSION statements
- FORMAT statements.

The explanations accompanying each example have attempted to show some of the possible applications and variations of these statements. It is felt that these examples give a representative picture of the FORTRAN language; however, many of its features have had to be omitted. There are 23 other types of statements in the language, many of them completely analogous to some of those described here. They provide facilities for referring to other input-output and auxiliary storage devices (tapes, drums, and card punch), for specifying preset and computed branching of control, for detecting various conditions which may arise such as an attempt to divide by zero, and for providing various information about a program to the translator. A complete description of the language is to be found in *Programmer's Reference Manual, the FORTRAN Automatic Coding System for the IBM 704*.

Preparation of a Program for Translation

The translator accepts statements punched one per card (continuation cards may be used for very long statements). There is a separate key on the keypunching device for each character used in FORTRAN statements and each character is represented in the card by several holes in a single column of the card. Five columns are reserved for a statement number (if present) and 66 are available for the statement. Keypunching a FORTRAN program is therefore a process similar to that of typing the program.

Translation

The deck of cards obtained by keypunching may then be put in the card reader of a 704 equipped with the translator program. When the load button is pressed one gets either 1) a list of input statements which fail to conform to specifications of the FORTRAN language accompanied by remarks which indicate the type of error in each case; 2) a deck of binary cards representing the desired 704 program, 3) a binary tape of the program which can either be preserved or loaded and executed immediately after translation is complete, or 4) a tape containing the output program in symbolic form suitable for alteration and later assembly. (Some of these outputs may be unavailable at the time of publication.)

THE FORTRAN TRANSLATOR

General Organization of the System

The FORTRAN translator consists of six successive sections, as follows.

Section 1: Reads in and classifies statements. For arithmetic formulas, compiles the object (output) instructions. For nonarithmetic statements including input-output, does a partial compilation, and records the remaining information in tables. All instructions compiled in this section are in the COMPAIL file.

Section 2: Compiles the instructions associated with indexing, which result from DO statements and the occurrence of subscripted variables. These instructions are placed in the COMPDO file.

Section 3: Merges the COMPAIL and COMPDO files into a single file, meanwhile completing the compilation of nonarithmetic statements begun in Section 1. The object program is now complete, but assumes an object machine with a large number of index registers.

Section 4: Carries out an analysis of the flow of the object program, to be used by Section 5.

Section 5: Converts the object program to one which involves only the three index registers of the 704.

Section 6: Assembles the object program, producing a relocatable binary program ready for running. Also on demand produces the object program in SHARE symbolic language.

(Note: Section 3 is of internal importance only; Section 6 is a fairly conventional assembly program. These sections will be treated only briefly in what follows.)

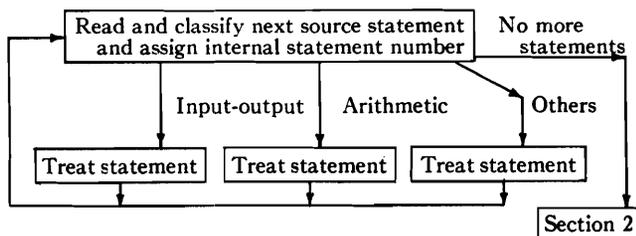
Within the translator, information is passed from section to section in two principal forms: as compiled instructions, and as tables. The compiled instructions (e.g., the COMPAIL and COMPDO files, and later their merged result) exist in a four-word format which contains all the elements of a symbolic 704 instruction; i.e., symbolic location, three-letter operation code, symbolic address with relative absolute part, symbolic tag, and absolute decrement. (Instructions which refer to quantities given symbolic names by the programmer have those same names in their addresses.) This symbolic format is retained until section 6. Throughout, the order of the compiled instructions is maintained by means of the symbolic locations (internal statement numbers), which are assigned in sequential fashion by section 1 as each new statement is encountered.

The tables contain all information which cannot yet be embodied in compiled instructions. For this reason the translator requires only the single scan of the source program performed in section 1.

A final observation should be made about the organization of the system. Basically, it is simple, and most of the complexities which it does possess arise from the effort to cause it to produce object programs which can compete in efficiency with hand-written programs. Some of these complexities will be found within the individual sections; but also, in the system as a whole, the sometimes complicated interplay between compiled instructions and tables is a consequence of the desire to postpone compiling until the analysis necessary to produce high object-program efficiency has been performed.

Section 1 (Beeber, Herrick, Nutt, Sheridan, and Stern)

The over-all flow of section 1 is



For an input-output statement, section 1 compiles the appropriate read or write select (RDS or WRS) instruction, and the necessary copy (CPY) instructions (for binary operations) or transfer instructions to pre-written input-output routines which perform conversion between decimal and binary and govern format (for decimal operations). When the list of the input-output statement is repetitive, table entries are made which will cause section 2 to generate the indexing instructions necessary to make the appropriate loops.

The treatment of statements which are neither input-output nor arithmetic is similar; i.e., those instructions

which can be compiled are compiled, and the remaining information is extracted and placed in one or more of the appropriate tables.

In contrast, arithmetic formulas are completely treated in section 1, except for open (built-in) subroutines, which are added in section 3; a complete set of compiled instructions is produced in the COMPAIL file. This compilation involves two principal tasks: 1) the generation of an appropriate sequence of arithmetic instructions to carry out the computation specified by the formula, and 2) the generation of (symbolic) tags for those arithmetic instructions which refer to subscripted variables (variables which denote arrays) which in combination with the indexing instructions to be compiled in section 2 will refer correctly to the individual members of those arrays. Both these tasks are accomplished in the course of a single scan of the formula.

Task 2) can be quickly disposed of. When a subscripted variable is encountered in the scan, its subscript(s) are examined to determine the symbols used in the subscripts, their multiplicative coefficients, and the dimensions of the array. These items of information are placed in tables where they will be available to section 2; also from them is generated a subscript combination name which is used as the symbolic tag of those instructions which refer to the subscripted variable.

The difficulty in carrying out task 1) is one of *level*; there is implicit in every arithmetic formula an order of computation, which arises from the control over ordering assigned by convention to the various symbols (parentheses, +, -, *, /, etc.) which can appear, and this implicit ordering must be made explicit before compilation of the instructions can be done. This explicitness is achieved, during the formula scan, by associating with each operation required by the formula a *level number*, such that if the operations are carried out in the order of increasing level number the correct sequence of arithmetic instructions will be obtained. The sequence of level numbers is obtained by means of a set of rules, which specify for each possible pair formed of operation type and symbol type the increment to be added to or subtracted from the level number of the preceding pair.

In fact, the compilation is not carried out with the raw set of level numbers produced during the scan. After the scan, but before the compilation, the levels are examined for empty sections which can be deleted, for permutations of operations on the same level which will reduce the number of accesses to memory, and for redundant computation (arising from the existence of common subexpressions) which can be eliminated.

An example will serve to show (somewhat inaccurately) some of the principles employed in the level-analysis process. Consider the following arithmetic expression:

$$A + B**C*(E + F).$$

In the level analysis of this expression parentheses are in effect inserted which define the proper order in which the operations are to be performed. If only three implied levels are recognized (corresponding to +, * and ***) the expression obtains the following:

$$+((*A)) + ((*B**C)*[+((*E)) + ((*F))]).$$

The brackets represent the parentheses appearing in the original expression. (The level-analysis routine actually recognizes an additional level corresponding to functions.) Given the above expression the level-analysis routine proceeds to define a sequence of new dependent variables the first of which represents the value of the entire expression. Each new variable is generated whenever a left parenthesis is encountered and its definition is entered on another line. In the single scan of the expression it is often necessary to begin the definition of one new variable before the definition of another has been completed. The subscripts of the u 's in the following sets of definitions indicate the order in which they were defined.

$$\begin{aligned} u_0 &= + u_1 + u_3 \\ u_1 &= * u_2 \\ u_2 &= ** A \\ u_3 &= * u_4 * u_5 \\ u_4 &= ** B ** C \\ u_5 &= + u_6 + u_8 \\ u_6 &= * u_7 \\ u_7 &= ** E \\ u_8 &= * u_9 \\ u_9 &= ** F. \end{aligned}$$

This is the point reached at the end of the formula scan. What follows illustrates the further processing applied to the set of levels. Notice that u_9 , for example, is defined as $**F$. Since there are not two or more operands to be combined the $**$ serves only as a level indication and no further purpose is served by having defined u_9 . The procedure therefore substitutes F for u_9 wherever u_9 appears and the line $u_9 = **F$ is deleted. Similarly, F is then substituted for u_8 and $u_8 = *F$ is deleted. This elimination of "redundant" u 's is carried to completion and results in the following:

$$\begin{aligned} u_0 &= + A + u_3 \\ u_3 &= * u_4 * u_5 \\ u_4 &= ** B ** C \\ u_5 &= + E + F. \end{aligned}$$

These definitions, read up, describe a legitimate procedure for obtaining the value of the original ex-

pression. The number of u 's remaining at this point (in this case four) determines the number of intermediate quantities which *may* need to be stored. However, further examination of this case reveals that the result of u_3 is in the accumulator, ready for u_0 ; therefore the store and load instructions which would usually be compiled between u_3 and u_0 are omitted.

Section 2 (Nelson and Ziller)

Throughout the object program will appear instructions which refer to subscripted variables. Each of these instructions will (until section 5) be tagged with a symbolic index register corresponding to the particular subscript combination of the subscripts of the variable [e.g., (I, K, J) and (K, I, J) are two different subscript combinations]. If the object program is to work correctly, every symbolic index register must be so governed that it will have the appropriate contents at every instant that it is being used. It is the source program, of course, which determines what these appropriate contents must be, primarily through its DO statements, but also through arithmetic formulas (e.g. $I = N + 1$) which may define the values of variables appearing in subscripts, or input formulas which may read such values in at object time. Moreover, in the case of DO statements, which are designed to produce loops in the object program, it is necessary to provide tests for loop exit. It is these two tasks, the governing of symbolic index registers and the testing of their contents, which section 2 must carry out.

Much of the complexity of what follows arises from the wish to carry out these tasks optimally; i.e., when a variable upon which many subscript combinations depend undergoes a change, to alter only those index registers which really require changing in the light of the problem flow, and to handle exits correctly with a minimum number of tests.

If the following subscripted variable appears in a FORTRAN program

$$A(2*I + 1, 4*J + 3, 6*K + 5),$$

the index quantity which must be in its symbolic index register when this reference to A is made is

$$(c_1 i - 1) + (c_2 j - 1)d_i + (c_3 k - 1)d_i d_j + 1,$$

where c_1 , c_2 , and c_3 in this case have the values 2, 4, and 6; i , j , and k are the values of I , J , and K at the moment, and d_i and d_j are the I and J dimensions of A . The effect of the addends 1, 3, and 5 is incorporated in the address of the instruction which makes the reference.

In general, the index quantity associated with a subscript combination as given above, once formed, is not recomputed. Rather, every time one of the variables in a subscript combination is incremented under control of a DO, the corresponding quantity is incremented by the appropriate amount. In the example given, if K

is increased by n (under control of a DO), the index quantity is increased by $c_3 d_3 n$, giving the correct new value. The following paragraphs discuss in further detail the ways in which index quantities are computed and modified.

Choosing the Indexing Instructions; Case of Subscripts Controlled by DO's

We distinguish between two classes of subscript; those which are in the range of a DO having that subscript as its index symbol, and those subscripts which are not controlled by DO's.

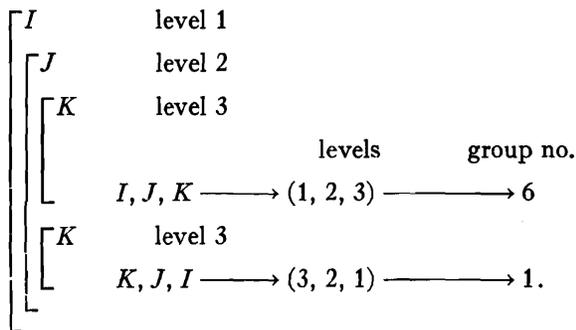
The fundamental idea for subscripts controlled by DO's is that a sequence of indexing instruction groups can be selected to answer the requirements, and that the choice of a particular instruction group depends mainly on the arrangement of the subscripts within the subscript combination and the order of the DO's controlling each subscript.

DO's often exist in *nests*. A nest of DO's consists of all the DO's contained by some one DO which is itself not contained by any other. Within a nest, DO's are assigned level numbers. Wherever the index symbol of a DO appears as a subscript within the range of that DO, the level number of the DO is assigned to the subscript. The relative values of the level numbers in a subscript combination produce a group number which, along with other information, determines which indexing instruction group is to be compiled.

The source language,

```
DO 10 I = 1, 5
DO 10 J = 1, 5
DO 5 K = 1, J
5 ... A(I, J, K) ... (some statement referring to
    A(I, J, K))
DO 10 K = J, 5
10 ... A(K, J, I) ...
```

produces the following DO structure and group combinations:



Producing the Decrement Parts of Indexing Instructions

The part of the 704 instruction used to change or test the contents of an index register is called the decrement part of the instruction.

The decrement parts of the FORTRAN indexing instructions are functions of the dimensions of arrays and of the parameters of DO's; that is, of the initial value n_1 , the upper bound n_2 , and the increment n_3 appearing in the statement DO 1 $i=n_1, n_2, n_3$. The general form of the function is $[(n_2-n_1+n_3)/n_3]n_3g$ where g represents necessary coefficients and dimensions, and $[x]$ denotes the integral part of x .

If all the parameters are constants, the decrement parts are computed during the execution of the FORTRAN executive program. If the parameters are variable symbols, then instructions are compiled in the object program to compute the proper decrement values. For object program efficiency, it is desirable to associate these computing instructions with the outermost DO of a nest, where possible, and not with the inner loops, even though these inner DO's may have variable parameters. Such a variable parameter (e.g., N in "DO 7 $I=1, N$ ") may be assigned values by the programmer by any of a number of methods; it may be a value brought in by a READ statement, it may be calculated by an arithmetic statement, it may take its value from a transfer exit from some other DO whose index symbol is the pertinent variable symbol, or it may be under the control of a DO in the nest. A search is made to determine the smallest level number in the nest within which the variable parameter is not assigned a new value. This level number determines the place at which computing instructions can best be compiled.

Case of Subscripts not Controlled by DO's

The second of the two classes of subscript symbols is that of subscript symbols which are not under control of DO's. Such a subscript can be given a value in a number of ways similar to the defining of DO parameters: a value may be read in by a READ statement, it may be calculated by an arithmetic statement, or it may be defined by an exit made from a DO with that index symbol.

For subscript combinations with no subscript under the control of a DO, the basic technique used to introduce the proper values into a symbolic index register is that of determining where such definitions occur, and, at the point of definition, using a subroutine to compute the new index quantity. These subroutines are generated at executive time, if it is determined that they are necessary.

If the index quantity exists in a DO nest at the time of a transfer exit, then no subroutine calculations are necessary since the exit values are precisely the desired values.

Mixed Cases

In cases in which some subscripts in a subscript combination are controlled by DO's, and some are not, instructions are compiled to compute the initial value

of the subscript combination at the beginning of the outside loop. If the non-DO-controlled subscript symbol is then defined inside the loop (that is, after the computing of the load quantity) the procedure of using a subroutine at the point of subscript definition will bring the new value into the index register.

An exception to the use of a subroutine is made when the subscript is defined by a transfer exit from a DO, and that DO is within the range of a DO controlling some other subscript in the subscript combination. In such instances, if the index quantity is used in the inner DO, no calculation is necessary; the exit values are used. If the index quantity is not used, instructions are compiled to simulate this use, so that in either case the transfer exit leaves the correct function value in the index register.

Modification and Optimization

Initializing and computing instructions corresponding to a given DO are placed in the object program at a point corresponding to the lowest possible (outermost) DO level rather than at the point corresponding to the given DO. This technique results in the desired removal of certain instructions from the most frequent innermost loops of the object program. However, it necessitates the consideration of some complex questions when the flow within a nest of DO's is complicated by the occurrence of transfer escapes from DO-type repetition and by other IF and GO TO flow paths. Consider a simple example, a nest having a DO on I containing a DO on J , where the subscript combination (I, J) appears only in the inner loop. If the object program corresponded precisely to the FORTRAN language program, there would be instructions at the entrance point of the inner loop to set the value of J in (I, J) to the initial value specified by the inner DO. Usually, however, it is more efficient to reset the value of J in (I, J) at the end of the inner loop upon leaving it, and the object program is so constructed. In this case it becomes necessary to compile instructions which follow every *transfer* exit from the inner loop into the outer loop (if there are any such exits) which will also reset the value of J in (I, J) to the initial value it should have at the entrance of the inner loop. These instructions, plus the initialization of both I and J in (I, J) at the entrance of the outer loop (on I), insure that J always has its proper initial value at the entrance of the inner loop even though no instructions appear at that point which change J . The situation becomes considerably more complicated if the subscript combination (I, J) also appears in the outer loop. In this case two independent index quantities are created, one corresponding to (I, J) in the inner loop, the other to (I, J) in the outer loop.

Optimizing features play an important role in the modification of the procedures and techniques outlined above. It may be the case that the DO structure and

subscript combinations of a nest describe the scanning of a two- or three-dimensional array which is the equivalent of a sequential scan of a vector; *i.e.*, a reference to each of a set of memory locations in descending order. Such an equivalent procedure is discovered, and where the flow of a nest permits, is used in place of more complicated indexing. This substitution is not of an empirical nature, but is instead the logical result of a generalized analysis.

Other optimizing techniques concern, for example, the computing instructions compiled to evaluate the functions (governing index values and decrements) mentioned previously. When some of the parameters are constant, the functions are reduced at executive time, and a frequent result is the compilation of only one instruction, a reference to a variable, to obtain a proper initializing value.

In choosing the symbolic index register in which to test the value of a subscript for exit purposes, those index registers are avoided which would require the compilation of instructions to modify the test instruction decrement.

Section 4 (Haibt) and Section 5 (Best)

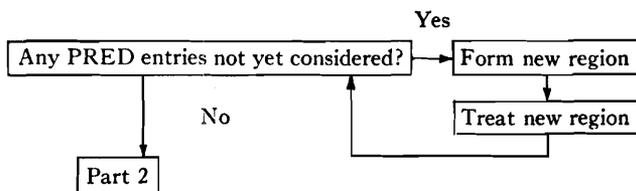
The result of section 3 is a complete program, but one in which tagged instructions are tagged only symbolically, and which assumes that there will be a real index register available for every symbolic one. It is the task of sections 4 and 5 to convert this program to one involving only the three real index registers of the 704. Generally, this requires the setting up, for each symbolic index register, of a storage cell which will act as an *index cell*, and the addition of instructions to load the real index registers from, and store them into, the index cells. This is done in section 5 (tag analysis) on the basis of information about the pattern and frequency of flow provided by section 4 (flow analysis) in such a way that the time spent in loading and storing index registers will be nearly minimum.

The fundamental unit of program is the *basic block*; a basic block is a stretch of program which has a single entry point and a single exit point. The purpose of section 4 is to prepare for section 5 a table of predecessors (PRED table) which enumerates the basic blocks and lists for every basic block each of the basic blocks which can be its immediate predecessor in flow, together with the absolute frequency of each such basic block link. This table is obtained by an actual "execution" of the program in Monte-Carlo fashion, in which the outcome of conditional transfers arising out of IF-type statements and computed GO TO's is determined by a random number generator suitably weighted according to whatever FREQUENCY statements have been provided.

Section 5 is divided into four parts, of which part 1 is the most important. It makes all the major decisions concerning the handling of index registers, but records

them simply as bits in the PRED table and a table of all tagged instructions, the STAG table. Part 2 merely reorganizes those tables; part 3 adds a slight further treatment to basic blocks which are terminated by an assigned GO TO; and finally part 4 compiles the finished program under the direction of the bits in the PRED and STAG tables. Since part 1 does the real work involved in handling the index registers, attention will be confined to this part in the sequel.

The basic flow of part 1 of section 5 is,



Consider a moment partway through the execution of part 1, when a new region has just been treated. The less frequent basic blocks have not yet been encountered; each basic block that has been treated is a member of some region. The existing regions are of two types: transparent, in which there is at least one real index register which has not been used in any of the member basic blocks, and opaque. Bits have been entered in the STAG table, calling where necessary for an LXD (load index register from index cell) instruction preceding, or an SXD (store index register in index cell) instruction following, the tagged instructions of the basic blocks that have been treated. For each basic block that has been treated is recorded the required contents of each of the three real index registers for entrance into the block, and the contents upon exit. In the PRED table, entries that have been considered may contain bits calling for interblock LXD's and SXD's, when the exit and entrance conditions across the link do not match.

Now the PRED table is scanned for the highest-frequency link not yet considered. The new region is formed by working both forward over successors and backward over predecessors from this point, always choosing the most frequent remaining path of control. The marking out of a new region is terminated by encountering 1) a basic block which belongs to an opaque region, 2) a basic block which has no remaining links into it (when working backward) or from it (when working forward), or which belongs to a transparent region with no such links remaining, or 3) a basic block which closes a loop. Thus the new region generally includes both basic blocks not hitherto encountered, and entire regions of basic blocks which have already been treated.

The treatment of hitherto untreated basic blocks in the new region is carried out by simulating the action of the program. Three cells are set aside to represent the object machine index registers. As each new tagged instruction is encountered these cells are examined to see

if one of them contains the required tag; if not, the program is searched ahead to determine which of the three index registers is the least undesirable to replace, and a bit is entered in the STAG table calling for an LXD instruction to that index register. When the simulation of a new basic block is finished, the entrance and exit conditions are recorded, and the next item in the new region is considered. If it is a new basic block, the simulation continues; if it is a region, the index register assignment throughout the region is examined to see if a permutation of the index registers would not make it match better, and any remaining mismatch is taken care of by entries in PRED calling for interblock LXD's.

A final concept is that of index register activity. When a symbolic index register is initialized, or when its contents are altered by an indexing instruction, the value of the corresponding index cell falls out of date, and a subsequent LXD will be incorrect without an intervening SXD. This problem is handled by activity bits, which indicate when the index cell is out of date; when an LXD is required the activity bit is interrogated, and if it is on an SXD is called for immediately after the initializing or indexing instruction responsible for the activity, or in the interblock link from the region containing that instruction, depending upon whether the basic block containing that instruction was a new basic block or one in a region already treated.

When the new region has been treated, all of the old regions which belonged to it simply lose their identity; their basic blocks and the hitherto untreated basic blocks become the basic blocks of the new region. Thus at the end of part 1 there is but one single region, and it is the entire program. The high-frequency parts of the program were treated early; the entrance and exit conditions and indeed the whole handling of the index registers reflect primarily the efficiency needs of these high-frequency paths. The loading and unloading of the index registers is therefore as much as possible placed in the low-frequency paths, and the object program time consumed in these operations is thus brought near to a minimum.

CONCLUSION

The preceding sections of this paper have described the language and the translator program of the FORTRAN system. Following are some comments on the system and its application.

Scope of Applicability

The language of the system is intended to be capable of expressing virtually any numerical procedure. Some problems programmed in FORTRAN language to date include: reactor shielding, matrix inversion, numerical integration, tray-to-tray distillation, microwave propagation, radome design, numerical weather prediction, plotting and root location of a quartic, a procedure for playing the game "nim," helicopter design, and a number

of others. The sizes of these first programs range from about 10 FORTRAN statements to well over 1000, or in terms of machine instructions, from about 100 to 7500.

Conciseness and Convenience

The statement of a program in FORTRAN language rather than in machine language or assembly program language is intended to result in a considerable reduction in the amount of thinking, bookkeeping, writing, and time required. In the problems mentioned in the preceding paragraph, the ratio of the number of output machine instructions to the number of input FORTRAN statements for each problem varied between about 4 and 20. (The number of machine instructions does not include any library subroutines and thus represents approximately the number which would need to be hand coded, since FORTRAN does not normally produce programs appreciably longer than corresponding hand-coded ones.) The ratio tends to be high, of course, for problems with many long arithmetic expressions or with complex loop structure and subscript manipulation. The ratio is a rough measure of the conciseness of the language.

The convenience of using FORTRAN language is necessarily more difficult to measure than its conciseness. However the ratio of coding times, assembly program language vs FORTRAN language, gives some indication of the reduction in thinking and bookkeeping as well as in writing. This time reduction ratio appears to range also from about 4 to 20 although it is difficult to estimate accurately. The largest ratios are usually obtained by those problems with complex loops and subscript manipulation as a result of the planning of indexing and bookkeeping procedures by the translator rather than by the programmer.

Education

It is considerably easier to teach people untrained in the use of computers how to write programs in FORTRAN language than it is to teach them machine language. A FORTRAN manual specifically designed as a teaching tool will be available soon. Despite the unavailability of this manual, a number of successful courses for nonprogrammers, ranging from one to three days, have been completed using only the present reference manual.

Debugging

The structure of FORTRAN statements is such that the translator can detect and indicate many errors which may occur in a FORTRAN-language program. Furthermore, the nature of the language makes it possible to write programs with far fewer errors than are to be expected in machine-language programs.

Of course, it is only necessary to obtain a correct FORTRAN-language program for a problem, therefore all debugging efforts are directed toward this end. Any

errors in the translator program or any machine malfunction during the process of translation will be detected and corrected by procedures distinct from the process of debugging a particular FORTRAN program.

In order to produce a program with built-in debugging facilities, it is a simple matter for the programmer to write various PRINT statements, which cause "snapshots" of pertinent information to be taken at appropriate points in his procedure, and insert these in the deck of cards comprising his original FORTRAN program. After compiling this program, running the resulting machine program, and comparing the resulting snapshots with hand-calculated or known values, the programmer can localize the specific area in his FORTRAN program which is causing the difficulty. After making the appropriate corrections in the FORTRAN program he may remove the snapshot cards and recompile the final program or leave them in and recompile if the program is not yet fully checked.

Experience in debugging FORTRAN programs to date has been somewhat clouded by the simultaneous process of debugging the translator program. However, it becomes clear that most errors in FORTRAN programs are detected in the process of translation. So far, those programs having errors undetected by the translator have been corrected with ease by examining the FORTRAN program and the data output of the machine program.

Method of Translation

In general the translation of a FORTRAN program to a machine-language program is characterized by the fact that each piece of the output program has been constructed, instruction by instruction, so as not only to produce an efficient piece locally but also to fit efficiently into its context as a result of many considerations of the structure of its neighboring pieces and of the entire program. With the exception of subroutines (corresponding to various functions and input-output statements appearing in the FORTRAN program), the output program does not contain long precoded instruction sequences with parameters inserted during translation. Such instruction sequences must be designed to do a variety of related tasks and are often not efficient in particular cases to which they are applied. FORTRAN-written programs seldom contain sequences of even three instructions whose operation parts alone could be considered a precoded "skeleton."

There are a number of interesting observations concerning FORTRAN-written programs which may throw some light on the nature of the translation process. Many object programs, for example, contain a large number of instructions which are not attributable to any particular statement in the original FORTRAN program. Even transfers of control will appear which do not correspond to any control statement (*e.g.*, DO, IF, GO TO) in the original program. The instructions arising from an arithmetic expression are optimally

arranged, often in a surprisingly different sequence than the expression would lead one to expect. Depending on its context, the same DO statement may give rise to no instructions or to several complicated groups of instructions located at different points in the program.

While it is felt that the ability of the system to translate algebraic expressions provides an important and necessary convenience, its ability to treat subscripted variables, DO statements, and the various input-output and FORMAT statements often provides even more significant conveniences.

In any case, the major part of the translator program is devoted to handling these last mentioned facilities rather than to translating arithmetic expressions. (The near-optimal treatment of arithmetic expressions is simply not as complex a task as a similar treatment of "housekeeping" operations.) A list of the approximate number of instructions in each of the six sections of the translator will give a crude picture of the effort expended in each area. (Recall that Section 1 completely treats

arithmetic statements in addition to performing a number of other tasks.)

<i>Section Number</i>	<i>Number of Instructions</i>
1	5500
2	6000
3	2500
4	3000
5	5000
6	2000

The generality and complexity of some of the techniques employed to achieve efficient output programs may often be superfluous in many common applications. However the use of such techniques should enable the FORTRAN system to produce efficient programs for important problems which involve complex and unusual procedures. In any case the intellectual satisfaction of having formulated and solved some difficult problems of translation and the knowledge and experience acquired in the process are themselves almost a sufficient reward for the long effort expended on the FORTRAN project.

Editor Record Number	Description of Subroutine	Last Card No.	TRA Address	Load Address	Contents of TR Word	Contents of First Word	Last Address
SECTION 1							
FN090	Clear Drum	001	30	30	053400 100037	070000 000001	40
FN100	Common Storage	83	4	30		050000 100001	3437
FN110	Write Drum	03	4	7700		053400 102105	7763
FN120	State D	48	7700	3440	053400 102105	053400 103163	5370
FN130	State C	35	7703	3440	053400 102103	056000 002102	5030
FN140	State B	45	7706	3440	053400 102104	076000 000140	5351
FN150	State A	79	7711	3440	053400 102106	077200 000222	6750
FN160	Diagnostic	85	3472	3472	300000 403510	300000 403510	7136
SECTION 1'							
FN170	Drum tables to tape	19	4	6613		076600 000302	7407
FN180	Program constants and subroutine	06	4	1666		000000 000000	2072
FN190	Subroutine	02	4	30		063400 200102	103
FN200	Program constant	01	6613	3177	076600 000302	000000 000000	3177
FN210	Part I AMW	20	104	104	053400 101707	053400 101707	753
FN220	Part II AMW	12	104	104	007400 101710	007400 101710	502
SECTION 2							
FN230	Block 1	41	5522	5474	077200 000224	000000 000000	7320
FN240	Block 2, RELCON state	24	4	5566	1 - CS	053400 100030	6565
FN250	Drum set up	02	7730	7730	053400 207766	050000 003777	7773
FN260	Block 2, normal state	26	7732	5566	053400 207765	063400 405622	6650
FN265			4	7742		076600 000333	7761
FN270	Block 2, common	42	4012	3751	077200 000223	000000 000000	5565
FN280	Block 3, common and Part 1	07	4	6665		000000 000000	7073
FN290	Block 3, Part 2	05	6721	7616	050000 007776	076600 000303	7774
FN300	Block 3A		6721	6721	053400 100733	053400 100733	7113
FN310	Block 3B		6721	6721	053400 100733	053400 100733	7613
FN320	Block 3C		6721	6721	053400 100733	053400 100733	7232
FN330	Block 4, RELCON		470	450	050000 001430	000000 000000	1443
FN340	Block 5, Initializing	006	30	30	053400 105377	053400 105377	217
FN350	Block 5, Alpha cycle	055	62	5256	053400 200145	063400 406376	7620
FN360	Block 5, Beta cycle and common	110	66	3646	053400 200072	053400 105374	6637
FN370	Instruction inversion	005	30	30	077200 000224	077200 000224	162
SECTION 3							
FN380	Merges DO file and COMPAIL file	004	4	6350		050000 102136	6520
FN390	Merges DO file and COMPAIL file	050	30	30	076200 000222	076200 000222	2135
FN400	Creates TIFGO file	005	4	7200		063400 407217	7324
FN410	Creates TIFGO file	055	30	30	053400 402225	053400 402225	2367
FN420	Merges TIFGO file and file resulting from 1	026	30	30	053400 401055	053400 401055	1122

TABLE I. STRUCTURE OF SYSTEM TAPE, FORTRAN I, PAGE 1

Editor Record Number	Description of Subroutine	Last Card No.	TRA Address	Load Address	Contents of TR Word	Contents of First Word	Last Address
SECTION 4							
FN430	Part I	33	107	30	002000 000107	000000 000000	1305
FN440	Part II	26	4	33		002000 001062	1104
FN450		04	1063	3064	002000 001062	060100 000147	3210
FN460	Part III	12	111	33	002000 000240	002000 000240	437
FN470	Part IV	04	42	33	002000 000042	002000 000042	161
FN480	Part V	10	42	33	002000 000042	002000 000042	205
FN490	Part VI	23	100	33	002000 000101	002000 000101	1021
SECTION 5 (TAG ANALYSIS)							
FN500	Part IA	121	4	30		076700 000022	5672
FN510	Part IB	04	15651	15651	053400 100356	053400 100356	16001
FN520	Part 2	11	314	317	076200 000223	000000 077777	650
FN530	Part 3A	08	314	320	076200 000222	076200 000222	653
FN550	Part 4	99	3470	317	001622 000000	000000 000003	4463
SECTION 5'							
FN560		19	30	30		076400 000203	444
FN570		02	4	1033		050000 000000	1063
FN580		01	4	4020		000000 000000	4020
FN590		01	30	7000	076400 000203	076000 000006	7013
SECTION 6							
FN600		59	166	30	076200 000304	000000 200000	205
FN610		10	166	166	077200 000202	077200 000202	501
FN620		08	166	166	076200 000202	076200 000202	424
FN630		08	166	166	053400 100145	053400 100145	434
FN640		15	166	166	053400 100146	053400 100146	664
FN650		08	166	166	053400 100111	053400 100111	424
FN660		09	166	166	076200 000303	076200 000303	472
FN670		08	166	166	076200 000304	076200 000304	450
FN680		18	166	166	076200 000221	076200 000221	1001
FN690		END OF FILE					
REWIND SYSTEMS TAPE							
FN000			1-CS	(Not in edit deck)			
FN010		02	335	110	077200 000222	076200 000321	357
FN020		08	4	1400		056000 001412	2113
FN030		16	166	166	077200 000203	056000 001412	1024
FN040		19	166	166	077200 000204	056000 001412	474
FN050		10	166	166	076000 000162	056000 001412	1011
FN060		19	4	77766		070000 000001	77777
FN070		01	74	74	077200 000202	070000 000001	309
FN080		07	FILE MARK (END OF FILE)				

TABLE I. STRUCTURE OF SYSTEM TAPE, FORTRAN I, PAGE 2

Record Number	Description of Record	Transfer Address 8L Decrement	Load Address 8L Address	Last Address 8R Address	Contents of Transfer Word	Contents of Load Word
FILE 1						
000	1-CS (Loc's 0-27)	----	0000	0027	----- - -----	-0 53400 1 00000
001	Card to Tape	0342	0110	0416	0 77200 0 00202	0 76200 0 00321
SECTION 6						
002	Diag. Caller for Rec. 115	4000	4000	4021	-0 63400 2 00000	-0 63400 2 00000
003	CIT to SAP Conver.	0210	0210	1200	0 50000 0 00162	0 50000 0 00162
004	Diag. Caller for Rec. 003	1500	1500	1521	-0 63400 2 00000	-0 63400 2 00000
005	On-Line Print	0210	0210	0551	0 77200 0 00202	0 77200 0 00202
006	Diag. Caller for Rec. 005	1500	1500	1521	-0 63400 2 00000	-0 63400 2 00000
007	Tape 3, 7 to 2, 6	0210	0210	1372	0 77200 0 00202	0 77200 0 00202
008	Diag. Caller for Rec. 007	1500	1500	1521	-0 63400 2 00000	-0 63400 2 00000
009	Successful Compilation	0030	0030	0037	-0 76000 0 00007	-0 76000 0 00007
010	Source Program Error	0030	0030	0056	-0 76000 0 00007	-0 76000 0 00007
011		----	----	----		
FILE 2						
SECTION 1 (4k)						
012	Batch Compilation Monitor	0030	0030	0620	0 76000 0 00166	0 76000 0 00166
013	Machine Error	0030	0030	0035	0 00000 0 00031	0 00000 0 00031
014	Common (4k)	0004	0030	3437	-0 53400 1 00027	0 00000 0 00000
014A	Delete (8k) Common, Initial, and State A.	0004	0030	0000	----- - -----	----- - -----
015	Write Drum (Init.)	0004	0471	0600	-0 53400 1 00027	0 53400 1 00575
016	State D (4k)	0471	3440	6157	0 53400 1 00575	-0 53400 1 01117
016A	Delete (8k) States B, C, and D.	0471	6323	0000	----- - -----	----- - -----
017	State C (4k)	0504	3440	5043	0 53400 4 01407	0 56000 0 01406
018	State B (4k)	0506	3440	5214	0 53400 4 01410	0 76000 0 00140
019	State A (4k)	0510	3440	7306	0 53400 4 01412	-0 53400 4 02575
020*	Diagnostic for Sec. 1	13440	13440	17777	3 00000 4 13543	3 00000 4 13543
* Record 020 Uses Modulo Addressing.						
SECTION 1'						
021	Common	0004	4550	7760	-0 53400 1 00027	0 00000 0 00000
022	Part A	1146	1146	3161	0 76100 0 00000	0 76100 0 00000
023	Diag. Caller for Rec. 022	7755	7755	7776	-0 63400 2 00000	-0 63400 2 00000
024	Part B	0507	0507	1613	0 76100 0 00000	0 76100 0 00000
025	Diag. Caller for Rec. 024	7755	7755	7776	-0 63400 2 00000	-0 63400 2 00000
026	Section 1'	0031	0031	1302	0 77200 0 00202	0 77200 0 00202
SECTION 2						
027	Block 1	5522	5474	7320	0 77200 0 00224	0 00000 0 00000
028	Diag. Caller for Rec. 027	7400	7400	7421	-0 63400 2 00000	-0 63400 2 00000
029	Drum Setup	0004	7730	7775	-0 53400 1 00027	-0 53400 2 07770
030	Block 2 - RELCON	7730	5566	6565	-0 53400 2 07770	-0 53400 1 00030
031	Diag. Caller for Rec. 029 and 030	7400	7400	7421	-0 63400 2 00000	-0 63400 2 00000
032	Block 2 - Normal	7732	5566	6650	-0 53400 2 07767	-0 63400 4 05622
033	Diag. Caller for Rec. 032	7400	7400	7421	-0 63400 2 00000	-0 63400 2 00000
034	Block 2 - Common	4012	3751	5565	0 77200 0 00223	0 00000 0 00000
035	Diag. Caller for Rec. 034	7756	7756	7777	-0 63400 2 00000	-0 63400 2 00000

Note: All Record Numbers suffixed by an "A" are 8k records.

TABLE II. 704 FORTRAN II, PAGE 1

Record Number	Description of Record	Transfer Address 8L Decrement	Load Address 8L Address	Last Address 8R Address	Contents of Transfer Word	Contents of Load Word
SECTION 2 (cont'd)						
036	Block 3 - Common, Part 1	0004	6665	7073	-0 53400 1 00027	0 00000 0 00000
037	Block 3 - Part 2	6721	7614	7774	0 50000 0 07776	-0 63400 1 77777
038	Diag. Caller for Rec. 036 and 037	0300	0300	0321	-0 63400 2 00000	-0 63400 2 00000
039	Block 3A	6721	6721	7113	-0 53400 1 00733	-0 53400 1 00733
040	Diag. Caller for Rec. 039	0300	0300	0321	-0 63400 2 00000	-0 63400 2 00000
041	Block 3B	6721	6721	7613	-0 53400 1 00733	-0 53400 1 00733
042	Diag. Caller for Rec. 041	0300	0300	0321	-0 63400 2 00000	-0 63400 2 00000
043	Block 3C	6721	6721	7232	-0 53400 1 00733	-0 53400 1 00733
044	Diag. Caller for Rec. 043	7400	7400	7421	-0 63400 2 00000	-0 63400 2 00000
045	Block 4 - RELCON	0470	0450	1443	0 50000 0 01430	+010000000001
046	Diag. Caller for Rec. 045	2000	2000	2021	-0 63400 2 00000	-0 63400 2 00000
047	Block 5 - Initialization	0030	0030	0217	0 53400 1 00131	0 53400 1 00131
048	Diag. Caller for Rec. 047	2000	2000	2021	-0 63400 2 00000	-0 63400 2 00000
049	Block 5 - Alpha	0062	5256	7620	0 53400 2 00146	-0 63400 4 06252
050	Diag. Caller for Rec. 049	6000	6000	6021	-0 63400 2 00000	-0 63400 2 00000
051	Block 5 - Beta and Common	0066	3646	6637	0 53400 2 00126	000000000000
052	Diag. Caller for Rec. 051	6000	6000	6021	-0 63400 2 00000	-0 63400 2 00000
053	Block 6 - Inversion	0030	0030	0162	0 77200 0 00224	0 77200 0 00224
054	Diag. Caller for Rec. 053	7000	7000	7021	-0 63400 2 00000	-0 63400 2 00000
SECTION 3						
055	Open Subroutines	0004	7071	7777	-0 53400 1 00027	2 00001 4 01306
056	Part 1 of Merge	0030	0030	2326	0 53400 1 02164	0 53400 1 02164
057	Diag. Caller for Rec. 055 and 056	7755	7755	7776	-0 63400 2 00000	-0 63400 2 00000
058	Part 2 of Merge	0030	0030	2367	0 53400 4 02274	0 53400 4 02274
059	Diag. Caller for Rec. 058	7755	7755	7776	-0 63400 2 00000	-0 63400 2 00000
060	Part 3 of Merge	0030	0030	2715	0 53400 4 01202	0 53400 4 01202
061	Diag. Caller for Rec. 060	7755	7755	7776	-0 63400 2 00000	-0 63400 2 00000
SECTION 4 (4k)						
062	Part 1	0112	0030	1327	0 77200 0 00224	000000000000
063	Diag. Caller for Rec. 062	1400	1400	1421	-0 63400 2 00000	-0 63400 2 00000
064	Part 2, First Rec.	0004	0033	1104	-0 53400 1 00027	0 00000 0 00000
065	Part 2, Second Rec.	1063	3064	3210	-0 53400 1 00551	-0 63400 4 00122
066	Diag. Caller for Rec. 064 and 065	3211	3211	3232	-0 63400 2 00000	-0 63400 2 00000
067	Part 3	0111	0033	0437	-0 54300 3 07774	000000000000
068	Diag. Caller for Rec. 067	0440	0440	0461	-0 63400 2 00000	-0 63400 2 00000
069	Part 4	0042	0033	0161	-0 53400 1 00031	0 00001 0 00000
070	Diag. Caller for Rec. 069	0162	0162	0203	-0 63400 2 00000	-0 63400 2 00000
071	Part 5	0042	0033	0205	0 50000 0 00032	0 00000 7 00000
072	Diag. Caller for Rec. 071	0444	0444	0465	-0 63400 2 00000	-0 63400 2 00000
073	Part 6	0100	0033	1021	0 77200 0 00224	000000000000
073A	Delete (8k) Part 4	0100	0033	0000	-----	-----
074	Diag. Caller for Rec. 073	1022	1022	1043	-0 63400 2 00000	-0 63400 2 00000
SECTION 4 (8k)						
062 through 072	Same Records as used by the 4k Version					

Note: All Record Numbers suffixed by an "A" are 8k records.

TABLE II. 704 FORTRAN II, PAGE 2

Record Number	Description of Record	Transfer Address 8L Decrement	Load Address 8L Address	Last Address 8R Address	Contents of Transfer Word	Contents of Load Word
SECTION 4 (8k) (cont'd)						
073	Delete (4k) Part 6	0000	0001	0000	-----	-----
073A	Part 6 (8k)	0100	0033	1021	0 77200 0 00224	000000000000
074	Same as 4k					
SECTION 5 (4k)						
075	Part 1A - Optimize	0004	0030	5215	-0 53400 1 00027	0 76200 0 00221
075A	Delete (8k) Part 1A	0004	0030	0000	-----	-----
076	Part 1B - Initialize and predict Limit	7337	7337	7470	0 54300 1 00362	0 54300 1 00362
076A	Delete (8k) Part 1B	15674	15674	0000	-----	-----
077	Diag. Caller for Rec. 075 and 076	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
078	Part 1C - Succ. Limit	4740	4740	4773	-0 53400 1 00103	-0 53400 1 00103
078A	Delete (8k) Part 1C	4740	4740	0000	-----	-----
079	Diag. Caller for Rec. 078	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
080	Part 1D - Pred. UNDO	4740	4740	4773	-0 53400 1 00103	-0 53400 1 00103
080A	Delete (8k) Part 1D	4740	4740	0000	-----	-----
081	Diag. Caller for Rec. 080	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
082	Part 1E - Succ. UNDO	4740	4740	4773	-0 53400 1 00103	-0 53400 1 00103
082A	Delete (8k) Part 1E	4740	4740	0000	-----	-----
083	Diag. Caller for Rec. 082	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
084	Part 2 - Permute	0320	0317	0655	0 76200 0 00223	0 00000 0 77777
084A	Delete (8k) Part 2	0320	0317	0000	-----	-----
085	Diag. Caller for Rec. 084	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
086	Part 3 - GO TO N, ASCONS	0320	0320	0646	0 76200 0 00222	0 76200 0 00222
086A	Delete (8k) Part 3	0320	0320	0000	-----	-----
087	Diag. Caller for Rec. 086	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
088	Part 4 - COMPILE	3541	0317	4515	-0 53400 2 00331	0 00000 0 00003
088A	Delete (8k) Part 4	3541	0317	0000	-----	-----
089	Diag. Caller for Rec. 088	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
SECTION 5'						
090	Entire Section	0030	0030	0444	0 76400 0 00203	0 76400 0 00203
091	Diag. Caller for Rec. 090	0450	0450	0471	-0 63400 2 00000	-0 63400 2 00000
SECTION 6						
092	Pre-6	0037	0037	1500	0 77200 0 00202	0 00000 0 00000
093	Diag. Caller for Rec. 092	1367	1367	1410	-0 63400 2 00000	-0 63400 2 00000
094	Binary Search	0210	0037	0300	0 53400 2 00255	1 00037 2 00037
095	Diag. Caller for Rec. 094	1000	1000	1021	-0 63400 2 00000	-0 63400 2 00000
096	Assign Common	0400	0210	0601	-0 75400 0 00004	1 00001 2 00211
097	Diag. Caller for Rec. 096	1000	1000	1021	-0 63400 2 00000	-0 63400 2 00000
098	Equiv - DIM	0400	0400	0647	0 50000 0 00171	0 50000 0 00171
099	Diag. Caller for Rec. 098	1000	1000	1021	-0 63400 2 00000	-0 63400 2 00000
100	Common Mapping	0210	0210	0461	0 77200 0 00202	0 77200 0 00202
101	Fortran FTN Assn	0210	0210	0424	-0 53400 1 00154	-0 53400 1 00154
102	Diag. Caller for Rec. 100 and 101	1000	1000	1021	-0 63400 2 00000	-0 63400 2 00000
103	First Pass CIT	0210	0210	0700	-0 53400 1 00155	-0 53400 1 00155
104	Diag. Caller for Rec. 103	1125	1125	1146	-0 63400 2 00000	-0 63400 2 00000
105	Map Fortran Funct.	0210	0210	0460	0 53400 4 00402	0 53400 4 00402
106	Diag. Caller for Rec. 105	1125	1125	1146	-0 63400 2 00000	-0 63400 2 00000

Note: All Record Numbers suffixed by an "A" are 8k records.

TABLE II. 704 FORTRAN II, PAGE 3

Record Number	Description of Record	Transfer Address 8L Decrement	Load Address 8L Address	Last Address 8R Address	Contents of Transfer Word	Contents of Load Word
SECTION 6 (cont'd)						
107	Map EIFN	0210	0210	0500	0 53400 4 00427	0 53400 4 00427
108	Diag. Caller for Rec. 107	0040	0040	0061	-0 63400 2 00000	-0 63400 2 00000
109	Map Program	0210	0210	0550	0 50000 0 00171	0 50000 0 00171
110	Map Other Variables	0210	0210	0670	-0 53400 1 00032	-0 53400 1 00032
111	Write Prog. Card	0210	0210	0244	0 77200 0 00203	0 77200 0 00203
112	OP Tables	0004	1400	2113	-0 53400 1 00027	0 56000 0 01412
113	Second Pass CIT	0210	0210	0735	0 77200 0 00204	0 77200 0 00204
114	Diag. Caller for Rec. 109, 110, 111, 112, 113	0040	0040	0061	-0 63400 2 00000	-0 63400 2 00000
115	Library Search and Punch	0210	0210	4400	0 53400 1 00544	0 53400 1 00544
SECTION 1 (8k)						
014	Delete (4k) Common	0000	0001	0000	-----	-----
014A	Common, Initial, and State A (8k)	0004	0030	6322	-0 53400 1 00027	0 00000 0 00000
015	Delete (4k) Write Drum and Initial	0000	0001	0000	-----	-----
016	Delete (4k) State D	0000	0001	0000	-----	-----
016A	States B, C, and D (8k)	0471	6323	12764	0 53400 1 00504	0 76000 0 00140
017	Delete (4k) State C	0000	0001	0000	-----	-----
018	Delete (4k) State B	0000	0001	0000	-----	-----
019	Delete (4k) State A	0000	0001	0000	-----	-----
020*	Diagnostic for Sec. 1	13440	13440	17777	3 00000 4 13543	3 00000 4 13543
* Record 020 Uses Modulo Addressing.						
SECTION 5 (8k)						
075	Delete (4k) Part 1A	0000	0001	0000	-----	-----
075A	Part 1A - Optimize	0004	0030	5715	-0 53400 1 00027	0 76200 0 00221
076	Delete (4k) Part 1B	0000	0001	0000	-----	-----
076A	Part 1B - Initialize and Pred. Limit	15674	15674	16025	0 53400 1 00362	0 53400 1 00362
077	Diag. Caller for Rec. 075 and 076A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
078	Delete (4k) Part 1C	0000	0001	0000	-----	-----
078A	Part 1C - Succ. Limit	4740	4740	4773	-0 53400 1 00103	-0 53400 1 00103
079	Diag. Caller for Rec. 078A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
080	Delete (4k) Part 1D	0000	0001	0000	-----	-----
080A	Part 1D - Pred. Undo	4740	4740	4773	-0 53400 1 00103	-0 53400 1 00103
081	Diag. Caller for Rec. 080A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
082	Delete (4k) Part 1E	0000	0001	0000	-----	-----
082A	Part 1E - Succ. Undo	4740	4740	4773	-0 53400 1 00103	-0 53400 1 00103
083	Diag. Caller for Rec. 082A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
084	Delete (4k) Part 2	0000	0001	0000	-----	-----
084A	Part 2 - Permute	0320	0317	0655	0 76200 0 00223	0 00000 0 77777
085	Diag. Caller for Rec. 084A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
086	Delete (4k) Part 3	0000	0001	0000	-----	-----
086A	Part 3 - GO TO N, ASCONS	0320	0320	0647	0 76200 0 00222	0 76200 0 00222
087	Diag. Caller for Rec. 086A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000
088	Delete (4k) Part 4	0000	0001	0000	-----	-----
088A	Part 4 - COMPILE	3541	0317	4516	-0 53400 2 00331	0 00000 0 00003
089	Diag. Caller for Rec. 088A	77750	77750	77771	-0 63400 2 00000	-0 63400 2 00000

Note: All Record Numbers suffixed by an "A" are 8k records.

TABLE II. 704 FORTRAN II, PAGE 4



