April, 1960

SYSTEMS MANUAL

FOR 704 FORTRAN

AND 709 FORTRAN

Applied Programming Department
International Business Machines Corporation
590 Madison Avenue
New York 22, New York

TABLE OF CONTENTS

PREFATORY NOTE

This manual is an attempt to fulfill a long standing, much-pressed request. That is, a request for an over-all, comprehensive explanation of the workings of the entire Fortran System. This includes, in addition to the compiler proper, the monitor, the editor programs, and other corollary routines. It should be noted at the outset, however, that there are a number of difficulties involved in such a presentation. We want to take note of them immediately so that you can better appreciate the form and organization of the manual that follows.

First and foremost, Fortran is a vast, comprehensive system. This, alone, provides its own difficulties. It means that any description of its workings can not be subsumed under the directional efforts of a single individual who understands it all. One individual could not know all the details and subtleties comprising the insides of all the sections marked off by the fourteen Roman numerals of this manual. We have chosen to make the attempt to bring you many of the fine points of the system; this is done by having the "expert" on each of the sections do the writing for that section. The price that must be paid for this approach is obvious: a single style of presentation and a singly oriented organization cannot easily be obtained.

A certain lack of uniformity of the level of generalization used in the various descriptions results. We trust this will be understood. In some cases this lack of uniformity results from the nature of the subject matter; in others, it results from the difficulty described. We attempt to minimize this difficulty by having an introduction which discusses the main points of each of the sections on approximately the same level, and, in the case of section two of the compiler, having a general level discussion then a detailed description.

Certain redundancies, of course, must result. We do not apologize for these redundancies; rather, we suspect you will find them of value. As a matter of fact, in a manual of this kind, repetition will prove useful, especially since each added treatment of a subject matter will present it from a unique viewpoint. What we do apologize for here is not having more full cross-referencing.

The descriptions are kept on a general level. We deliberately have avoided making references to machine and tape locations. This is in line with our regarding this treatment as an explanation of the system from a logical standpoint. In other words, we are presenting what is permanent -- or relatively permanent -- ignoring those things which are subject to momentary change, such as absolute core locations. Futhermore, reference can

be made to the listings, of course, for supplementation along this line. A further advantage of this is that it means we can, in general, give simultaneous treatment to the 704 and 709 Fortran systems. With respect to this, however, it should be noted that we orientate our discussion primarily to the 709 system, making reference to the manner in which the 704 system differs.

We wish to remind you, at this point, that the Fortran reference and operations manuals, particularly the latter, comprise useful supplements to the present discussion. In addition, an excellent paper on the compiler is included in the Proceedings of the Western Joint Computer Conference in Los Angeles, California, of February 1957.

I

INTRODUCTION

1. The primitive elements of the Fortran system are the master tape and appropriate editor deck. With these two a system tape is made. This is then used in the operation of the system.

2. In both 704 Fortran and 709 Fortran the system tape consists of four files. In 704 Fortran, the first two files are compiler files. In 709 Fortran, the first file is the monitor file and the second is the compiler file. In both the systems the third file is the library subroutine file and the fourth is the diagnostic file. When 709 Fortran is not being used in the monitor mode (i. e., a single compile only is occurring) the first record only of the first file is used. This is the Card-Tape Simulator.

3. The system tape, itself, is manipulated by the tape record caller routine, 1-CS. This sits in lower core all during compilation and calls in the succeeding record from the system tape. The compiler records are always called in in sequence. Once a system record has been called and executed, it is not called again. It should be noted that in 704 Fortran some of the records of the first file, comprising the compiler, are not executed until after the records of the second file. These are the records comprising the second pass of the section 6 assembly program. Records of the compiler may be called in singly or in a string, consisting of two or three records.

Each record has a control word telling whether control, after the record is read in, is to go back to 1-CS or to the executive program record itself. If control goes to 1-CS, it means that another record in the string is to be read from the system tape and execution recommences. This control word, incidentally, is contained in the information of the Master Record Card, corresponding to it, in the Editor Deck (see XIV).

4. What follows is a brief over all survey of the Fortran compiler proper. It will attempt to serve as the coordinating unit for the separate detailed write-ups, covering each of the sections of Fortran, which follow. As mentioned in the prefatory note, the material will necessarily cross-cut some of the material of the specific write-ups; it will, however, in most cases, be at a different level of generality.

a. The six sections. As is now fairly well known, Fortran falls naturally into six main divisions, which we call sections. These sections are always executed sequentially. There is never a return to one section once it has been relinquished to go on to its successor. In addition to the six primary sections, there are four secondary sections. These should, however, be considered as nothing more than appendages to the primary sections. These are sections 1', 1'',

5', and pre-6.

Fortran may conceptually be considered as falling into two divisions: the first, comprised by sections 1, 2, and 3; and the second, by sections 4, 5, and 6. This is because at the end of section 3, the entire object program is essentially compiled. It is, in fact, compiled except for the fact that it exists in the C. I. T. format and that it has symbolic tags (reference to index registers) instead of absolute tags. It is then the job of the remaining three sections to remedy these two features. Sections 4 and 5 handle the task of inserting the absolute tag references for the symbolic tag references. This, of course, includes the obligation to insert the necessary loading and saving index register instructions. Section 6, then, places the instructions in the C. I. T. format into the proper relocatable binary format.

As for the first three sections, it may be considered that the first two of these do the entire task of source program analysis. This task includes performing most of the instruction (C. I. T.) compilation. With reference to some of the instructions, however, sections 1 and 2 simply compile information, in tabular form, to pass on to section 3, which uses these as the key to insert the proper instructions. Because the analyses of sections 1 and 2 are independent, the C. I. T's compiled are kept in separate files, which must subsequently be merged. Section 3, therefore, has the task of performing this merge as well as the second merge implied by the instruction file which it, itself, creates. Both section 3 and the last part of section 5, because of their position at the end of necessary primary analyses, perform certain optimizing tasks consisting mostly of removing or inserting certain instructions.

It is well to note that the Fortran compiler makes extensive use of tables. These may be considered as of two types: those which are made up directly from the source program statements, and those which result from further analysis. It is the former class of tables which are mostly included in the reference manual list of tables and their size limitations. The latter class do, in some cases, impose further size limitations. Most tables are passed on from one section to another; some, however, are created purely for use within a section. The source program statements, once scanned, are placed in tabular form and the source program statements are not referred to again.

With one exception, Fortran may be considered as a one pass system. That is, it looks at the source program only once, and it makes a scan of each statement once only. From then on, references are to tables only. The exception noted is in section 1 of the 709 systems. In this case a preliminary scan is made to separate the non-executable from

the executable statements.

Among all the information placed in tabular form, it would be well for the reader to keep in mind the so-called C. I. T. table -- Compiled Instruction Table. This becomes central for it is, indeed, the ultimate object of the compiler. Instructions, throughout the system, until the section 6 assembly, are kept in the four-word per entry table in buffers and records of 100 words (25 instructions). This table is illustrated. We will merely note here that word one contains the internal statement number, with an increment in the address of the word, if necessary; word two contains the mnemonic instruction code, with the address having the decrement value if the code is TXI, TXL, or TIX; word three has the symbolic address (BCD); word four has the relative address (binary) with the address having first, the symbolic tag, then, the absolute tag.

b. Section One. Section one has the primary output of a file of instructions called the Compail file. In addition, it turns out a secondary file of instructions, resulting from any Arithmetic Statement Functions in the problem. The Compail file consists of the following: All the instructions resulting from a translation of the Arithmetic Statements. These arithmetic instructions, of course, refer to symbolic tags in the word four address. Also included in this file are a partial translation of the IF and GO TO Statements, the subprogram definition statements, and the input/output statements.

With respect to the IF and GO TO Statements, section one compiles the necessary test instructions, but it cannot compile the transfer instructions. This is because section one does not know whether any given IF and GO TO statement is in the range of a DO and involves a transfer out of the DO. It is not until this is known that it can be determined whether or not any given transfer should be directly to the statement indicated in the source program or to a set of instructions providing necessary indexing, then the transfer to the specified source program statement. The analysis pertaining to these indexing instructions is left to section two, with the physical instructions being compiled by a second part of section three. In some cases, a C. I. T. is created containing the transfer instruction, but without the address. The address is filled in section three.

With respect to subprogram definition statements, information is gathered which is used by section pre-6 in actually filling in the prologue and index-saving instructions.

With respect to I/O statements, all instructions are compiled except those involving DO's implied by I/O statement lists. After section one has scanned and identified the source program statement, it handles it by transferring to a routine corresponding to it. Then, of course, all information is tabulated and, when possible, compilation performed.

A new internal formula number, incremented by one, is assigned to each input statement, whether that statement is executable or non-executable. Where external statement numbers -- i. e., statement numbers assigned by the source programmer -- exist, the TEIFNO table serves to correlate the external and internal statement numbers.

The greatest division in the handling of statements in section one is between the arithmetic statements and all others. The arithmetic compiler proper constitutes the major portion of section one in number of instructions. The arithmetic compiler in making its scan of the arithmetic formula makes an enormous number of table entries in addition to doing its statement analysis necessary for compilation. Among these tables are the TAU tables, recording subscript combination information, the FORVAL and FORVAR tables recording fixed point variables occurring on the left and right hand sides of arithmetic statements, FIXCON and FLOCON, recording the converted fixed and floating point numbers. It should be noted that the IF and CALL statements fall onto both sides of this division. They are treated as arithmetic statements, with compilation occurring that is not due directly to the arithmetic compiler as well.

The arithmetic compiler is divided into the Scan, Level Analysis, various Optimizing routines, and the Compiler. The Level Analysis sifts out into one group all those algebraic operations which form a unit. A unit is a group that must be performed together and have the same order of binding strength for its operators. 'Plus'and 'minus'are one order of operators,' multiply'and'divide' are another order. The latter has greater binding strength than the former; consequently when they occur in the same context the latter are assigned a higher level number. Needless to say, the use of parentheses in an arithmetic statement is a prime factor in determining units and, hence, level numbers. Optimization occurs to minimize storage accesses. This means that every attempt is made to link one operation to its successor via the machine registers rather than the storage cells. The compilation then proceeds from highest level number to lowest.

c. Section One-Prime. Section One-Prime is the longest of all the secondary sections. It has an enormous number of tasks to perform involving sorting, combining, and moving of table information. Among other things, using the TEIFNO table, it substitutes internal formula numbers for external formula numbers wherever these have had to be retained in tables. This means that from this point on, all Fortran handling is in terms of its own assigned internal statement numbers. An example of where the external statement number has had to be retained till this point is in the TDO table. Here, the number referring to the statement number of the DO itself may be an internal formula number because it is readily known due to the constant updating of the current internal formula number. On the other hand, the number

designating the end of the DO range had to be recorded as an external statement number at the time the TDO table entry was made. This is because it could not then be known how many statements further on in the program the end of the DO range occurred.

d. Section One Double-Prime. Section One Double-Prime is a diagnostic section. It attempts to find as many as possible of the source program errors that were not found by section one. Errors pertaining to the syntax of any of the statements are detected by section one and noted in section one's own diagnostic. Section One Double-Prime, then, finds as many as possible of the source program errors arising from an interrelationship of the statements. These, of course, pertain mainly to flow. Such things as a part of the program that can't be reached or a transfer to a non-executable statement are found here. In general, then, it is true that by the end of one double-prime very nearly all source program errors have been found. Such things as over-lapping DO ranges and certain rare cases of faulty flow still may not be found until sections two and four or five. In addition, it should be mentioned that there are a variety of table overflow errors which may be found after one double-prime. Most of the tables listed in the Reference Manuals are, however, tested prior to this point and any overflow discovered. Both one-prime and one double-prime use the general diagnostic of the fourth file, while section one uses its own diagnostic.

e. Section Two. Section Two has for its primary output a file of compiled instructions called the Compdo file. In addition it, too, creates a secondary file, closed subroutines for the computation of relative constant subscript combination load values. An additional important output are the TRALEV and TRASTO tables, which are essential for section three in producing the third file of Fortran instructions, the TIFGO file.

The Compdo file of instructions contains the computing and indexing instructions for the various subscript combinations contained within DO ranges and any necessary additional tags. These instructions are associated with the beginnings and ends of DO's. At the beginning of DO's they will contain the computing instructions necessary to determine the load value for a tag (subscript combination index register) and the load instructions. In addition, index saving instructions may occur. At the end of DO's these instructions refer to the indexing required to increment subscript combination values for the next DO loop execution, to test whether or not control may pass out of the DO range and, in the latter case, to reset the DO's subscript combinations to their lowest values if control is still in a DO containing the first DO. The instructions performing these three functions are TXI, TXL, and TIX, respectively.

All of these instructions result from the configuration of the combination of DO-nest structure on the one hand and subscript combinations

within the DO-nest, on the other. A DO-nest is defined as any set of DO's all of which are bounded -- contained within -- a single DO. Figuratively, this means that the outside single DO is on level one, the next DO which it contains, on level two, and so forth. Of course, in a single nest there may be more than one DO on any one level greater than level 1. (Please see IV for illustrations.)

Because this discussion of section two will be on the most general level, illustrations will not be provided. However, brief references to the structures of DO-nest in IV may prove useful. What we wish to do here is present in general outline the origin of the problems that section two must solve, which are explained in greater detail in IV.

Section two is a long section and much of its analysis complicated. A great deal of this complexity arises from the desire to provide an highly optimized object program. In other words, some of the problems could have been solved more simply, but at the cost of extra and inefficiently placed object instructions.

In any given DO-nest, section two attempts to place the subscript combination load value computation instructions as far toward the outer DO of the nest as possible. Where these instructions cannot be placed with the DO of level one, a search is carefully made for the point of definition of all the parameters (n_1, n_2, n_3) of the inner or higher level DO's. These values are, of course, necessary for the DO computing instructions. As soon as they are found the next DO serves as the base for the required instructions. This serves the purpose of avoiding the unnecessary repetitions of the computing instructions if they were associated with the inner DO's or the DO's containing the subscript combinations to which they refer.

Another interesting way in which section two seeks maximum optimization is in its attempt to take advantage of the "carry" condition wherever possible. The "carry" condition may be described in this way. There are cases where the configuration of DO's and subscript combinations for a two or three dimensional array makes it possible to consider that a single one dimensional sweep over the array is being made. In other words, the words are being referred to in core storage with the sequential references that a one dimensional array would have. Wherever conditions permit, section two treats such an array as if it were, indeed, single dimensional. The practical effect is to save on indexing instructions. Here, a considerable, sophisticated analysis is required and it is undertaken on the belief that greater object program efficiency makes it worthwhile.

Section two always uses a single tag (index register) for every subscript combination, no matter how complex the subscript combination is. By complexity we refer here to number of subscript symbols and their associated coefficients and addendas. In order to achieve this in all cases

it is sometimes necessary to compile instructions, associated with the DOβ which provide proper reinitialization of the decrement value for the TXL instruction ending the DO on each successive pass through the DO range. The SXDTX table is used in this connection. It is made up in section two and passed on to section three, part one. A configuration of indexing instructions is required for each possible configuration of subscript combinations -- resulting from a permutation of the three possible subscript symbols. This means there are six possible blocks of such indexing instructions.

When a DO, X, is within another DO, Y, and the X DO has been executed its maximum number of times, there are two possible ways of handling the resetting of the X DO's subscript combinations for the next re-entry into the X DO. These, of course, must be reset to the value indicated by the n_1 parameter of the X DO. They may be reset at the point of re-entry into the DO or at the point of departure from the DO. It is the latter course which Fortran has chosen to take. This accounts for the resetting TIX instruction following the TXL instruction terminating the DO. This, in general, produces more efficient object programs, though it does create the problem of handling "resetting" where exit from the X (inner) DO occurs via a transfer to a point in the Y DO rather than through a normal termination of the X DO. To handle this problem, among others, it is necessary to have a third file of instructions, the TIFGO file.

Whenever a transfer is made from a DO to a point completely outside its DO-nest, the values of all the indices of all the DO's within whose range the transfer instruction exists are saved. If, on the other hand, the transfer goes to a point -- really, a level -- outside the immediate DO but still within the DO-nest, section two makes a search to determine if it is necessary to save the index of the immediate DO or DO's from which the transfer occurred. This search is made by checking all FORVAR entries existing on the level of the transfer point. One source of FORVAR table entries was mentioned above; others are listed in II.

With respect to transfers, legal and illegal, section two does catch transfers from within a DO into another DO. It does not, however, stop transfers from entirely outside a DO-nest into a DO. This is to allow programmers to take advantage of the feature enabling them to transfer out of a DO, execute a stretch of program, and return to the point of origin within the DO.

There are certain cases where section two creates a tag; that is, a tag does not correspond to a source program subscript combination. The most obvious case where this is done is where a counter for a DO is required. This is where a DO on I does not have I appearing as a subscript in its range. In this case, an I tag is created. Tags are

also created to handle the conditions described immediately above -- where FORVAR's are involved and the DO does not have its index symbol as a tag anywhere in its range.

But these instances are the simplest cases of added tags: they refer only to a DO index. In other cases, more complicated tags, involving two or three dimensioned subscript combinations, are created. Assume a DO on K within a DO on J within a DO on I, and the appearance of the subscript combination (I, J, K,) on level two; i. e., not in the range of the DO on K, but in the range of the other two DO's. Assume further that the value of K in this subscript combination is set by a transfer from within the DO on K to a point in level two. In this case, if the subscript combination (I, J, K,) does not already exist within the DO on K, one is created and placed there. This tag will then have the value needed at the time of transfer. This situation accounts for another of the six types of TRASTO table entries required to inform section three of the TIFGO file instructions it must compile.

This last case also help to point up another important function of section two: Tag Name Changes. Subscript combinations or tags are given names which are nothing more than the table entry recording the information of the subscript combination. When section one makes up the relevant TAU table entries it does so while examining each statement separately, independent of its position within DO's. Therefore, subscript combinations which syntactically look alike receive the same TAU table entry and,consequently, the same name. However, where subscript combinations receive their definitions and derive their load values independently of each other they are, for all practical purposes, different even though their syntactic appearance is identical. Section two, therefore, must see that the names are changed to assure independent treatment of their indexing. For this purpose, a table called Unedited Change Tag Table is made up. Section three then physically inserts the name changes.

A considerable portion of the work of section two is devoted to the proper handling of subscript combinations which are called relative constants. A relative constant is a subscript symbol not under control of a DO on that symbol. That is, it receives its defintion in some fashion other than the indexing normally associated with a DO. A subscript combination may, therefore, be a pure relative constant (where none of its symbols is under control of a DO), a mixed relative constant (where at least one is not under control of a DO while the others are), or a normal DO-subscript combination (where all subscript symbols are under control of a DO). Each of these three types requires its own mode of treatment by section two. A basic point concerning handling of relative constants is that the computation of the relative constant subscript combination load value is done at the point of definition of the relative constant rather than at the point of use. This decision was made primarily on the supposition that uses of relative constants would occur more often than definitions of relative constants. Placing the required computation instructions at the

point of definition, then, covers a variety of uses.

For pure relative constants, there are two ways in which the computation appears at the point of definition. One is simply by means of the LXD instruction, loading from the relcon (relative constant) cell. This way applies only where the relative constant subscript combination is one dimensional and has no coefficient. The other is by means of a transfer to a closed subroutine, mentioned earlier, which computes the load value. This applies where the relevant subscript combination is greater than one dimension or has a coefficient. Where relcons are of the mixed type, the closed subroutine form will be used in some cases and, in others, the computation will be associated with the DO in the usual way. The deciding factor here is the level of definition of the relcon symbol. If the definition occurs within the same DO as the mixed relcon itself, the closed subroutine must be used. In this case, in order to assure that the closed subroutine has all the subscript symbols available for computation, section two must see to it that the DO subscript symbols of the mixed relcon are stored before the transfer is made. Where the definition of the subscript symbol is outside the DO, the computing instructions are associated with the DO of the next possible higher level DO.

The table FORVAL is the key in determining point of definition of relative constants. Causes of entries in this table were indicated above; others are described in II. Every point of definition is used as the base for a relcon computation (of one of the two forms described above). Section two cannot make the flow analysis necessary to eliminate superfluous points. For example, where I is a relcon and the problem contains two arithmetic statements in which I appears on the left side and only one of them gets executed on the path of flow leading to the I relcon, section two makes the computation instruction entries at both points even though only one of them is effectively valid.

Where possible, one of the subscript combinations appearing in a DO is used to serve as the test tag for the end of the DO; that is, it is referred to by the terminating TXL. Where the DO index, itself, appears as a separate tag (whether because it is a subscript in the source program or section two created one for it), this tag is used to test the end of DO. In all other cases, section two attempts to determine the best tag for use in the end of DO test.

As a result of analyses like those mentioned above, and some others that are indicated in the section two write-up, the COMPDO file is made to contain instructions giving highly efficient handling of DO loops.

f. Section three, first, merges the two existent files of instruction, the Compail and Compdo. It then creates the indexing instructions necessary for each transfer branch originating from a transfer out of a DO. This entire set of instructions is called the TIFGO file. This is then merged

with the FIRSTFILE.

g. The program up to this point assumes an object machine with as many index registers as symbolic tags are used in the section two instructions. Since, however, the machine will have three index registers, it is necessary to substitute assignments of these three for the indefinitely high number of symbolic tags. The object here will be to minimize the number of LXD's and SXD's -- load and save instructions -- required by this fact. By "number" here, we mean not only the number of separate physical instructions, but also the number of executions of them. That is, optimization with respect to time takes precedence over optimization with respect to space. For example, if a tag is used in a very high frequency part of the program (such as the inner DO of a DO-nest three levels deep), and a branch transfer is made to four different areas , each of which requires saving of the tag before it is reused, a single save instruction before transferring out of the high frequency area is logically sufficient. However, our method is to place four separate save instructions at the point of entry to each of the four branch points, thus eliminating the instruction from the path which would require most frequent executions of it.

This case also serves to illustrate some of the problems confronting sections four and five -- the two sections whose concern this task is. It shows that there is a linkage, with respect to index registers, of different parts of the program and that details of the linkage must be known for efficient insertion of load and save instructions. For example, in the above case, the SXD will not be used on any of the four paths where it is not required. Furthermore, a comprehensive knowledge of areas and their expected frequencies of object time flow is necessary. As a corollary to these problems, there is the one of avoiding the SXD instruction for a tag which is no longer to be used. That is, the tag can be efficiently killed by over-loading it in its index register. There is also the problem of knowing when to save an index register when the next use of the tag in it requires a load instruction. If the last reference to this tag is one that changed its value, it must be saved; if the last references did not change its value but merely used its earlier established value, it is not necessary to save. Here, a distinction between active and passive references to tags is necessary.

This entire complex of problems comprise the task of sections four and five. The work required of these sections falls naturally into two divisions, allowing the division of labor between them. Section four informs section five of the divisions of the object program for purposes of flow analysis and the relative frequency of paths of flow over these divisions. Its task is much the lesser of the two sections. Section five then uses this information along with a knowledge of the specific tags required by each of the "divisions" to assign absolute index registers and compile necessary indexing instructions.

Before giving the general discussion of the work of these two sections,

it is well to note how this work was presupposed in the handling of symbolic index registers by the earlier sections of Fortran. Essentially, this can be stated very simply: the earlier sections simply ignored the problem and acted as if as many index registers as were wanted were available. That is, load instructions may appear in sequence up to any number. The assumption is the "saves" necessary to make the "loads" effective will be added later. The important thing to note here is that SXD's and LXD's are not always coupled as the previous discussion might imply. There is an asymmetry between them; the earlier sections have complete freedom with respect to LXD's, very rarely compiling an SXD. On the object program level this difference is reflected in the cells which the SXD's and LXD's address. Section two's instructions, for example, mostly refer to the subscript symbol cells in the regular data area of core storage. On the other hand, section five's instructions always refer to the specially designated erasable area for storage of index registers. These erasable storage cells are referred to as the C) cells. The actual designation is C)i, where i is an increment resulting from the conversion of the symbolic tag name. By means of this device there is co-ordination between section five references to such tag storage cells and whatever section two references are necessary.

h. Section four has for its main task the assembling of four different tables. These are the BBB table, the Predecessor, the Successor table, and the Tag List table. The primary input to section four is the single file of merged C. I. T. 's; section four also uses other tables created earlier. The BBB table is a list of the Basic Blocks of the object program, plus indices referring to each Basic Block's Successors and Predecessors. A Basic Block is the primary unit that section four works with -- it was referred to by the word "division" in g. above. A Basic Block is a stretch of program into which there is only one entrance and from which there is only one exit. "Exit" must here be interpreted in the logical sense; that is, it may consist of more than one transfer instruction, going to a variety of Basic Blocks. Each of these Basic Blocks, then, is a Successor Basic Block. As implied by this, section four must mark off the Basic Blocks of the program and determine the Successor and Predecessor Basic Blocks for any one Basic Block. A BBB entry corresponds to each Basic Block; it has references to the Predecessor and Successor tables denoting its Predecessor and Successor Basic Blocks. But section four's work goes beyond this. It must provide the information to section five concerning frequency of paths of flow. Therefore, the form of the Predecessor and Successor table entries which section four passes on to section five will contain, in addition to the Basic Block reference number, a number denoting relative frequency of transition between the two Basic Blocks. Here, the two Basic Blocks refer to the BBB Basic Block and the Basic Block or Blocks of the Predecessor and Successor table that it designates. In order to achieve these relative frequency numbers, section four performs a simulated flow over the program going from Basic Block to Basic Block.

The major problem here is in determining which Successor Basic Block to go to when, as a result of a conditional transfer, a possibility of more than one Successor Basic Block exists. At this point a "Monte Carlo" technique is used. A random number is generated and, in accordance with the numeric possibilities of succession indicated by the frequency statement entries for that conditional transfer, a particular Successor Basic Block is chosen. The random number is meant to assure that over the long run of the entire simulated flow, the possible Successors will be chosen in the proportions indicated by the Frequency entries. Where no Frequency entry is made by the source programmer, the assumption is that of equal probability for all paths of succession.

Some of the special problems encountered during the performing of this simulated flow are those given by conditional transfers where the conditions are set directly in the source program (such as ASSIGN GO TO's and Sense Light Tests) and DO's involving variable parameters. For both of these additional intermediate tables are necessary. In the case of DO-nests, three general circumstances, involving flow analysis problems, may occur. One is a DO-nest whose DO's all have constant parameters and contain no transfers, another is constant parameters with transfers, and the third is a DO-nest at least one of whose DO's has variable parameters. For the last mentioned circumstance, either the frequency entry for the DO must be used or barring that, a frequency of five is posited for the number of times of repetition of the DO range.

For purposes of the simulated flow, a large number is chosen, which is a function of the number of Basic Blocks and distinct transfer branches occurring in the problem. For every transition between a Basic Block and its Succ essor that is made during the simulation, this number is ticked off by one. The flow ends when this number equals zero.

It should be pointed out, finally, that this simulated flow has nothing whatever to do with the individual instructions of the problem. It is concerned only with Basic Blocks as units and not with the contents of a Basic Block. As far as section four is concerned a Basic Block may actually contain one hundred instructions or two instructions, and these instructions may contain many tags or no tags: section four's treatment of it is the same. It may also be mentioned here that the division into Basic Blocks is based on an examination of the compiled instructions. Of course, the recognition of transfers -- beginning with the letter "T" -- is vital. For this reason, section one finds it necessary to use pseudo-names in the C. I. T. 's of some of its instructions. It does not wish section four to think that these end Basic Blocks when actually they do not.

After the flow analysis is completed, section four assembles the BBB, Predecessor, and Successor tables. These are a summary of the Basic Block flow and relative frequency of this flow. The BBB entries also contain a designation of the type of ending for each Basic Block: absolute transfer, pre-set transfer, conditional transfer, and so forth etc. The

last significant item that each BBB entry contains is an index to the Tag List entries belonging to it. The Tag List table is made up at the end of section four; it is a list of all symbolic tags contained in the C. I. T. 's of the program together with a code designating the type of instruction referring to the tag. The index to this table that is placed in the BBB entry, then tells which tags occur in each Basic Block of the program and how they are used.

i. Section Five must now substitute references to tags 1, 2, and 4 for the symbolic tags which occupy the address portion of word 4 of the C. I. T. 's. As a corollary to this, the loading and saving instructions would be inserted for the appropriate index registers. These will load from and save in the group of cells designated as C)--cells. The information contained in the four tables created for it by section four are sufficient to do this.

To perform this main task, section five operations fall logically into two broad divisions. These are Region Generation and LXD and SXD Assignment.

Region Generation is the method of setting aside a portion of the program, consisting of one or more basic blocks, for independent treatment with respect to index register assignment. After a set of basic blocks have been set aside as a region and treated, it then, as a region, becomes a separate unit liable to be incorporated in a new region along with other basic blocks. The flow configuration of a problem determines when a region itself becomes part of another region. When it does it loses its identity for the new region is an independent and separate unit. Ultimately, of course, all regions and basic blocks become absorbed into a single region which is the entire program. At this point the section five analysis is complete. In referring to "treatment" above, we mean the LXD and SXD Assignment.

There is, then, an interweaving of the operations of the two main divisions of section five, Region Generation and SXD and LXD Assignment. (The second of these divisions is often referred to as the LXing Pass.) The regions grow recursively until the entire problem is one region. At any given time during this recursive treatment, several regions may exist independently or one only may exist.

Priority is given the high frequency path of flow in index register assignment by the manner in which regions are generated. Basic blocks are traced forward and backwards in flow, via the Predecessor and Successor Tables, and those basic blocks are used first whose numeric linkages are highest with other basic blocks, as indicated in the figure on comparative frequency of paths of flow given by section four. When a region has been treated, if all three index registers are assigned to the tags of that region, it is considered to be an opaque region. The tracing of basic blocks and regions, first backwards, then forwards, proceeds until

either a) there are no more untreated linkages, b) an opaque region is encountered, c) a loop is formed. The c) case occurs where the Predecessor or Successor basic block is one already in the string. In this case, all basic blocks not within the scope of the loop are cut off. Where a region encountered during this trace is a transparent region, as distinct from an opaque region, the trace continues by way of the highest frequency untreated link from it or into it, depending upon which direction the trace is taking. Because, by definition, all the index registers of a transparent region have not been used, it is subject to further treatment and, consequently, may be absorbed into the region as a basic block is.

The "treatment" of a region is based on another type of simulated flow through it. This simulated flow affects the symbolic index register usage occurring in the region. In cells representing the three index registers, the symbolic tags are loaded, then comparisons made with successive symbolic tags, as these are revealed in Tag list. When it becomes necessary to save one of the three index registers, a look ahead through Tag list is made to determine which it is preferable to save; that is, which is last used further ahead in the program. It should be noted that two fundamental problems are involved here. One is simply the problem of assignment of index registers; this involves the compilation of LXD's and the choice of an index register. The other is the problem determining when to save an index register when the quantity is subsequently going to be over-written by a load into that index register.

With respect to the second of these two problems, a tag must be saved to initialize the appropriate C) cell for later loading, and to handle "active" index registers. "Activity" is denoted by the type of reference made to the tag in the tag instruction. The Tag List code referring to the tagged instruction tells essentially whether that instruction is active or passive. An active instruction is simply one that changes the value of an index register (such as TXI or LXD) and a passive instruction is one that uses the tag only (such as CLA). Where "activity" is present and a subsequent load will over-write the index register, an SXD is inserted following the last use of the symbolic tag. Activity has meaning applying beyond the context of the immediate region in which it is discovered. It may subsequently be found that a pass on the flow from this region requires the new tag value. Activity for regions, then, must be carefully noted.

As a result of this simulation within a region, the index registers upon entry into a region and upon exit from it are assigned certain symbolic tags. These are noted in the BBB entry for the basic block as its entrance and exit conditions. When a region -- which, of course, has been previously treated -- is encountered a match must be made of the exit conditions of the last basic block with the entrance conditions of the basic block by which they region is entered. Where necessary, permutation of the index registers within the already

treated region takes place to force compliance. If a match cannot be made, LXD's are called for at the head of the region. These LXD's are called inter-block LXD's because they concern the linkage between regions as distinct from basic blocks. There are also inter-block SXD's. These result from activity within a region already treated. The SXD is placed at the head of the region using the active tag. In this way, incidentally, the deployment of save instructions among different low frequencies paths raother than the single save instruction within the high frequency path occurs. This was referred to in g.above.

Continuing to work in this way, from region to region, the high frequency paths of flow naturally receive priority in the assignment of index registers. The SXD's and LXD's are inserted enforcing conformity of the low frequency paths with the already assigned high frequency paths.

During this entire analysis, Section 5 records within tables the information needed to make the actual compilation and insertions of the LXD and SXD instructions. The compilation itself occurs later. A new table, the STAG table, is created for recording these instructions as needed within a region. The necessity for inter-block instructions is recorded in the Predecessor table.

The inter-block instructions, because they are at the head of a region, must take their own location symbols so that transfers may occur to the block. These location symbols are: D), when the instruction is an LXD, and E), when it is an SXD. A TRA instruction may have to be added to bypass these instructions when entry to the block occurs from the part of the progr am immediately preceeding it.

Section Five, also because it makes a pass over the entire program, performs certain small optimizing operations on the compiled program.

j. Section Five prime places the information, which represents program constants, in the CIT format. Section Pre-six does some compilation. This covers mostly the prologue to Fortran sub-programs. Section Six does the final assembly for the program. The Section Six write-up that follows is also presented on two levels of generality.

5. This survey of the Fortran compiler is supplemented in detail by the sections that follow. By means of this survey, some of the details may more easily be inter-related.

FORTRAN II, Section One (704 Version)

This section is the initial processor of the FORTRAN compiler. It makes those entries in the Compiled Instruction Table which are possible at a first level. All information which cannot be processed is recorded in one or more tables.

Input: The input to Section One is the Source Program on a BCD tape. It is a single file.

Output: Tables which may be classified into two groups:

1. Generated by Section One and required for reference. These tables, retained in cores and on drums, are:

DIM1	TAU1	FIXCON	END
DIM2	TAU2	FLOCON	
DIM3	TAU3	FORSUB	

2. Generated by Section One and not required for reference. These tables, written on tape(s) in buffer-sized records, with labels where needed are:

CIT	FORTAG	CLOSUB	NONEXC
TEIFNO	FORVAR	FORMAT	TSTOPS
TDO	FORVAL	SUBDEF	CALLFN
TIFGO	FRET	COMMON	FMTEFN
TRAD	EQUIT	HOLARG	TSKIPS

3. Parameters describing above tables.
4. Residual contents of buffers.

Most tables are simple in format and their meaning and usage fairly obvious. The following discussions of processors will show specific table entries. Briefly the tables are:

NAME	DESCRIPTION
DIM1	one-dimensional arrays
DIM2	two-dimensional arrays
DIM3	three-dimensional arrays
TAU1	one-dimensional subscripts
TAU2	two-dimensional subscripts
TAU3	three-dimensional subscripts
FIXCON	fixed-point constants
FLOCON	floating-point constants
FORSUB	arithmetic statement functions
END	options specified in END statement
TEIFNO	corresponding IFNs and EFNs
TDO	DO statements
TIFGO	IFs, GO TOs, ASSIGN statements
TRAD	GO TO statements
FORTAG	IFNs - I - TAU tags
FORVAR	fixed-point variable usage
FORVAL	fixed-point variable definition
FRET	FREQUENCY statements

EQUIT	EQUIVALENCE statements
CLOSUB	names of closed subroutines referenced
FORMAT	FORMAT statements
SUBDEF	SUBROUTINE or FUNCTION statements
COMMON	COMMON statements
HOLARG	Hollerith arguments in CALL statements
NONEXC	IFNs of non-executable statements
TSTOPS	IFNs of STOP and RETURN statements
CALLFN	first and last IFNs of CALL statements call formula number
FMTEFN	I-O statement references to FORMAT numbers
TSKIPS	IFNs of possible machine language skips

ASSEMBLY routine reads records from the BCD input tape until a statement and all its continuation cards are assembled in an erase-able buffer termed the F- region. This region remains until replaced by the following statement. In order to ascertain that all continuation cards have been read the program reads one record ahead into an area termed FT. Blank cards and comments cards are ignored.

A word of all-ones is written after the last non-blank word in the F- region to serve as an end-of-statement marker.

An internal statement number ∝(IFN) is assigned.

If an external statement number (EFN) appears in the source statement it is converted to binary and following table entry made:

TEIFNO table

word 1	∝(IFN)	∝(EFN)

Any special mode character in cc 1 are isolated and saved.

CLASSIFICATION After assembly each statement is classified according to type. This classification is a two-phase procedure.

I. The statement is classified as arithmetic if:
   1) There exists an = sign not within "(" ")".
   2) This = sign is not followed by a " , " not within "(" ")".
   Control goes to the ARITHMETIC processor.

II. If the statement is not classified as arithmetic by the above procedure it is assumed to be non-arithmetic. The beginning of statement is compared to entries in a dictionary of non-arithmetic statement beginnings. When identified as to type control goes to the appropriate processor. Failure to identify causes a Diagnostic message.

ARITHMETIC Processor

The reader is advised that this preliminary paper does not include a description of the ARITHMETIC Processor. A paper, describing this processor from a theoretical standpoint, may be found in the communications of the Association for Computing Machinery, Vol. 2, No. 2, February, 1959.

DIMENSIONS $V(I_1, \ldots, I_k)$, $V(I_1, \ldots, I_k), \ldots$

The statement is scanned collecting the variable name V and associated specification $(I_1, \ldots, I_k)$ where $K \leq 3$. It is verified that V has not been previously defined in a DIMENSION statement. Dimensionality is based on the number of specifications I_K where $1 \leq K \leq 3$. There are thus three possible cases:

1. $K = 1$. The following table entry is made:

DIM1 table

	Variable Name (BCD)	
word 1		
word 2	0	I_1

2. $K = 2$. The following table entry is made:

DIM2 table

	Variable Name (BCD)	
word 1		
word 2	I_1	I_2

3. $K = 3$. The following table entry is made:

DIM3 table

	Variable Name (BCD)	
word 1		
word 2	I_1	I_2
word 3	0	I_3

This procedure is repeated until all $V(I_1, \ldots, I_k)$ have been processed.

EQUIVALENCE $(V_1(I_1), V_2(I_2), \ldots), (V_i(K_1), \ldots), \ldots$

Each specification of equivalence is scanned. The variable name is collected. The constant, if present, is collected and converted to binary. If not present it is understood to be 1. The following table entry is made:

EQUIT table

	Variable Name (BCD)	
word 1		
word 2		N

where N is the associated constant or 1.

This is repeated for each variable of a specification until the last. On the last such the following table entry is made:

EQUIT table

	Variable Name (BCD)	
word 1		
word 2	-	N

where the - signifies the end of a specification.

The entire procedure is repeated for each specification.

COMMON $V_1, \ldots, V_n$

Each variable name is collected and the following table entry made:

COMMON table

word 1 | Variable Name (BCD)

If the variable name is fixed-point the following table entry is made:

FORVAL table

word 1 | 1 | 0

word 2 | Variable Name (BCD)

The above procedure is repeated for each argument name.

FREQUENCY B_1 $(N_1, N_2, \ldots)$, B_2 $(N_1, N_2, \ldots)$, . . .

The statement is scanned, collecting the statement number Bi. It is converted to binary and the following table entry is made:

FRET table

word 1 | - | 0 | Bi(EFN)

Each branch frequency Ni is collected and converted to binary. The following table entry is made:

FRET table

word 1 | 0 | Ni(EFN)

This is repeated for each branch frequency.

The entire procedure is repeated for each specification.

END $(I_1, \ldots, I_n)$

Each specification is collected and the following table entry made:

END table

word 1 | I

This is repeated for each specification.

FORMAT (. . .)

The following table entry is made:

FORMAT table

word 1 | 1 | ∝(EFN) |

Each word of the FORMAT specification as found in the F - region is made into the following table entry:

FORMAT table

word 1 | Format Specification (BCD) |

The first word, if less than six characters, is prefaced by blanks.

When the entire Format specification has been entered the following table entry is made:

FORMAT table

word 1 | - - - - - all ones - - - - - - - - - |

During the above processing a scan is made of the Format specification for legality of characters and balance of parenthesis (excluding hollerith fields).

DO N I = N1, N2, N3

The termination of the DO range N is collected and converted to binary. The variable I is collected. The parameters N1, N2, N3 are collected. Any constant parameter is converted to binary. N3 is understood to be 1 if not specified. The following table entry is made:

TDO table

word 1	∝(IFN) N(EFN)
word 2	Variable Name (BCD)
word 3	N1
word 4	N2
word 5	N3

where N1, N2, N3 may each be constant or variable and where for each which is variable a 1 is placed in bit 20, 19, 18 respectively of word 1.

IF (. . .) N1, N2, N3

Each branch address N1, N2, N3 is collected and converted to binary. The following table entry is prepared:

TIFGO table

word 1	- ∝(IFN) N1(EFN)
word 2	N2(EFN) N3(EFN)

This entry is held until treatment of the arithmetic expression is completed. If the expression contained any references to subprograms, resulting in the final ∝(IFN)i ≠ ∝(IFN) then such ∝(IFN)i replaces ∝(IFN) in the pending entry. The entry is then made.

The statement is modified by the following transformation.

I is replaced by X
F is replaced by non-BCD character 12
(is replaced by =
) is replaced by non-BCD character 77.

The statement is then treated by the ARITHMETIC processor.

IF ACCUMULATOR OVERFLOW N1, N2

The branch addresses N1, N2 are collected and converted to binary. The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	T O V	0
word 3		0
word 4	0	0

TIFGO table

word 1	∝(IFN)	5
word 2	N1(EFN)	N2(EFN)

IF QUOTIENT OVERFLOW N1, N2

The branch addresses N1, N2 are collected and converted to binary. The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	T Q O	0
word 3		0
word 4	0	0

TIFGO table

word 1	∝(IFN)	5
word 2	N1(EFN)	N2(EFN)

IF DIVIDE CHECK N1, N2

The branch addresses N1, N2 are collected and converted to binary. The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	D C T	0
word 3		0
word 4	0	0

TIFGO table

word 1	∝(IFN)	4
word 2	N1(EFN)	N2(EFN)

SENSE LIGHT I

The sense light designation I is collected. It is converted to binary and added to 140_8. The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	P S E	0
word 3		0
word 4	(140_8+I)	0

IF (SENSE LIGHT I) N1, N2

The sense light designation I is collected. It is converted to binary and added to 140_8. The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	M S E	0
word 3		0
word 4	(140_8+I)	0

The branch addresses N1, N2 are collected and converted to binary. The following table entry is made:

TIFGO table

word 1	∝(IFN)	3
word 2	N1(EFN)	N2(EFN)

IF(SENSE SWITCH I) N1, N2

The sense switch designation I is collected. It is converted to binary and added to 160_8. The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	P S E	0
word 3		0
word 4	(160_8+I)	0

The branch addresses N1, N2 are collected and converted to binary. The following table entry is made:

TIFGO table

word 1	∝(IFN)	3
word 2	N1(EFN)	N2(EFN)

GO TO N

The branch address N is collected and converted to binary.
The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	T R A	0
word 3		0
word 4	0	0

TIFGO table

word 1	∝(IFN)	0
word 2	0	N(EFN)

GO TO (N1, N2, . . . , N_m), I

Each branch address Ni is collected and converted to binary. The following table entry is made:

TRAD table

word 1	0	Ni(EFN)

This is repeated for each Ni until i = m.

The following table entry is made:

TIFGO table

word 1	∝(IFN)	2
word 2	TRAD(Ni)	TRAD(N_m)

where TRAD(Ni) is the complement (table size) of position of Ni in TRAD table.

The variable I is collected and treated by the subscript processor as a one-dimensional subscript (I). The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	T R A	0
word 3		0
word 4	0	1 - T

GO TO N, (I_1, I_2, . . . , I_m)

The variable N is collected. The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	T R A	0
word 3	V a r i a b l e N a m e	(BCD)
word 4	0	0

Each permissible branch address I_k is collected and converted to binary. The following table entry is made:

TRAD table

word 1	0	I_k(EFN)

This is repeated for each I_k until k = m. The following table entry is made:

TIFGO table

word 1	∝(IFN)	1
word 2	TRAD(I_1)	TRAD(I_m)

where TRAD(I_k) is the complement (table size) of position of I_k in TRAD table.

ASSIGN I TO N

The EFN being assigned (I) is collected and converted to binary. The variable N is collected. The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	C L A	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	S T O	0
word 3	V a r i a b l e N a m e	(BCD)
word 4	0	0

TIFGO table

word 1	∝(IFN)	6
word 2	0	I(EFN)

STOP N

The identification N, if any, is collected and converted from octal to binary. The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	H P R	0
word 3		0
word 4	N	0

where N = 0 if not specified.

CIT table

word 1	0	0
word 2	T R A	0
word 3	∝(IFN)	
word 4	0	0

TSTOPS table

word 1	∝(IFN)	∝(EFN)

where ∝(EFN) is the last such encountered.

PAUSE N

The identification N, if any, is collected and converted from octal to binary. The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	H P R	0
word 3		0
word 4	N	0

where N = 0 if not specified.

CONTINUE

The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	B S S	0
word 3		0
word 4	0	0

SUBROUTINE NAME (ARGI, ..., ARGN)

The statement is verified to be the first of the program. The name is collected and the following table entry made:

SUBDEF table

word 1	Subprogram Name (BCD)

Each argument name is collected and the following table entry made:

SUBDEF table

word 1	Argument Name (BCD)

If the argument name is fixed-point the following table entry is made:

FORVAL table

word 1	1	0
word 2	Argument Name (BCD)	

The above procedure is repeated for each argument name. A count is kept of the arguments for use in processing RETURN statements.

FUNCTION NAME (ARGI, ..., ARGN)

This statement is processed in the same manner as SUBROUTINE NAME (ARGI, ..., ARGN). In addition, the subprogram name is retained for use in processing RETURN statements.

CALL NAME (Arg 1, . . . , Arg N)

There are two possible cases:

1. No arguments. The subprogram name is collected.
   The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	Subprogram Name (BCD)	
word 4	0	4

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

CALLFN table

word 1	∝(IFN)	∝(IFN)

2. Some arguments. The statement is modified by the following transformation.
   C is replaced by Z
   A is replaced by non-BCD character 12
   L is replaced by =
   L is replaced by +
   The statement is then treated by the ARITHMETIC processor.

RETURN

There are two possible cases:

1. The RETURN occurs in a program defined by SUBROUTINE.
   The following table entry is made:

   CIT table

word 1		∝(IFN)		0
word 2	L	X	D	0
word 3	$			
word 4		0		1

2. The RETURN occurs in a program defined by FUNCTION.
   The following table entries are made:

   CIT table

word 1		∝(IFN)		0
word 2	C	L	A	0
word 3	S u b p r o g r a m	N a m e	(BCD	
word 4		0		0

   CIT table

word 1		0		0
word 2	L	X	D	0
word 3	$			
word 4		0		1

The following table entries are made:

CIT table

word 1		0		0
word 2	L	X	D	0
word 3	$			
word 4		1		2

CIT table

word 1		0		0
word 2	Q	X	D	0
word 3	$			
word 4		2		4

CIT table

word 1		∝(IFN)1		0
word 2	Q	P	R	0
word 3				0
word 4		(K+1)		0

where ∝(IFN)1 is ∝(IFN) incremented by 1 and
where K is the number of arguments of the subprogram.

CIT table

word 1		0		0
word 2	T	R	A	0
word 3		∝(IFN)1		
word 4		0		0

TSTOPS table

word 1	∝(IFN)1	∝(EFN)

where ∝(EFN) is the last such encountered.

SUBSCRIPT Processor

Subscripts may appear in I/O LISTs or in ARITHMETIC expressions. There exists a closed subroutine to scan these subscripts and make the following table entries:

1. Subscript is one-dimensional:

   TAU 1 table

word 1	C1	0
word 2	Variable Name (BCD)	

   where C1 is coefficient.

2. Subscript is two-dimensional:

   TAU 2 table

word 1	C1	C2
word 2	Variable Name 1 (BCD)	
word 3	Variable Name 2 (BCD)	
word 4	d1	0

   where C1 is first coefficient,
   C2 is second coefficient, and
   d1 is first dimension from DIM2 table.

3. Subscript is three-dimensional:

   TAU 3 table

word 1	C1	C2
word 2	C3	0
word 3	Variable Name 1 (BCD)	
word 4	Variable Name 2 (BCD)	
word 5	Variable Name 3 (BCD)	
word 6	d1	d2

   where C1 is first coefficient,
   C2 is second coefficient,
   C3 is third coefficient;
   d1 is first dimension and
   d2 is second dimension from
   DIM3 table.

For a subscript having one or more variables the following table entry is made.

   FORTAG table

word 1	α(IFN)	I - γ

   where I specifies the dimensionality and γ the position of the entry in respective TAU table.

The I - γ tag is returned to the calling processor.

The addends are used to form a relative address which is returned to the calling processor.

ERROW FLOW TRACE

This feature is optional and normally suppressed. When activated it causes the following table entries to be made at the point of return from a called subprogram, arithmetic library subroutine, or arithmetic statement function.

CIT table

word 1			0		0
word 2		N	T	R	∝(EFN)
word 3	1	7			
word 4			2		0

where ∝(EFN) is the last such encountered.

There are three possible cases for the second table entry.

1. Program being compiled is a main program.

CIT table

word 1			0		0
word 2		P	Z	E	∝(IFN)
word 3					0
word 4			0		0

2. Program being compiled is a subprogram.

CIT table

word 1			0		0
word 2		P	Z	E	∝(IFN)
word 3	$				0
word 4			2		0

3. Portion of program being compiled, whether main or subprogram, is an arithmetic statement function.

CIT table

word 1			0		0
word 2		P	Z	E	∝(IFN)
word 3					0
word 4		77777_8			0

Input-Output Statements

There are thirteen statements pertaining to object program input-output. These may be grouped as follows:

1. Five statements for input-output from-to external medium:
   READ N, LIST
   READ INPUT TAPE I, N, LIST
   PRINT N, LIST
   PUNCH N, LIST
   WRITE OUTPUT TAPE I, N, LIST
2. One statement specifying such external input-output:
   FORMAT (...)
3. Four statements for input-output from-to intermediate storage medium in binary form:
   READ TAPE I, LIST
   READ DRUM I, J, LIST
   WRITE TAPE I, LIST
   WRITE DRUM I, J, LIST
4. Three statements for tape handling not involving data transmission:
   END FILE I
   REWIND I
   BACKSPACE I

where N is format designation,
I is unit designation
J is drum address,
and LIST is a string of variable names to be transmitted.

To avoid a high degree of redundancy the following descriptions of individual processors of data transmission statements will sometimes refer to each other. In general, any such reference will be to a processor previously discussed.

All statements having a LIST cause the following table entries to be made prior to those unique to the individual statement.

CIT table

word 1					∝(IFN)				0
word 2			C		A		L		0
word 3	1	7							
word 4					0				0

CIT table

word 1					0				0
word 2			X		I		T		0
word 3			(		L		E	V	)
word 4					0				0

CLOSUB table

word 1			(		L		E	V	)

READ N, LIST

The following table entries are made:

CIT table

word 1			0		0
word 2	E	T	M		0
word 3					0
word 4			0		0

CIT table

word 1			0		0
word 2	C	A	L		0
word 3	(	D	B	C	)
word 4			0		0

CLOSUB table

word 1	(	D	B	C	)

CIT table

word 1			0		0
word 2	S	L	W		0
word 3					0
word 4			1		0

CIT table

word 1			0		0
word 2	C	A	L		0
word 3	(	C	S	H	)
word 4			0		0

CLOSUB table

word 1	(	C	S	H	)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FMTEFN table

word 1	1	N

   CIT table

word 1		0		248
word 2	N	T	R	81
word 3	1			N
word 4		0		0

2. Variable designation which is verified to be an array.

   The following table entry is made:

   CIT table

word 1		0		248
word 2	N	T	R	81
word 3	Variable Name (BCD)			
word 4		0		0

The LIST is now scanned and table entries made in the following manner:

The following table entry is made:

CIT table

word 1	∝(IFN)*	0
word 2	E T M	0
word 3		0
word 4	0	0

Each variable is collected. There are several possible cases:

1. Variable is not subscripted and is not the name of an array. The following table entry is made:

CIT table

word 1	∝(IFN)*	0
word 2	N T R	0
word 3	Variable Name	(BCD)
word 4	0	0

If the variable is fixed-point the following table entry is made:

FORVAL table

word 1	∝(IFN)i	
word 2	Variable Name	(BCD)

2. Variable is subscripted. There are two possible cases:

a) Subscript is constant. The following table entry is made:

CIT table

word 1	∝(IFN)*	0
word 2	N T R	0
word 3	Variable Name	(BCD)
word 4	K	0

where K is the resultant relative address.

b) Subscript has some variable part. The following table entry is made:

CIT table

word 1	∝(IFN)*	0
word 2	N T R	0
word 3	Variable Name	(BCD)
word 4	K	I-7

3. Variable is an array name. The dimension(s) of the array as found in the appropriate DIM table and multiplied to form the total size K of the array. There are two possible cases:

a) K≤1. Treat as a non-subscripted variable. See 1 above.

b) K>1. The following table entries are made:

FIXCON table (if not previously entered)

word 1	(K-1)

CIT table

word 1	∝(IFN)*	0
word 2	L X D	0
word 3	2	
word 4	i	8

where i is the position of (K-1) in the FIXCON table.

CIT table (only if last trapping mode instruction was LTM)

word 1	∝(IFN)*	0
word 2	E T M	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	N T R	0
word 3	Variable Name (BCD)	
word 4	0	8

CIT table

word 1	0	0
word 2	L T M	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	T I X	1
word 3	∝(IFN)*	
word 4	0	8

CIT table

word 1	0	0
word 2	E T M	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	D E D	0
word 3		0
word 4	0	8

CIT table

word 1	0	0
word 2	N T R	0
word 3	Variable Name (BCD)	
word 4	0	0

Upon completion of the LIST the following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	C A L	0
word 3	1 7	
word 4	0	0

CIT table

word 1	0	0
word 2	X I T	0
word 3	(R T N)	
word 4	0	0

CLOSUB table

word 1	(R T N)	

READ INPUT TAPE I, N, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary. It is placed in address of word 2 of pending CIT table entry whose operation is NTR.
2. Variable designation which is verified to be fixed-point. The following table entries are made:

FORVAR table

word 1	∝(IFN)
word 2	V a r i a b l e N a m e (BCD)

CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

CIT table

word 1	0	0
word 2	S T D	0
word 3	∝(IFN)1	
word 4	0	0

where ∝(IFN)1 is ∝(IFN) incremented by 1.

The following table entries are made:

CIT table

word 1	0	0
word 2	E T M	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	C A L	0
word 3	(D B C)	
word 4	0	0

CLOSUB table

word 1	(D B C)

CIT table

word 1	0	0
word 2	S L W	0
word 3		0
word 4	1	0

CIT table

word 1	0	0
word 2	C A L	0
word 3	(T S H)	
word 4	0	0

CLOSUB table

word 1	(T S H)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FMTEFN table

word 1	1	N

   CIT table

word 1		∝(IFN)1	248
word 2	N	T R	I
word 3	1		N
word 4		0	0

   where I is the unit designation or is 0 if unit designation is variable.

2. Variable designation which is verified to be an array.

   The following table entry is made:

   CIT table

word 1		∝(IFN)1	248
word 2	N	T R	I
word 3	V a r i a b l e	N a m e (BCD)	
word 4		0	0

   where I is the unit designation or is 0 if unit designation is variable.

The LIST is now scanned and table entries made in the same manner as for READ N, LIST.

PRINT N, LIST

The following table entries are made:

CIT table

word 1			0	0
word 2	E	T	M	0
word 3				0
word 4			0	0

CIT table

word 1			0		0
word 2	C	A	L		0
word 3	(	B	D	C	)
word 4			0		0

CLOSUB table

word 1	(	B	D	C	)

CIT table

word 1			0	0
word 2	S	L	W	0
word 3				0
word 4			1	0

CIT table

word 1			0	0
word 2	C	A	L	0
word 3	(	S	P	H)
word 4			0	0

CLOSUB table

word 1	(	S	P	H	)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FMTEFN table

word 1	1	N

   CIT table

word 1			0	248
word 2	N	T	R	0
word 3	1			N
word 4			0	0

2. Variable designation which is verified to be an array.

   The following table entry is made:

   CIT table

word 1			0	248
word 2	N	T	R	0
word 3	V a r i a b l e	N a m e	(BCD)	
word 4			0	0

The LIST is now scanned and table entries made in the same manner as for READ N, LIST with one exception. Fixed-point non-subscripted variables are entered in FORVAR rather than in FORVAL.

Upon completion of the LIST the following table entries are made:

CIT table

word 1			α(IFN)*				0
word 2		C	A	L			0
word 3	1	7					
word 4				0			0

CIT table

word 1				0			0
word 2		X	I	T			0
word 3		(	F	I	L	)	
word 4				0			0

CLOSUB table

word 1		(	F	I	L	)	

PUNCH N, LIST

The following table entries are made:

CIT table

word 1			0	0
word 2	E	T	M	0
word 3				0
word 4			0	0

CIT table

word 1			0		0
word 2	C	A	L		0
word 3	(	B	D	C	)
word 4			0		0

CLOSUB table

word 1	(	B	D	C	)

CIT table

word 1			0	0
word 2	S	L	W	0
word 3				0
word 4			1	0

CIT table

word 1			0		0
word 2	C	A	L		0
word 3	(	S	C	H	)
word 4			0		0

CLOSUB table

word 1	(	S	C	H	)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FMTEFN table

word 1	1	N

   CIT table

word 1			0	248
word 2	N	T	R	0
word 3	1			N
word 4			0	0

2. Variable designation which is verified to be an array.

   The following table entry is made:

   CIT table

word 1			0	248
word 2	N	T	R	0
word 3	V a r i a b l e	N a m e		(BCD)
word 4			0	0

The LIST is now scanned and table entries made in the same manner as for PRINT N, LIST.

WRITE OUTPUT TAPE I, N, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary. It is placed in address of word 2 of pending CIT table entry whose operation is NTR.
2. Variable designation which is verified to be fixed-point. The following table entries are made:

FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

CIT table

word 1	0	0
word 2	S T D	0
word 3	∝(IFN)1	
word 4	0	0

where ∝(IFN)1 is ∝(IFN) incremented by 1.

The following table entries are made:

CIT table

word 1	0	0
word 2	E T M	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	C A L	0
word 3	(B D C)	
word 4	0	0

CLOSUB table

word 1	(B D C)

CIT table

word 1	0	0
word 2	S L W	0
word 3		0
word 4	1	0

CIT table

word 1	0	0
word 2	C A L	0
word 3	(S T H)	
word 4	0	0

CLOSUB table

word 1	(S T H)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FMTEFN table

word 1	1	N

   CIT table

word 1		∝(IFN)1	248
word 2	N	T R	I
word 3	1		N
word 4		0	0

   where I is the unit designation or is 0 if unit designation is variable.

2. Variable designation which is verified to be an array. The following table entry is made:

   CIT table

word 1		∝(IFN)1	248
word 2	N	T R	I
word 3	V a r i a b l e	N a m e (BCD)	
word 4		0	0

   where I is the unit designation or is 0 if unit designation is variable.

The LIST is now scanned and table entries made in the same manner as for PRINT N, LIST.

READ TAPE N, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 220_8 to form the binary tape address.
   The following table entry is made:

   CIT table

word 1	∝(IFN)1	0
word 2	R T B	0
word 3		0
word 4	$(220_8 + N)$	0

   where ∝(IFN)1 is ∝(IFN) incremented by 1.

2. Variable designation which is verified to be fixed-point.
   The following table entries are made:

   FORVAR table

word 1	∝(IFN)1	
word 2	Variable Name (BCD)	

   CIT table

word 1	∝(IFN)1	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

   FIXCON table (if not previously entered)

word 1	220_8	

   CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

   where i is the position of 220_8 in the FIXCON table.

   CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

   CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)2	
word 4	0	0

   where ∝(IFN)2 is ∝(IFN)1 incremented by 1.

   CIT table

word 1	∝(IFN)2	0
word 2	R T B	0
word 3		0
word 4	0	0

The following table entries are made:

CIT table

word 1			0	0
word 2		C P Y		0
word 3	1			
word 4			0	0

CIT table

word 1			0	0
word 2		X I T		0
word 3	1	7		
word 4			3	0

CIT table

word 1			0	0
word 2		H P R		0
word 3				0
word 4			0	0

CIT table

word 1			0	0
word 2		X I T		0
word 3			∝(IFN)2	248
word 4				0

The LIST is now scanned and table entries made in the following manner:

Each variable is collected. There are several possible cases:

1. Variable is not subscripted and is not the name of an array. The following table entry is made.

CIT table

word 1			∝(IFN)*	0
word 2		C P Y		0
word 3	V a r i a b l e N a m e			(BCD)
word 4			0	0

If the variable is fixed-point the following table entry is made.

FORVAL table

word 1		∝(IFN) i	
word 2	V a r i a b l e N a m e		(BCD)

2. Variable is subscripted. There are two possible cases:

a) Subscript is constant. The following table entry is made:

CIT table

word 1			∝(IFN)*	0
word 2		C P Y		0
word 3	V a r i a b l e N a m e			(BCD)
word 4			K	0

where K is the resultant relative address.

b) Subscript has some variable part. The following table entry is made:

CIT table

word 1	∝(IFN)*	0
word 2	C P Y	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	K	I -

3. Variable is an array name. The dimension(s) of the array as found in the appropriate DIM table are multiplied to form the total size K of the array. There are two possible cases:

a) K=1. Treat as a non-subscripted variable. See 1 above.

b) K>1. The following table entries are made:

FIXCON table

word 1	(K-1)

CIT table

word 1	∝(IFN)*	0
word 2	L X D	0
word 3	2	
word 4	i	8

where i is the position of (K-1) in the FIXCON table.

CIT table

word 1	∝(IFN)i	0
word 2	C P Y	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	8

CIT table

word 1	0	0
word 2	T I X	1
word 3	∝(IFN)i	
word 4	0	8

CIT table

word 1	0	0
word 2	D E D	0
word 3		0
word 4	0	8

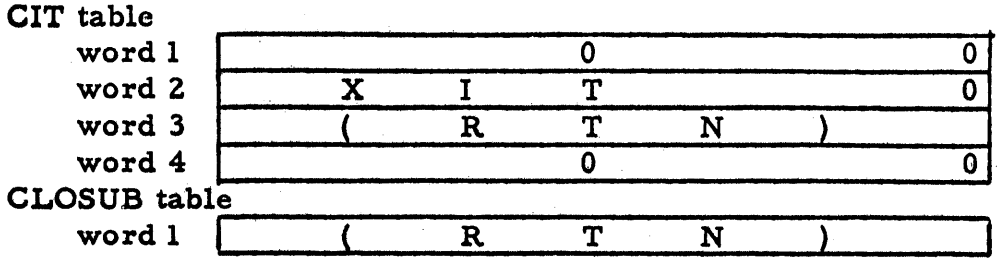
CIT table

word 1	0	0
word 2	C P Y	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

Upon completion of the LIST the following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	C A L	0
word 3	1 7	
word 4	0	0


CIT table
word 1
0
0
word 2
X
I
T
0
word 3
(
R
T
N
)
word 4
0
0
CLOSUB table
word 1
(
R
T
N
)

WRITE TAPE N, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 220_8 to form the binary tape address.
   The following table entry is made:
   CIT table

word 1	∝(IFN)1	0
word 2	W T B	0
word 3		0
word 4	(220_8+N)	0

2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)1
word 2	V a r i a b l e N a m e (BCD)

   CIT table

word 1	∝(IFN)1	0
word 2	C A L	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

   FIXCON table (if not previously entered)

word 1	220_8

   CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

   where i is the position of 220_8 in the FIXCON table.
   CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

   CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)2	
word 4		0

   where ∝(IFN)2 is ∝(IFN)1 incremented by 1.
   CIT table

word 1	∝(IFN)2	0
word 2	W T B	0
word 3		0
word 4	0	0

The following table entry is made:

CIT table

word 1			0		0
word 2	C	P	Y		0
word 3	6				
word 4			2		0

The LIST is now scanned and table entries made in the same manner as for READ TAPE N, LIST with one exception. Fixed-point non-subscripted variables are entered in FORVAR rather than in FORVAL.

READ DRUM N, J, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 300_8 to form the binary drum address.
   The following table entry is made:
   CIT table

word 1	∝(IFN)1	0
word 2	R D R	0
word 3		0
word 4	(300_8+N)	0

   where ∝(IFN)1 is ∝(IFN) incremented by 1.
2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)1
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)1	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

   FIXCON table (if not previously entered)

word 1	300_8

   CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

   where i is the position of 300_8 in the FIXCON table.
   CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

   CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)2	
word 4	0	0

   where ∝(IFN)2 is ∝(IFN)1 incremented by 1.
   CIT table

word 1	∝(IFN)2	0
word 2	R D R	0
word 3		0
word 4	0	0

The drum address is collected. There are two possible cases:

1. Constant address which is converted to binary.

The following table entries are made:

CIT table

word 1	∝(IFN)3	370_8
word 2	P X D	0
word 3		0
word 4	J	0

where ∝(IFN)3 is ∝(IFN)2 incremented by 1.

CIT table

word 1		370_8
word 2	L D A	0
word 3	∝(IFN)3	
word 4	0	0

2. Variable address which is verified to be fixed-point.

The following table entries are made:

FORVAR table

word 1	∝(IFN)3	
word 2	Variable Name (BCD)	

CIT table

word 1	∝(IFN)3	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)4	
word 4	0	0

where ∝(IFN)4 is ∝(IFN)3 incremented by 1.

CIT table

word 1	∝(IFN)4	0
word 2	P X D	0
word 3		0
word 4	0	0

CIT table

word 1	0	370_8
word 2	L D A	0
word 3	∝(IFN)4	
word 4	0	0

The LIST is now scanned and table entries made in the same manner as for READ TAPE N, LIST with one exception. Subscripted variables where the subscript contains some variable part are not permitted.

WRITE DRUM N, J, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 300_8 to form the binary drum address.
   The following table entry is made:
   CIT table

word 1	∝(IFN)1	0
word 2	W D R	0
word 3		0
word 4	$(300_8 + N)$	0

   where ∝(IFN)1 is ∝(IFN) incremented by 1.
2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)1	
word 2	V a r i a b l e N a m e (BCD)	

   CIT table

word 1	∝(IFN)1	0
word 2	C A L	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

   FIXCON table (if not previously entered)

word 1	300_8	

   CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

   where i is the position of 300_8 in FIXCON table.
   CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

   CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)2	
word 4	0	0

   where ∝(IFN)2 is ∝(IFN)1 incremented by 1.
   CIT table

word 1	∝(IFN)2	0
word 2	W D R	0
word 3		0
word 4	0	0

The drum address is collected. There are two possible cases:

1. Constant address which is converted to binary.

   The following table entries are made:

   CIT table

word 1		∝(IFN)3	370_8
word 2	P	X D	0
word 3			0
word 4		J	0

   where ∝(IFN)3 is ∝(IFN)2 incremented by 1.

   CIT table

word 1		0	370_8
word 2	L	D A	0
word 3		∝(IFN)3	
word 4		0	0

2. Variable address which is verified to be fixed-point.

   The following table entries are made:

   FORVAR table

word 1	∝(IFN)3
word 2	Variable Name (BCD)

   CIT table

word 1		∝(IFN)3	0
word 2	C	A L	0
word 3	Variable Name (BCD)		
word 4		0	0

   CIT table

word 1		0	0
word 2	A	R S	0
word 3			0
word 4		18	0

   CIT table

word 1		0	0
word 2	S	T A	0
word 3		∝(IFN)4	
word 4		0	0

   where ∝(IFN)4 is ∝(IFN)3 incremented by 1.

   CIT table

word 1		∝(IFN)4	0
word 2	P	X D	0
word 3			0
word 4		0	0

   CIT table

word 1		0	370_8
word 2	L	D A	0
word 3		∝(IFN)4	
word 4		0	0

The LIST is now scanned and table entries made in the same manner as for READ DRUM N, J, LIST with one exception. Fixed-point non-subscripted variables are entered in FORVAR rather than in FORVAL.

END FILE I

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 220_8 to form the binary tape address.
   The following table entry is made:
   CIT table

word 1	∝(IFN)	0
word 2	W E F	0
word 3		0
word 4	(220_8+I)	0

2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

   FIXCON table (if not previously entered)

word 1 220_8

   CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

   where i is the position of 220_8 in the FIXCON table.
   CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

   CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)1	
word 4	0	0

   where ∝(IFN)1 is ∝(IFN) incremented by 1.
   CIT table

word 1	∝(IFN)1	0
word 2	W E F	0
word 3		0
word 4	0	0

REWIND I

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 220_8 to form binary tape address.
   The following table entry is made:

CIT table

word 1	∝(IFN)	0
word 2	R E W	0
word 3		0
word 4	(220_8+I)	0

2. Variable designation which is verified to be fixed-point.
   The following table entries are made:

FORVAL table

word 1	∝(IFN)
word 2	Variable Name (BCD)

CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

FIXCON table (if not previously entered)

word 1	220_8

CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

where i is the position of 220_8 in the FIXCON table.

CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)1	
word 4	0	0

where ∝(IFN)1 is ∝(IFN) incremented by 1.

CIT table

word 1	∝(IFN)1	0
word 2	R E W	0
word 3		0
word 4	0	0

BACKSPACE I

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary and added to 220_8 to form binary tape address.
   The following table entry is made:
   CIT table

word 1	∝(IFN)	0
word 2	B S T	0
word 3		0
word 4	(220_8+I)	0

2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

   FIXCON table (if not previously entered)

word 1	220_8

   CIT table

word 1	0	0
word 2	A D D	0
word 3	2	
word 4	i	0

   where i is the position of 220_8 in the FIXCON table.
   CIT table

word 1	0	0
word 2	A R S	0
word 3		0
word 4	18	0

   CIT table

word 1	0	0
word 2	S T A	0
word 3	∝(IFN)1	
word 4	0	0

   where ∝(IFN)1 is ∝(IFN) incremented by 1.
   CIT table

word 1	∝(IFN)1	0
word 2	B S T	0
word 3		0
word 4	0	0

I-O LIST DO specification

The data transmission statements except READ DRUM and WRITE DRUM may have a LIST DO specification. This is information that specifies the range and initial, increment, and final values of the controlling variable.

The appearance of "(", not a part of a subscript, is assumed to be the beginning of a LIST DO specification. The variable and parameters are explicitily written and the termination of the LIST DO is at the point of their appearance:

Example:

. . . (V, . . . , Vi, I = N1, N2, N 3), . . .

The processing of LIST DOs is a simultaneous procedure with the processing of LIST variables. Since LIST DOs reading from left to right are nested and the first "(" and the last "I = N1, N2, N3)" are paired, the processing is disjoint.

As each "(" is encountered it is assumed to be the beginning of a LIST DO. The following table entry is made:

Temporary LIST DO table

word 1	- ∝(IFN)i	248
word 2		0
word 3		0
word 4		0
word 5		0

The ∝(IFN) is incremented. A counter associated with the table is incremented.

As each "I = N1, N2, N3)" is reached it is scanned and the incompleted entry in the table associated with the matching "(" is completed as follows:

Temporary LIST DO table

word 1	- ∝(IFN)i	B(IFN)
word 2	V a r i a b l e N a m e	(BCD)
word 3		N1
word 4		N2
word 5		N3

where B(IFN) is the current value of (IFN) counter.

N1, N2, N3 may each be variable or constant, if variable bits 20, 19, 18 respectively of word 1 are set equal to 1.

The table counter is decremented.

The appearance of ")" not part of a subscript nor of a LIST DO specification causes the table counter to be decremented and effectively nullifies the entry in the temporary LIST DO table associated with the matching "(".

When the LIST is completed, all significant (completed) entries in the temporary LIST DO table are transferred to the TDO table. Any entries not completed are considered null and are ignored.

Certain table entries result from the appearance of "(" and ")" not surrounding a subscript.

1. "(" appearing at the beginning of a nest.
   The following table entry is made:
   CIT table

word 1		∝(IFN)i		0
word 2	B	S	S	0
word 3				0
word 4		0		0

2. "(" appearing within a nest.
   The following table entry is made:
   CIT table

word 1		∝(IFN)i		0
word 2	L	T	M	0
word 3				0
word 4		0		0

3. ")" same as 2 above

DIAGNOSTIC

A diagnostic program exists for source program errors found or machine errors occuring during Section One processing. This program consists of:

Program to prepare message
Print program
Table of comments

When an error is found or occurs during Section One control goes to the Diagnostic Program by means of TSX using IR4.
There are several possible cases:

I. IR4 $\neq$ 0 signifies an error call.
   1) First error: Print "DIAGNOSTIC PROGRAM" heading and procede as in 2) below.
   2) Not first error: Construct parameters for printing statement being processed and comment describing error. Restore any modified statements to their original form and print statement and comment.
      Return control to Section One for next statement.

II. IR4 = 0 signifies completion of Section One.
   1) No errors had occurred. Go to Section One Prime.
   2) Some errors had occurred. Print "END OF DIAGNOSTIC" message.
      a) If any error was source program go to Source Program Error supervisor program.
      b) If all errors were machine errors go to Machine Error supervisor program.

FORTRAN II, Section One (709 Version)

The 709 translator for FORTRAN II is similar to the 704 version. Those processors and features which are the same will not be restated. The reader is referred to the 704 version for the following:

Classification routines

Processors for

DIMENSION
EQUIVALENCE
COMMON
FREQUENCY

FORMAT
GO TO (. . .), I
GO TO N, (. . .)
DO N I = N1, N2, N3
IF (. . .) N1, N2, N3
SENSE LIGHT I
IF (SENSE LIGHT I) N1, N2
IF (SENSE SWITCH I) N1, N2
IF DIVIDE CHECK N1, N2
ASSIGN I TO N
CONTINUE
PAUSE N
STOP N
CALL NAME (. . .)
RETURN
ARITHMETIC (Scan, Level Analysis, Optimization)
SUBSCRIPTS
ERROR FLOW TRACE

Those processors which differ from the 704 version are described below:

SUBROUTINE NAME (ARG1, . . . , ARGN)

The assumed CIT and CLOSUB table entries for Floating-Point Trap initiatization in a main program are nullified. The statement is then processed in the same manner as 704 version.

FUNCTION NAME (ARG1, . . . , ARGN)

The assumed CIT and CLOSUB table entries for Floating-Point Trap initiatization in a main program are nullified. The statement is then processed in the same manner as 704 version.

GO TO N

The branch address N is collected and converted to binary. The following table entry is made:

TIFGO table

word 1	∝(IFN)	0
word 2	0	N(EFN)

IF ACCUMULATOR OVERFLOW N1, N2

The branch addresses N1, N2 are collected and converted to binary. The following table entries are made:

CIT table

word 1		∝(IFN)		0
word 2		C A L		0
word 3	4			
word 4	-	315		0

CIT table

word 1		0		0
word 2		S T Z		0
word 3	4			
word 4	-	315		0

CIT table

word 1		∝(IFN)1		0
word 2		T N Z		0
word 3				0
word 4		0		0

where ∝(IFN)1 is ∝(IFN) incremented by 1.

TIFGO table

word 1	∝(IFN)1	5
word 2	N1(EFN)	N2(EFN)

IF QUOTIENT OVERFLOW N1, N2

This statement is processed in same manner as IF ACCUMULATOR OVERFLOW N1, N2.

READ N, LIST

The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(C S H)	
word 4	0	4

CLOSUB table

word 1	(C S H)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:

FMTEFN table

word 1	1	N

CIT table

word 1	0	0
word 2	P Z E	0
word 3	1	N
word 4	0	0

2. Variable designation which is verified to be an array.
   The following table entry is made:

CIT table

word 1	0	0
word 2	P Z E	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

The following table entry is made:

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

The LIST is now scanned and table entries made in the following manner:

Each variable in the LIST is collected. There are several possible cases:

1. Variable is not subscripted and is not the name of an array. The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	S T R	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	S T Q	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

If the variable is fixed-point the following table entry is made:

FORVAL table

word 1	∝(IFN)i	0
word 2	V a r i a b l e N a m e (BCD)	

2. Variable is subscripted. There are two possible cases:

a) Subscript is constant.

The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	S T R	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	S T Q	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	L	0

where L is the resultant addend.

b) Subscript has some variable part.

The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	S T R	0
word 3		0
word 4	0	0

CIT table

word 1	0	0
word 2	S T Q	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	L	I-7

3. Variable is an array name. The dimension(s) of the array as found in the appropriate DIM table are multiplied to form the total size K of the array. There are two possible cases:

a) K = 1. Treat as a non-subscripted variable. See 1 above.
b) K > 1. The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(S L I)	
word 4	0	4

CLOSUB table

word 1	(S L I)

CIT table

word 1	0	0
word 2	P Z E	0
word 3	Variable Name (BCD)	
word 4	1	0

CIT table

word 1	0	0
word 2	P Z E	0
word 3	0	0
word 4	K	0

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

Upon completion of the LIST the following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(R T N)	
word 4	0	4

CLOSUB table

word 1	(R T N)

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

READ INPUT TAPE I, N, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of I in the FIXCON table.

2. Variable designation which is verified to be fixed-point.

   The following table entries are made:

   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entries are made:

   CIT table

word 1	0	0
word 2	S X D	0
word 3	6	
word 4	4	4

   CIT table

word 1	0	0
word 2	T S X	0
word 3	(T S H)	
word 4	0	4

   CLOSUB table

word 1	(T S H)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FMTEFN table

word 1	1	N

   CIT table

word 1	0	0
word 2	P Z E	0
word 3	1	N
word 4	0	0

2. Variable designation which is verified to be an array.
The following table entry is made:

CIT table

word 1	0	0
word 2	P Z E	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entry is made:

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

The LIST is now scanned and table entries made in the same manner as for READ N, LIST.

PRINT N, LIST

The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(S P H)	
word 4		4

CLOSUB table

word 1	(S P H)	

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:

FMTEFN table

word 1	1	N

CIT table

word 1	0	0
word 2	P Z E	0
word 3	1	N
word 4	0	0

2. Variable designation which is verified to be an array.
   The following table entry is made:

CIT table

word 1	0	0
word 2	P Z E	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entry is made:

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

The LIST is now scanned and table entries made in the following manner:

Each variable in the LIST is collected. There are several possible cases:

1. Variable is not subscripted and is not the name of an array. The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	L D Q	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

CIT table

word 1	0	0
word 2	S T R	0
word 3		0
word 4	0	0

If the variable is fixed-point the following table entry is made:

FORVAR table

word 1	∝(IFN)i	0
word 2	V a r i a b l e N a m e (BCD)	

2. Variable is subscripted. There are two possible cases.

a) Subscript is constant.

The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	L D Q	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	L	0

where L is the resultant addend.

CIT table

word 1	0	0
word 2	S T R	0
word 3		0
word 4	0	0

b) Subscript has some variable part.

The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	L D Q	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	L	I-7

CIT table

word 1	0	0
word 2	S T R	0
word 3		0
word 4	0	0

3. Variable is an array name. The dimension(s) of the array as found in the appropriate DIM table are multiplied to form the total size K of the array. There are two possible cases:

a) K = 1. Treat as a non-subscripted variable. See 1 above.
b) K > 1. The following table entries are made:

CIT table

word 1	∝(IFN)*	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(S L O)	
word 4	0	4

CLOSUB table

word 1	(S L O)

CIT table

word 1	0	0
word 2	P Z E	0
word 3	Variable Name (BCD)	
word 4	1	0

CIT table

word 1	0	0
word 2	P Z E	0
word 3	0	0
word 4	K	0

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

Upon completion of the LIST the following table entries are made:

CIT table

word 1	∝ (IFN)*	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(F I L)	
word 4	0	0

CLOSUB table

word 1	(F I L)

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

PUNCH N, LIST

The following table entries are made:

CIT table

word 1	∝(IFN)	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(S C H)	
word 4	0	4

CLOSUB table

word 1	(S C H)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:

FMTEFN table

word 1	1	N

CIT table

word 1	0	0
word 2	P Z E	0
word 3	1	N
word 4	0	0

2. Variable designation which is verified to be an array.
   The following table entry is made:

CIT table

word 1	0	0
word 2	P Z E	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entry is made:

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

The LIST is now scanned and table entries made in the same manner as for PRINT N, LIST.

WRITE OUTPUT TAPE I, N, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:
   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of I in the FIXCON table.
2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entries are made:
CIT table

word 1	0	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(S T H)	
word 4	0	4

CLOSUB table

word 1	(S T H)

The format designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:
   FMTEFN table

word 1	1	N

   CIT table

word 1	0	0
word 2	P Z E	0
word 3	1	N
word 4	0	0

2. Variable designations which is verified to be an array.
The following table entry is made:

CIT table

word 1	0	0
word 2	P Z E	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entry is made:

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

The LIST is now scanned and table entries made in the same manner as for PRINT N, LIST.

READ TAPE I, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1		∝(IFN)	0
word 2	C	A L	0
word 3	2		
word 4		i	0

   where i is the position of I in the FIXCON table.

2. Variable designation which is verified to be fixed-point.

   The following table entries are made:

   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1		∝(IFN)	0
word 2	C	A L	0
word 3	Variable Name (BCD)		
word 4		0	0

The following table entries are made:

   CIT table

word 1		0	0
word 2	S	X D	0
word 3	6		
word 4		4	4

   CIT table

word 1		0	0
word 2	T	S X	0
word 3	(	T S B)	
word 4		0	4

   CLOSUB table

word 1	(T S B)

   CIT table

word 1		0	0
word 2	L	X D	0
word 3	6		
word 4		4	4

The LIST is now scanned and table entries made in the same manner as for READ N, LIST. Upon completion of the LIST the following table entries are made:

CIT table

word 1			∝(IFN)*		0
word 2		S	X	D	0
word 3	6				
word 4			4		4

CIT table

word 1			0		0
word 2		T	S	X	0
word 3		(	R	L	R)
word 4			0		4

CLOSUB table

word 1		(	R	L	R)

CIT table

word 1			0		0
word 2		S	X	D	0
word 3	6				
word 4			4		4

WRITE TAPE I , LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:
   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1		∝(IFN)		0
word 2		C A L		0
word 3	2			
word 4		i		0

   where i is the position of I in the FIXCON table.
2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)
word 2	V a r i a b l e N a m e (BCD)

   CIT table

word 1		∝(IFN)		0
word 2		C A L		0
word 3	V a r i a b l e N a m e (BCD)			
word 4		0		0

The following table entries are made:
CIT table

word 1		0		0
word 2		S X D		0
word 3	6			
word 4		4		4

CIT table

word 1		0		0
word 2		T S X		0
word 3		(S T B)		
word 4		0		4

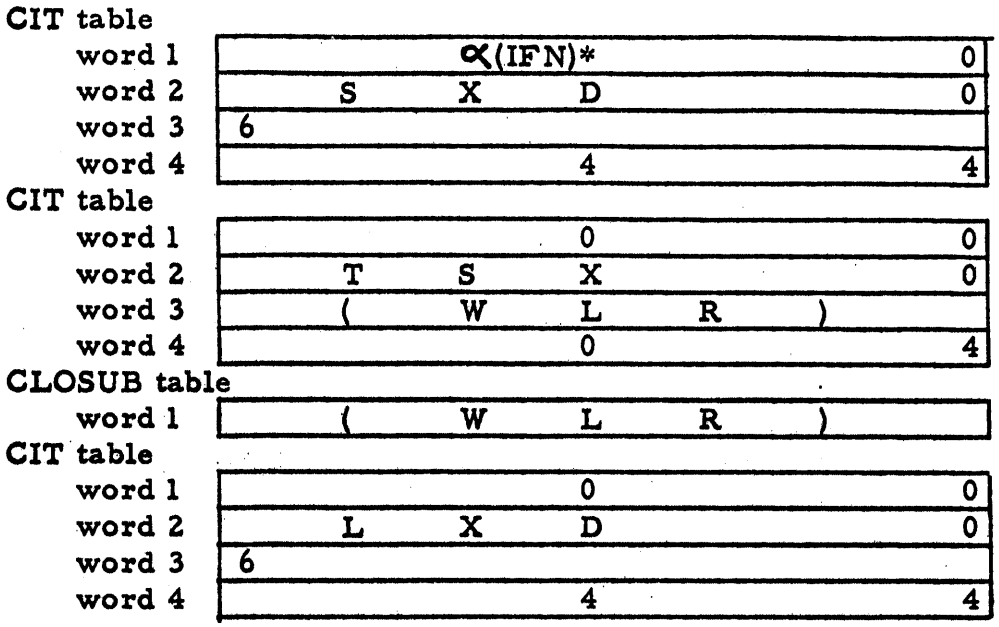
CLOSUB table

word 1	(S T B)

CIT table

word 1		0		0
word 2		L X D		0
word 3	6			
word 4		4		4

The LIST is now scanned and table entries made in the same manner as for PRINT N, LIST. Upon completion of the LIST the following table entries are made:


CIT table
word 1
∝(IFN)*
0
word 2
S
X
D
0
word 3
6
word 4
4
4
CIT table
word 1
0
0
word 2
T
S
X
0
word 3
(
W
L
R
)
word 4
0
4
CLOSUB table
word 1
(
W
L
R
)
CIT table
word 1
0
0
word 2
L
X
D
0
word 3
6
word 4
4
4

READ DRUM I , J, LIST

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:
   FIXCON table

word 1	I

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of I in the FIXCON table.
2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)
word 2	V a r i a b l e N a m e (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	V a r i a b l e N a m e (BCD)	
word 4	0	0

The following table entries are made:

CIT table

word 1	0	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(D R S)	
word 4	0	4

CLOSUB table

word 1	(D R S)

The drum address is collected. There are two possible cases:

1. Constant address which is converted to binary.
   The following table entries are made:
   FIXCON table (if not previously entered)

word 1	N

CIT table

word 1			0	0
word 2		C A L		0
word 3	2			
word 4			i	0

where i is the position of N in the FIXCON table.

2. Variable address which is verified to be fixed-point.

The following table entries are made:

FORVAL table

word 1	∝(IFN)
word 2	Variable Name (BCD)

CIT table

word 1			0	0
word 2		C A L		0
word 3	Variable Name (BCD)			
word 4			0	0

The following table entry is made:

CIT table

word 1			0	0
word 2		L D A		0
word 3				0
word 4			0	0

The LIST is now scanned and table entries made in the following manner:

Each variable in the LIST is collected. There are several possible cases:

1. Variable is not subscripted and is not the name of an array.

The following table entry is made:

CIT table

word 1			0	0
word 2		C P Y		0
word 3	Variable Name (BCD)			
word 4			0	0

If the variable is fixed-point the following table entry is made:

FORVAL table

word 1	∝(IFN)1	0
word 2	Variable Name (BCD)	

2. Variable is subscripted. There are two possible cases:

a) Subscript is constant

The following table entry is made:

CIT table

word 1			0	0
word 2		C P Y		0
word 3	Variable Name (BCD)			
word 4			L	0

where L is the resultant addend.

b) Subscript has some variable part.

Not permitted.

3. Variable is an array name. The dimension(s) of the array as found in the appropriate DIM table are multiplied to form the total size K of the array. There are two possible cases:
   a) K = 1. Treat as a non-subscripted variable. See 1 above.
   b) K > 1. The following table entries are made:

FIXCON table

word 1	(K-1)

CIT table

word 1			0	0
word 2		L X	D	0
word 3	2			
word 4			i	8

where i is the position of (K-1) in the FIXCON table.

CIT table

word 1			0	0
word 2		C P	Y	0
word 3	V a r i a b l e	N a m e		(BCD)
word 4			0	8

CIT table

word 1			0	0
word 2		T I	X	1
word 3	1 7			
word 4	-		1	8

CIT table

word 1			0	0
word 2		D E	D	0
word 3				0
word 4			0	8

CIT table

word 1			0	0
word 2		C P	Y	
word 3	V a r i a b l e	N a m e		(BCD)
word 4			0	0

Upon completion of the LIST the following table entry is made:

CIT table

word 1			0	0
word 2		L X	D	0
word 3	6			
word 4			4	4

WRITE DRUM I , J, LIST

The unit designation is collected. There are two possible cases.

1. Constant designation which is converted to binary.

   The following table entries are made:

   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of I in the FIXCON table.

2. Variable designation which is verified to be fixed-point.

   The following table entries are made:

   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entries are made:

CIT table

word 1	0	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(S D R)	
word 4	0	4

CLOSUB table

word 1	(S D R)

The drum address is collected. There are two possible cases:

1. Constant address which is converted to binary.

   The following table entries are made:

   FIXCON table (if not previously entered)

word 1	N

   CIT table

word 1	0	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of N in the FIXCON table.

2. Variable address which is verified to be fixed-point.
   The following table entries are made:

FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

CIT table

word 1	0	0
word 2	C A L	
word 3	Variable Name (BCD)	
word 4	0	0

The following table entry is made:

CIT table

word 1	0	0
word 2	L D A	0
word 3		0
word 4	0	0

The LIST is now scanned and table entries made in the same manner as for READ DRUM N, J, LIST with one exception. Fixed-point non-subscripted variables are entered in FORVAR rather than in FORVAL.

END FILE I

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

   The following table entries are made:

   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of I in the FIXCON table.

2. Variable designation which is verified to be fixed-point.

   The following table entries are made:

   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entries are made:

   CIT table

word 1	0	0
word 2	S X D	0
word 3	6	
word 4	4	4

   CIT table

word 1	0	0
word 2	T S X	0
word 3	(E F T)	
word 4	0	4

   CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

   CLOSUB table

word 1	(E F T)

REWIND I

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.

The following table entries are made:

FIXCON table (if not previously entered)

word 1	I

CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

where i is the position of I in the FIXCON table.

2. Variable designation which is verified to be fixed-point.

The following table entries are made:

FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4	0	0

The following table entries are made:

CIT table

word 1	0	0
word 2	S X D	0
word 3	6	0
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(R W T)	
word 4	0	4

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

CLOSUB table

word 1	(R W T)

BACKSPACE I

The unit designation is collected. There are two possible cases:

1. Constant designation which is converted to binary.
   The following table entries are made:
   FIXCON table (if not previously entered)

word 1	I

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	2	
word 4	i	0

   where i is the position of I in the FIXCON table.
2. Variable designation which is verified to be fixed-point.
   The following table entries are made:
   FORVAR table

word 1	∝(IFN)
word 2	Variable Name (BCD)

   CIT table

word 1	∝(IFN)	0
word 2	C A L	0
word 3	Variable Name (BCD)	
word 4		0

The following table entries are made:
CIT table

word 1	0	0
word 2	S X D	0
word 3	6	
word 4	4	4

CIT table

word 1	0	0
word 2	T S X	0
word 3	(B S T)	
word 4	0	4

CIT table

word 1	0	0
word 2	L X D	0
word 3	6	
word 4	4	4

CLOSUB table

word 1	(B S T)

FORTRAN II, Section One (709 Version)

DIAGNOSTIC

A diagnostic program exists for source program errors found or machine errors occurring during Section One. The program consists of:

Program to prepare message
Print program
Table of comments

When an error is found or occurs during Section One control goes to the Diagnostic Program by means of a TSX using IR4.
There are several possible cases:

I. IR4 $\neq$ 0 signifies an error call.
   1) First error: Print "DIAGNOSTIC PROGRAM" heading and procede as in 2) below.
   2) Not first error: Construct parameters for printing statement being processed and comment describing error. Restore any modified statements to their original form and print statement and comment.
      a) If error was source program, return control to Section One for next statement.
      b) If error was machine, print "END OF DIAGNOSTIC" message and go to Machine Error Supervisor program.

II. IR4 = 0 signifies completion of Section One.
   1) No errors had occurred. Go to Section One Prime.
   2) Some source program errors had occurred. Write all diagnostic information which has been printed on tape B2 following source program. Go to Source Program Error supervisor program.

III

SECTION ONE-PRIME (704 Version)

This section is a terminal processor for Section One. It combines fragments of those tables which Section One generated as labeled buffer sized records. It makes certain modifications, primarily the replacement of EFNs with corresponding IFNs, which can only be accomplished when the entire source program has been reduced to tabular form.

The input to Section One-Prime consists of:

1. Various parameters describing tables (in cores)
2. Buffers containing terminal entries in tables (in cores)
3. Tables which Section One required for reference (FORSUB,END in cores. DIMI, DIM2, DIM3, TAU1, TAU2, TAU3, FIXCON, FLOCON on drums.)
4. Tables which Section One did not require for reference. (COMPAIL on tape 3; TEIFNO, TDO, TIFGO, TRAD, FORTAG, FORVAR, FORVAL, FRET, EQUIT, CLOSUB, FORMAT, SUBDEF, COMMON, HOLARG, NONEXC, TSTOPS, CALLFN, FMTEFN, TSKIPS on tape 4.)

The output of Section One-Prime consists of:

1. Tables on drums: TAU1, TAU2, TAU3, FIXCON, FLOCON, FORVAL
2. Tables on tape:

Tape 2: File 1 is Source Program
File 2 is COMPAIL table
File 3 is Compail Record Count - FORSUB
File 4, Record 1 is FLOCON table.
Record 2 is FORMAT table.
Record 3 is SIZ table.
File 5, Record 1 is END table.
Record 2 is SUBDEF table.
Record 3 is COMMON table.
Record 4 is HOLARG table.
Record 5 is TEIFNO table.
Record 6 is TIFGO table.
Record 7 is TRAD table.
Record 8 is TDO table.
Record 9 is FORVAL table.
Record 10 is FORVAR table.
Record 11 is FORTAG table.
Record 12 is FRET table.
Record 13 is EQUIT Table.
Record 14 is CLOSUB table.

Tape 3, File 1, Record 1 is FORMAT error list.
Record 2 is NONEXC table.
Record 3 is TSTOPS table.
Record 4 is TSKIPS table.

The tables are processed in the following order and manner:

COMMON - The table of common variables is assembled from tape 4 and the Section One buffer. It is written as record 3 of file 5 on tape 2; preceded by its identification (12) and word count.

HOLARG - The table of hollerith arguments is assembled from tape 4 and the Section One buffer. It is written as record 4 of file 5 on tape 2; preceded by its identification (13) and word count.

FIXCON - The entry count of the table of fixed point constants is doubled to form the word count and written with the table on drum 2.

COMPAIL - The last buffer of entries in the table of compiled instructions is written on tape 3 followed by an EOF. This tape is then rewound and the COMPAIL table transferred to tape 2 as the second file. A record count for COMPAIL is formed and written as record 1 of file 3 on tape 2.

FORSUB - The table of names and degrees of arithmetic statement functions, if any, is written after the compail record count in record 1 of file 3 on tape 2.

FLOCON - The table of floating point constants is read from drum 2. The block check sums used by Section One are deleted. The word count and the table are written as record 1 of file 4 on tape 2.

FORMAT - The table of format statements is assembled from tape 4 and the Section One buffer. It is written as record 2 of file 4 on tape 2; preceded by its identification (10) and word count.

FMTEFN- The table of references to fixed format statements is assembled from tape 4 and the Section One buffer. Each reference to a format is checked against the FORMAT table. If any referenced statements are missing an error list is developed for Section One Double Prime. This list, or a single flag word if no errors, is written as record 1 of file 1 on tape 3.

DIM1 - The table of one dimensional arrays is read from drum 3. The check sums for each entry are deleted. This table is now renamed SIZ.

DIM2 - The table of two dimensional arrays is read from drum 3. The check sums for each entry are deleted. The two dimensions are multiplied to form the size of the array. This table is added to SIZ.

DIM3 - The table of three-dimensional arrays is read from drum 3. The check sums for each entry are deleted. The three dimensions are multiplied to form the size of the array. This table is added to SIZ.

SIZ - A check sum of the entire SIZ table is computed. The table is written as record 3 of file 4 on tape 2. It is preceded by EIFNØ and its word count and is followed by the check sum.

END - The five word END table is written as record 1 of file 5 on tape 2.

SUBDEF- The table of subprogram definition is assembled from tape 4 and the Section One buffer. It is written as record 2 of file 5 on tape 2; preceded by its identification (11) and word count.

TEIFNO - The table of corresponding external and internal formula numbers is assembled from tape 4 and the Section One buffer. It is searched for duplicate external formula numbers. If such are found they are flagged as errors for Section One Double Prime. Those cases where Section One assigned more than one internal number are not considered as duplicates and the flag is deleted. The table is written as record 5 of file 5 on tape 2 preceded by its identification (0) and word count. It is retained in memory for use in processing tables discussed below.

TIFGO - The table of IFs, GO TOs and ASSIGNs is assembled from tape 4 and the Section One buffer. Each external formula number is searched for in TEIFNO and its corresponding internal number replaces it in TIFGO. Any external formula numbers not found are set equal to 0 as an error signal to Section One Double Prime. When all entries have been modified the table is written as record 6 of file 5 on tape 2 preceded by its identification (2) and word count.

TRAD - The table of COMPUTED and ASSIGNED GO TO addresses is assembled from tape 4 and the Section One buffer. Each entry, which is an external formula number, is searched for in TEIFNO. When found it is replaced by the corresponding internal formula number. If not found it is set equal to 0 as an error signal to Section One Double Prime. When all entries have been treated the table is written as record 7 of file 5 on tape 2 preceded by its identification (3) and word count.

TDO - The table of DOs is assembled from tape 4 and the Section One buffer. Each entry is examined to determine if it originated from a DO or from an Input-Output List. If it originated from an I/O List the flag that so indicated is deleted. If it originated from a DO the EFN for the end of the DO is searched for in TEIFNO. When it is found the corresponding IFN replaces it in TDO. If not found it is set equal to 0 as an error signal to Section One Double Prime. In those cases where Section One assigned more than one IFN to an external number, the last such IFN is used so that the DO includes all instructions of the terminal statement. When all entries have been treated the table is written as record 8 of file 5 on tape 2 preceded by its identification (1) and word count.

FORVAL - The table of definitions of fixed point variables is assembled from tape 4 and the Section One buffer.

CALLNM - The table of first and last internal formula numbers of statements containing references to subprograms is assembled from tape 4 and the Section One buffer.

Each IFN in FORVAL is searched for as a first IFN in CALLNM. If found it is replaced by the corresponding last IFN. When all entries have been processed the FORVAL table is written as record 9 of file 5 on tape 2; preceded by its identification (6) and word count. The CALLNM table is dead. Check sums are now formed for each FORVAL entry. Each entry, followed by its check sum, is written on drum 2.

FORVAR - The table of usages of fixed point variables is assembled from tape 4 and the Section One buffer. It is written as record 10 of file 5 on tape 2 preceded by its identification (5) and word count.

FORTAG - The table of tag usages is assembled from tape 4 and the Section One buffer. It is written as record 11 of file 5 on tape 2 preceded by its identification (4) and word count.

FRET - The table of frequency statements is assembled from tape 4 and the Section One buffer. Each EFN in FRET is searched for in TEIFNO. When found it is replaced with the corresponding IFN. If not found, it is set equal to 0 as an error signal for Section One Double Prime. The FRET table is now sorted by IFN to form an ordered list.

The TIFGO table is now re-examined for any entries for COMPUTED GO TO statements. The IFN of each such statement is searched for in FRET. If found, the list of branch frequencies is reversed to correspond to the object program transfer vector. When all TIFGO entries have been examined, the FRET table is written as record 12 of file 5 on tape 2 preceded by its identification (7) and word count.

EQUIT - The table of equivalence statements is assembled from tape 4 and the Section One buffer. The table is reformatized to make those variables which are equated into strings of relativelized symbols. Any found to be inconsistant are flagged as errors for Section One Double Prime. Any redundancies are deleted. The table is then written as record 13 of file 5 on tape 2 preceded by its identification (8) and word count.

CLOSUB - The table of names of closed (library) subroutines is assembled from tape 4 and the Section One buffer. Duplicates are eliminated. Each name in the CLOSUB table is searched for in the SUBDEF table. If found it is deleted from CLOSUB as being a dummy name. The table is then written as record 14 of file 5 on tape 2 preceded by its identification (9) and word count.

NONEXC - The table of statement numbers of non-executable statements is assembled from tape 4 and the Section One buffer. It is written as record 2 on tape 3.

TSTOPS - The table of statement numbers of STOP and RETURN statements is assembled from tape 4 and the Section One buffer. It is written as record 3 on tape 3.

TSKIPS - The table of IFNs to which skip type machine language statements may skip is assembled from tape 4 and the Section One buffer. It is written as record 4 on tape 3 followed by an end of file mark.

One is added to the last IFN used and it is left for Section One Double Prime. The END card indication for sense switch 4 is examined and bit 16 of word 20 set accordingly. This will be interrogated by Section IV. 8

SUBROUTINES - There are three subroutines used by Section One Prime:

TAP00- Table Assembly Program assemble tables written on tape 4 during Section One. It uses the parameters left by Section One to determine for a given table:

1. number of records on tape 4,
2. number of words in each record,
3. number of words remaining in the core buffer,
4. first location of core buffer.

The calling sequence in Section One Prime supplies the:

1. table identification (which also serves to locate the parameters left by Section One),

2. first location of buffer into which the table is to be assembled.

The routine tests each table for overflow against a table of permissable maximums.

Tables Assembled by TAP00

Name	Identification	Maximum word count
TEIFNO	0	750
TDO	1	750
TIFGO	2	600
TRAD	3	250
FORTAG	4	1500
FORVAR	5	1500
FORVAL	6	1000
FRET	7	750
EQUIT	8	1500
CLOSUB	9	1500
FORMAT	10	1500
SUBDEF	11	180
COMMON	12	600
HOLARG	13	900
NONEXC	14	750
TSTOPS	15	300
CALLFN	16	400
FMTEFN	17	750
TSKIPS	18	425

WAT00 - Writes assembled table on tape 2 preceded by identification and word count. Calling sequence supplies identification and first location of buffer in which table has been assembled.

FOR1DP - Writes assembled table on tape 3 for Section One Double Prime.

SECTION ONE-PRIME (709 Version)

This section is a terminal processor for Section One. It combines fragments of those tables which Section One generated as labeled buffer sized records. It makes certain modifications, primarily the replacement of EFNs with corresponding IFNs, which can only be accomplished when the entire source program has been reduced to tabular form.

The input to Section One-Prime consists of:

1. Various parameters describing tables (in cores)
2. Buffers containing terminal entries in tables (in cores)
3. Tables which Section One required for reference (FORSUB, END, DIMI, DIM2, DIM3, TAUl, TAU2, TAU3, FIXCON, FLOCON in cores.)
4. Tables which Section One did not require for reference. (COMPAIL on tape A4 in 8K version, on tape B2 in 32K version; TEIFNO, TDO, TIFGO, TRAD, FORTAG, FORVAR, FORVAL, FRET, EQUIT, CLOSUB, FORMAT, SUBDEF, COMMON, HOLARG, NONEXC, TSTOPS, CALLFN, FMTEFN, TSKIPS on tape A4.)

The output of Section One-Prime consists of:

1. Tables in cores: TAUl, TAU2, TAU3, FIXCON, FLOCON, FORVAL, TRAD, TIFGO, TEIFNO.
2. Tables on tape:

Tape B2:
File 1 is Source Program
File 2 is COMPAIL table
File 3 is Compail Record Count - FORSUB
File 4, Record 1 is FLOCON table.
Record 2 is FORMAT table.
Record 3 is SIZ table.
File 5, Record 1 is END table.
Record 2 is SUBDEF table.
Record 3 is COMMON table.
Record 4 is HOLARG table.
Record 5 is TEIFNO table.
Record 6 is TIFGO table.
Record 7 is TRAD table.
Record 8 is TDO table.
Record 9 is FORVAL table.
Record 10 is FORVAR table.
Record 11 is FORTAG table.
Record 12 is FRET table.
Record 13 is EQUIT table.
Record 14 is CLOSUB table.

Tape B3: File 1, Record 1 is NONEXC table.
Record 2 is TSTOPS table.
(8K version only, in cores for 32K version)

The tables are processed in the following order and manner:

COMPAIL - 8K Version. Each record of COMPAIL is read from tape A4. Each entry is examined for appearance of floating-point constants. Any such found are entered in the FLOCON table (if not previously entered) and replaced in the COMPAIL entry with the position of the entry in FLOCON. The COMPAIL record is then written in file 2 on tape B2. When all COMPAIL records have been read from A4 the contents of the Section One buffer are processed as the last record.

32 K Version. The contents of the Section One buffer are written as the last record of file 2 on tape B2.

FORSUB - The table of names and degrees of arithmetic statement functions, if any, is written after the compail record count in record 1 of file 3 on tape B2.

FLOCON - The table of floating-point constants and its word count are written as record 1 of file 4 on tape B2.

FORMAT - The table of format statements is assembled from tape A4 and the Section One buffer. It is written as record 2 of file 4 on tape B2; preceded by its identification (10) and word count.

FMTEFN - The table of references to fixed format statements is assembled from tape A4 and the Section One buffer. Each reference to a format is checked against the FORMAT table. If any referenced statements are missing an error list is developed for Section One Double Prime.

DIM1 - The table of one dimensional arrays is renamed SIZ.

DIM2 - Each entry in the table of two dimensional arrays has its two dimensions multiplied to form the size of the array. This table is added to SIZ.

DIM3 - Each entry in the table of three-dimensional arrays has its three dimensions multiplied to form the size of the array. This table is added to SIZ.

SIZ - The table is written as record 3 of file 4 on tape B2. It is preceded by EIFNO and its word count.

END - The END table is written as record 1 of file 5 on tape B2.

SUBDEF - The table of subprogram definition is assembled from tape A4 and the Section One buffer. It is written as record 2 of file 5 on tape B2; preceded by its identification (11) and word count.

COMMON - The table of common variables is assembled from tape A4 and the Section One buffer. It is written as record 3 of file 5 on tape B2; preceded by its identification (12) and word count.

HOLARG - The table of hollerith arguments is assembled from tape A4 and the Section One buffer. It is written as record 4 of file 5 on tape B2; preceded by its identification (13) and word count.

TEIFNO - The table of corresponding external and internal formula numbers is assembled from tape A4 and the Section One buffer. It is searched for duplicate external formula numbers. If such are found they are flagged as errors for Section One Double Prime. Those cases where Section One assigned more than one internal number are not considered as duplicates and the flag is deleted. The table is written as record 5 of file 5 on tape B2; preceded by its identification (0) and word count. It is retained in memory for use in processing tables discussed below.

TIFGO - The tables of IFs, GO TOs and ASSIGNs is assembled from tape 4 and the Section One buffer. Each external formula number is searched for in TEIFNO and its corresponding internal number replaces it in TIFGO. Any external formula numbers not found are set equal to 0 as an error signal to Section One Double Prime. When all entries have been modified the table is written as record 6 of file 5 on tape 2 preceded by its identification (2) and word count.

TRAD - The table of COMPUTED and ASSIGNED GO TO addresses is assembled from tape 4 and the Section One buffer. Each entry, which is an external formula number, is searched for in TEIFNO. When found it is replaced by the corresponding internal formula number. If not found it is set equal to 0 as an error signal to Section One Double Prime. When all entries have been treated the table is written as record 7 of file 5 on tape 2 preceded by its identification (3) and word count.

TDO - The table of DOs is assembled from tape 4 and the Section One buffer. Each entry is examined to determine if it originated from a DO or from an Input-Output List. If it originated from an I/O List the flag that so indicated is deleted. If it originated from a DO the EFN for the end of the DO is searched for in TEIFNO. When it is found the corresponding IFN replaces it in TDO. If not found it is set equal to 0 as an error signal to Section One Double Prime. In those cases where Section One assigned more than one IFN to an external number, the last such IFN is used so that the DO includes all instructions of the terminal statement. When all entries have been treated the table is written as record 8 of file 5 on tape 2 preceded by its identification (1) and word count.

FORVAL - The table of definitions of fixed-point variables is assembled from tape A4 and the Section One buffer.

CALLNM - The table of first and last internal formula numbers of statements containing references to subprograms is assembled from tape A4 and the Section One buffer.

Each IFN in FORVAL is searched for as a first IFN in CALLNM. If found it is replaced by the corresponding last IFN. When all entries have been processed the FORVAL table is written as record 9 of file 5 on tape B2; preceded by its identification (6) and word count. The CALLNM table is dead.

FORVAR - The table of usages of fixed point variables is assembled from tape A4 and the Section One buffer. It is written as record 10 of file 5 on tape B2 preceded by its identification (5) and word count.

FORTAG - The table of tag usages is assembled from tape A4 and the Section One buffer. It is written as record 11 of file 5 on tape B2 preceded by its identification (4) and word count.

FRET - The table of frequency statements is assembled from tape A4 and the Section One buffer. Each EFN in FRET is searched for in TEIFNO. When found it is replaced with the corresponding IFN. If not found, it is set equal to 0 as an error signal for Section One Double Prime. The FRET table is now sorted by IFN to form an ordered list.

The TIFGO table is now re-examined for any entries for COMPUTED GO TO statements. The IFN of each such statement is searched for in FRET. If found, the list of branch frequencies is reversed to correspond to the object program transfer vector. When all TIFGO entries have been examined, the FRET table is written as record 12 of file 5 on tape B2 preceded by its identification (7) and word count.

EQUIT - The table of equivalence statements is assembled from tape A4 and the Section One buffer. The table is reformatized to make those variables which are equated into strings of relativelized symbols. Any found to be inconsistant are flagged as errors for Section One Double Prime. Any redundancies are deleted. The table is then written as record 13 of file 5 on tape B2 preceded by its identification (8) and word count.

CLOSUB - The table of names of closed (library) subroutines is assembled from tape A4 and the Section One buffer. Duplicates are eliminated. Each name in the CLOSUB table is searched for in the SUBDEF table. If found, it is deleted from CLOSUB as being a dummy name. The table is then written as record 14 of file 5 on tape B2 preceded by its identification (9) and word count.

NONEXC - The table of statement numbers of non-executable statements is assembled from tape A4 and the Section One buffer. It is written as record 1 on tape B3 in the 8K version. It is left in cores in the 32K version.

TSTOPS - The table of statement numbers of STOP and RETURN statements is assembled from tape A4 and the Section One buffer. It is written as record 2 on tape B3 in the 8K version. It is left in cores in the 32K version.

MISC. - One is added to the last IFN used and it is left for Section One Double Prime.

SUBROUTINES - There are two subroutines used by Section One Prime.

TAP00 - Table Assembly Program assemble tables written on tape A4 during Section One. It uses the parameters left by Section One to determine for a given table:

1. number of records on tape A4,
2. number of words in each record,
3. number of words remaining in the core buffer,
4. first location of core buffer.

The calling sequence in Section One Prime supplies the:

1. table identification (which also serves to locate the parameters left by Section One),

2. first location of buffer into which the table is to be assembled.

The routine tests each table for overflow against a table of permissible maximums.

Tables Assembled by TAP00:

Name	Identification	Maximum word count	
		8K Version	32K Version
TEIFNO	0	750	3000
TDO	1	750	3000
TIFGO	2	600	2400
TRAD	3	250	1000
FORTAG	4	1500	6000
FORVAR	5	1500	6000
FORVAL	6	1000	4000
FRET	7	750	3000
EQUIT	8	1500	6000
CLOSUB	9	1500	6000
FORMT	10	1500	6000
SUBDEF	11	180	180
COMMON	12	600	2400
HOLARG	13	900	3600
NONEXC	14	300	1200
TSTOPS	15	300	1200
CALFN	16	600	2400
FMTEFN	17	500	2000
	18	0	0
END	19	15	15

WAT00 - Writes assembled table on tape B2 preceded by identification and word count. Calling sequence supplies identification and first location of buffer in which table has been assembled.

IV

SECTION ONE DOUBLE PRIME

Section One Double Prime's purpose is to detect source program errors. It does not add any further information to the tables created in preceding Sections, nor does it create any new tables for the use of succeeding Sections. Although Section One makes a determined effort to eliminate the errors in any one statement, no effort is made towards relating a particular statement to the rest of the program, nor would it be convenient for Section One to do so.

The errors that Section One Double Prime is able to find are mainly errors involving program flow, i. e., transfers to non-executable or even non-existent statements, and conversely, no transfers to executable statements which are not in the direct path of flow. These, and other errors, are found through a scan of the various tables of information which comprise the 5th file of tape B2 in the 709, and tape 2 in the 704. These tables are of such rigid format that it is easy to examine them for correct ordering and content. All errors found by Section One Double Prime are accumulated in an error list by several different error routines which are described at the end of this chapter. The table scan is only discontinued by table overflow or a machine error.

Section One Double Prime first initializes the error list with the count of missing format statements. The EFN's of missing format statements are left in the error list by Section One Prime in the 709 and read from tape 3 in the 704.

The following tables are then scanned.

TEIFNO

The TEIFNO table is scanned for duplicate statement numbers. Duplicate statement numbers are flagged minus by Section One Prime when it assembles the TEIFNO table. If any minus entries are found, they are entered in the error list by the ERROR routine.

TIFGO

Each of the 2 word TIFGO entries is examined for references to non-existent statement numbers, i. e., that there are not any zeroes except those peculiar to the particular TIFGO format. Section One Prime gives a non-existent EFN an IFN of zero. Further, each reference must be to an executable statement. Therefore, a cannot be in the table of non-executable statements, the NONEXC table. Each of the different types of TIFGO entries is checked by a specific subroutine within the TIFGO processor. This scan of the TIFGO table will result in the checking of the TRAD table, if one exists.

If any errors are found, they are entered in the error list by either the ERROR routine if β is non-executable or the NOBETA routine if β is non-existent.

In order to do a quick flow analysis the IFNα of a TIFGO statement is entered in the ALPHA table, and the references (IFN β's) are entered in the BETA table. The number of branches associated with a particular TIFGO entry is also entered in the ALPHA table with the IFNα . All TIFGO entries, except ASSIGNS, are entered into these tables. The position of an ASSIGN in the source program does not effect the path of flow in the program.

The ALPHA and BETA tables are internal to Section One Double Prime and have the following format.

ALPHA

DECREMENT ,	TAG ,	ADDRESS	
N	0	IFNα	N:: Number of branches.

The table of STOP and RETURN statements, TSTOPS, is a part of the ALPHA table.

BETA

DECREMENT ,	TAG ,	ADDRESS
0 or 1*	0	IFN β

*Decrement will be 1 if β is non-executable.

The BETA table consists of the β's from TIFGO, the entire TRAD table, and the last IFNα + 1 in the program. In the 704, the inclusion of machine language necessitated the building of a second BETA table, the BETA2 table. This second BETA table is an extension of the BETA table and has the same format. BETA2 consists of the TSKIPS table, table of skip type instructions such as CPY, CAS, LBT, etc., and the α+ 1 of conditional transfers from TIFGO. Conditional transfers are TXH, TIX, TMI, etc.

FLOW ANALYSIS

Example 1

	.
	.
α	GO TO β
α + 1	DIMENSION X(5)
α + 2	FORMAT (F8. 3)
	.
	.
	. More non-executable state-
α + M	A=B + C ments.
	.
	.
	.

A brief flow analysis is performed using the information in the ALPHA, BETA, and NONEXC tables. Each α in the ALPHA table is the termination of a path of flow in the source program. Therefore, there must be a transfer to the first executable statement following each α in the ALPHA table. That is, that the IFNα + M in Example 1 must be in the BETA table, since β's are statements transferred to. In reference to Example 1, the flow analysis processor will first search the BETA table for α+ 1. Not finding α+ 1 in the BETA table, it will then search for α+ 1 in the NONEXC table, and a match will be found. Upon finding α+ 1 in the NONEXC table, the processor will then follow the same procedure for α+ 2, α+ 3,....., α + M. In searching for α+ M, if the processor finds it in the BETA table, the processor will then proceed to execute a flow analysis for the next α in the ALPHA table. However, if α+ M is not in the BETA table, and since it is an executable statement, α+ M will not be in the NONEXC table. Therefore, if α+ M is not in either the BETA or NONEXC tables, it is a part of the program not reached, i. e., an executable statement with no path of flow to it. If any errors are found, they are entered in the error list by the NOBETA routine.

TDO

The TDO table is examined for DO statements that specify an illegal β. The three legal references checked for by Section One Double Prime are:

1. That the IFNβ exists, i. e., that the reference β is not zero.

2. That the IFNβ is executable, i. e., that the reference β is not in the NONEXC table.

3. That the IFNβ is not a transfer, STOP, or RETURN statement, i. e., that the reference β is not in the ALPHA table.

If any errors are found, they are entered in the error list by both NOBETA and αDOβ routines, in that order.

FRET

The number of branches for a TIFGO statement is saved in the ALPHA table with the IFNα during the scan of TIFGO. Section One Double Prime ignores statement numbers in the FRET table which are not in the ALPHA table, but saves any statement number where the count of branches in FRET is greater than the count of branches shown in the ALPHA table. Section Four ignores extra frequencies given for statements other than TIFGO statements, but would be confused by misinformation generated when there are more frequencies given than there are branches. If any errors are found, they are entered in the error list by the NOBETA routine.

EQUIT

If Section One Prime has found any inconsistent equivalences when assembling the EQUIT table, it sets an error flag at the beginning of the table and only enters those variable names which are erroneous, and sets another flag at the end of the list. The errors are entered in the error list by the ERROR routine.

If any errors have been found by Section One Double Prime, it spaces the System Tape to the diagnostic and reads in D001. This is the only section of FORTRAN that does not use the usual diagnostic caller. If no errors have been found, tape B2 in the 709, tape 2 in the 704, is spaced over the 5th end of file mark and control is transferred to 1 to CS to continue compilation.

ERROR ROUTINES

The three error routines in Section One Double Prime make entries in a common error list which begins at location -1 and builds downwards. The error routines are reached by means of a TSX , 4 and control is returned to 1, 4.

ERROR

Makes a two word entry in the error list.

	DECREMENT ,	TAG ,	ADDRESS
WORD 1	C(IR4)	0	hash or C(IR4')*
WORD 2		CONTENTS OF MQ	

* The address of the first word may contain the location of a TSX to one of the checking routines that has called the ERROR routine.

NOBETA

Makes a one word entry in the error list.

	DECREMENT ,	TAG ,	ADDRESS
WORD 1	C(IR4)	0	IFN α

α DO β

Makes a two word entry in the error list. A TSX to α DO β is preceded by a TSX to the NOBETA routine.

	DECREMENT ,	TAG ,	ADDRESS
WORD 1	IFN α	V	IFN β
WORD 2		S Y M B O L	

The following is an example of a Section One Double Prime error list resulting from a problem run on the 32K System in the 709.*

LOCATION	DECREMENT ,	TAG ,	ADDRESS	ENTRY MADE BY	REASON
-1	L(ETE)	0	hash	ERROR	The EFN is duplicated in the source program
-2	IFN α	0	EFN α		
-3	L(TMNO2)	0	IFN α	NOBETA	β_2 is non-existent in TIFGO statement α. Statement is in the form of αIF (E) β_1, β_2, β_3
-4	L(BNOTX)	0	L(TMX3)	ERROR	β_3 is non-executable for the preceding TIFGO statement
-5	hash	0	IFN α		
-6	L(NOTRA)	0	IFN α	NOBETA	Statement number α is a part of the program not reached
-7	L(CONBET) or L(DOBX)	0	IFN α	NOBETA	TDO statement α
-8	IFN α	V	IFN β	αDO β	specifies a β that
-9	S Y M B O L				is either 1. CONBET in the ALPHA table 2. DOBX in the NONEXC table
-10	L(TOOFRQ)	0	IFN α	NOBETA	More frequencies have been given than there are branches for TIFGO statement α.
-11	L(BADEQU)	0	hash	ERROR	An inconsistent equivalence has been made concerning the variable Vn
-12	BCD VARIABLE Name (Vn)				

*L( ) implies the location of the symbol within the parentheses.

SECTION TWO

Preliminary Description Of The Problem.

A. Tags Created By Section One.

Section Two compiles the instructions necessary to compute and index so that the symbolic index registers, (tags), set up in Section One for tagged instructions will contain their proper values. These tagged instructions compiled by Section One refer to arrays, i. e. subscripted variables. For instance,

$$X = A(I, J)$$

will be handled by Section One as follows:

```
CLA   A+1,τ
STO   X
```

Section One makes up a table of these symbolic tags (τ). The symbolic tag is, in fact, a subscript combination [such as (I, J), (K, J, I), or (M)] with given dimensions and coefficients. The tags are divided into 3 classes, 1, 2, and 3 dimensional, and separate tables, Tau1, Tau2, and Tau3 are composed for these respective classes. The table entry for a particular tag [for instance, $(C1I + a_1,\ C2J + a_2,\ C3K + a_3)$] will contain (1) its symbols I, J and K, (2) its coefficients C1, C2, and C3, and the dimensions D1 and D2 of the array concerned. The tag is not affected by the addends a1, a2, and a3. The effect of the addends is handled in the address part of the tagged instructions.

B. DOs.

1. Basic Format

The biggest part of Section Two is compiling computing and indexing instructions for DO loops and tags within them. A general format for a DO loop is as follows:

```
A     LXD ___,τ  }  Section Two
B     OPN        }
      OPN        }  Section One instructions
      OPN        }
B1    TXI *+1,τ ,   (Decrement )
B2    TXL B,τ   ,   (Decrement )
B3    TIX *+1,τ ,   (Decrement )
```

This, of course is the simplest case. The TXI instruction at B1 increments the tag, the TXL tests the DO loop, and the TIX resets the tag to its original load value at A. Often, of course, the LXD address and the TXI, TXL, and TIX decrements are variable and may require computing and initialization instructions at location A. Further complications may result from DO nesting,

from transfers out of DO's, and many other factors some of which will be discussed herein.

2. TXI Variations

The TXI decrement is a function of (a) the permutation of subscripts in the tag, (b) the coefficients of the subscripts in the tag, and (c) the parameters N1, N2 and N3 of the DO. Sometimes the TXI must be expanded as follows:

```
TXI  *+1, τ,     (Decrement)
SXD  α, τ
TIX  *+1, τ,     (Decrement)
```

where the SXD reinitializes the TXL decrement (test) of an inner DO within the nest. A given TXI format with a set formula from which its decrement(s) may be computed is called a TXI BLOCK NUMBER. There are 6 such block numbers. This block number depends on the permutation of subscript order in relation to order of DO nesting (this permutation defines a group number, of which there are also 6), the position of the subscript in the tag, whether or not this tag will be used to test the loop, and whether there will be a carry in this loop.[1]

3. TXL (test)

The TXL (test) of a DO loop will generally use one of the tags which occurs within the loop. This tag will be chosen for its simplicity. If there are no tags in the DO, a counter tag will be created for this purpose.

4. TIX (Reset)

In inner DO's controlling a tag, a TIX is used to reset a tag by the amount it has been bumped upon satisfaction of the DO. This of course is not necessary for the outermost DO controlling a tag, for it will be reinitialized upon reentry.

5. Special tags created by Section Two in DO's.

a. Stored Counter. If the symbol (I) of a DO is required to be updated in its own cell within a DO (because of its appearance on the right side of an arithmetic expression or because a transfer out of the DO) a counter is set up as a tag, therefore

```
PXD  0, τ
STO  (I)
```

is compiled at the beginning of the DO, and

```
SXD  (I), τ
```

at the end of the DO.

[1] Sometimes an inner loop uses the test of a DO further out the nest for its test. i. e., 1 TXL may be sufficient for 2 or 3 DO's within a nest. When this situation occurs, the DO whose TXL is eliminated is considered to have CARRY.

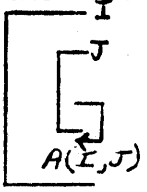
b. Normal Counter. A Normal Counter is created to record the incrementing of DO's if there is no other tag on which to test.

c. Reset Tag. If there is a transfer out of one DO into another (outer) DO in the nest, a Reset Tag is created to record how much another tag has been incremented, and to reset it by that amount at the time of the transfer. The resetting is done as follows:

SXD	α, τ	τ = Reset Tag
TIX	*+1, τ'	τ' = Tag to be reset
TRA	(out of DO)	

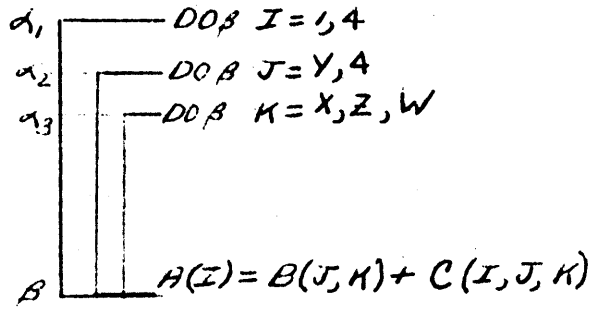
Section Two does not actually compile the SXD and TIX, but makes entries in a table (TRASTO) to cause Section Three to compile these instructions when it compiles the TRA instructions.

d. Added Tag. Consider the nest



with a transfer from the J loop into the I loop. Since the subscript J is dependent on the transfer from the J loop, an Added Tag (I, J) is created in the J loop to record the increments to (I, J) by the J loop. [2]

e. Loading and Initialization. Index Register loading and decrement initialization is always done as far out the nest as possible. For instance, consider the nest



In this nest, all initialization and loading would be done at $\alpha 1$. If, however, X were defined between $\alpha 2$ and $\alpha 3$, then the loading for (J, K) and (I, J, K) would have to be done at $\alpha 3$. Similarly, the test initialization for the K loop would have to be done after the definition of Z. The level of TXI and TIX initialization depends on all three parameters, and so will be done after all these parameters are defined, but still as early as possible.

In short all Loading and Initialization will be done at the very earliest point of complete definition.

[2] J in the tag (I, J) in the I loop is called a Relcon (relative constant) because it is not under the control of a DO.

C. Relative Constants.

Relative Constants are subscripts of a tag that are not currently under control of a DO. Pure Relcons are tags in which all of the subscripts are Relcons. Section Two compiles subroutines to compute load values for pure Relcons. The subroutine will be called at points of definition of any of the subscripts. An exception is the 1-dimensional pure Relcon. In this case the TSX to the subroutine is not needed, and an LXD I,τ (where I is the Relcon) is compiled at the point of definition. Examples.

1. Dimension greater than one.

Source Program		Object Program		
α	I = 5	α	CLA	[L(5)]
			STO	I
			TSX	A)1G, 4
			LXD	C)1G, τ
β	X = A(I, J)	β	CLA	A+1, τ
			STO	X
		A)1G		Computes load value for (I, J) and stores in C)1G

2. 1 Dimensional.

Source Program		Object Program		
α	I = 5	α	CLA	[L(5)]
			STO	I
			LXD	I,τ
β	X = A(I)		SXD	C)1G,τ
			LXD	C)1G,τ
		β	CLA	A+1,τ
			STO	X

Closed subroutines are also used sometimes in DO's; namely when a relative constant of a tag in a DO is defined within that DO. An example is:

Source Program		Object Program	
α	DO β2 I=1,2	α LXD	[L(1)], τ
β_1	M1 = M1 + 1	β_1 CLA	M1
β_2	A(I, M1) = A(I)	ADD	[L(1)]
		STO	M1
		SXD	6) + i, 4
		TSX	A)1G, 4
		LXD	C)1G, τ'
		CLA	A + 1, τ,
		STO	A + 1, τ'
		β_2 TXI	*+1, τ, 1
		TXI	*+1, τ', D
		TXL	β_1, τ, 2

If, however, M1 in the above problem had not been defined within the DO, the closed subroutine technique would not have been used, but rather the normal open subroutine would have been compiled at α.

Logic of Section Two.

To carry out the analysis and to deal with the various complexities involved, there are six logical blocks in Section Two.

Block 1. Nest analysis, flow analysis.
Block 2. Subscript combination analysis.
Block 3. Relative constant subscript analysis.
Block 4. Compilation of subroutines for computing relative constant index values.
Block 5. Compilation of loop initialization, incrementing, and testing instructions.
Block 6. Reordering the DO file for input to Section Three.

The write-up is divided into two parts. One part is a detailed description of the work done by each block, and the other gives general information about the Section which either is essential to more than one block, or does not fit in conveniently with a block description.

BLOCK 1. The task of this block is to examine the DO nesting structure and the flow of the program as regards possible transfers out of DO's and also to build up such general information as will be required by later blocks. The difficulty in following the coding of these first blocks lies in visualizing the particular DO configuration a routine is searching for, so examples will be when possible.

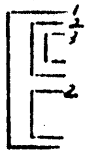
The information which Section One extracted from DO statements and Input-Output lists is contained in the tape table TDO, which, on being read in, is further expanded into the 9 word table DOTAG to accommodate the results of analysis.

First of all, a complete pass is made over DOTAG, determining for each DO the following:

i.) If none of the N's of the DO are variable, the quantity X where

$X=\left[\frac{N2-N1+N3}{N3}\right]\times N3$, $[\ ]$ signifying integral part. This is required for computing end test decrements in Block 5.

ii.) The level number associated with the DO. This is 1 if the DO is the outermost of a nest, 2 if the second outermost, and so on. E. G.,



iii.) Whether any of the rules for DO nesting have been violated by the source program.

iv.) The possibility of carry between the current DO and the next lower level DO of the nest, if any. No carry is possible if the N3 of the lower level DO is variable. If the DO's in question are

α_1 DO β_1

and α_2 DO β_2

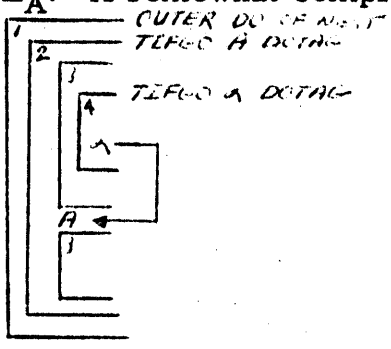
and $\alpha_2=\alpha_1+1$ and $\beta_1=\beta_2$ then carry type 1 is indicated as possible for the higher level DO. Otherwise carry type 2 is indicated for that DO.

Subroutine FLOW.

A thorough analysis is now made of the structure of transfers within DO's. This analysis is carried out nest by nest, i. e., the DOTAG table is stepped through, forwards, looking for DO's of level 1, since each such DO signifies the beginning of a new nest. Having thus found the outer DO of a nest, the TIFGO table is searched for a TIFGO (i. e., location of a transfer exit) which lies within the nest. If there are no such TIFGOS, the program proceeds with the next DO nest, and so on until the DOTAG table is exhausted. However, if transfers are found within a nest, the following analysis is made:

First, the highest level DO which contains the Tifgo α is found, and bit 1 of word 7 in DOTAG is set to indicate that this DO has a transfer within its range. This DO will now be referred to as the TIFGO α DO. At this point, an entry is made for this Tifgo α in the TRALEV table, which is used by Section Three. Then a flow analysis is carried out on each separate possible address A of the Tifgo, in the Subroutine FA.

Subroutine FA. The highest level Dotag containing A is obtained, and will be referred to as the Tifgo A Dotag, with level L_A. The bit corresponding to L_A in word 8 of the Tifgo α Dotag is set, to indicate that this DO has a transfer within its range to level L_A. A somewhat complex example of the above might be:



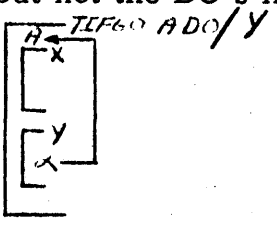
In this case, L_α is 4, and L_A is 2. The second bit in word 8 of the Tifgo α Dotag would be set, indicating the transfer level.

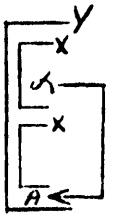
The entry already made in Tralev for the Tifgoα is implemented by a word giving the address A and its associated level. Also, the sign bit of word 6 is set in all Dotags for which the Tifgo transfers out of range. In the above example, this would be the TifgoαDotag, together with the DO with level 3 surrounding it.

Next, carry bits are erased for DO's satisfying both the following conditions:

i.) The DO is one level higher than the Tifgo A Dotag.
ii.) The β of the DO lies between the Tifgo α and the Tifgo A.

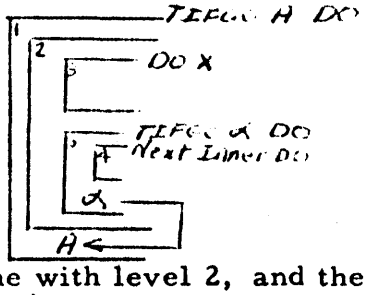
For example, the carry bits are erased for DO's marked "X" in the following configurations, but not the DO's marked "Y".





The final part of the FA routine - RNC, reset no carry condition - obtains information for possible no-carry situations. For all DO's out of which this Tifgo transfers, the DO of the next higher level is examined, and if its β is smaller than the Tifgoα, then L_A, is entered in word 7 of this Dotag, unless some previous L_A, inserted for some other address of a Tifgo , was larger, in which case it is left unchanged. This largest L_A is referred to as the No Carry Transfer Level.

An example is given:



Consider this DO configuration. The RNC routine works in this way. It starts with the Tifgoα DO and looks for the next inner DO whose β < Tifgoα. The DO of level 4 clearly satisfies the conditions, and so L_A (i.e., 1) would be entered as its No Carry Transfer Level. RNC next takes the next lower DO in the nest to the Tifgoα DO, i.e., the one with level 2, and the DO X now satisfies the condition for the next inner DO whose β < Tifgoα, so L_A (1) would be entered as its No Carry Transfer Level, too. The Tifgoα transfers out of no more DO's of the nest, so the analysis ends at this point. (If, subsequently, another address A_1 were found for this Tifgo, just above the Tifgoα DO, in level 2, then the No Carry Transfer Level of the DO of level 4 would be changed to 2, but that for the DO X would be unchanged. The results of RNC are used in the carry subroutine in Block 2.

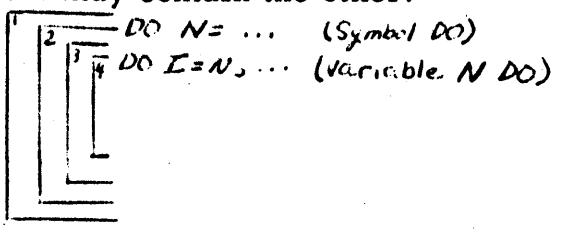
The FA analysis is now complete for this Tifgo A, and is repeated for any other addresses of the particular Tifgo under analysis.

The next part of Block 1 deals with DO's, one or more of whose N's are variable. The following routines examine the ways in which such N's may be defined.

Routine SV. The first of these routines, SV, looks for possible definition by another DO in the same nest, whose symbol is the variable N in question. The

two DO's (the variable N DO and the equivalent symbol DO) may take either of the following configurations:

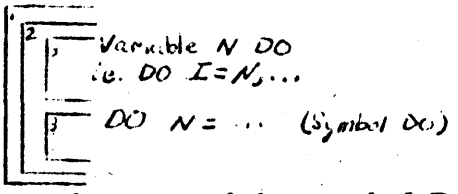
i.) One of them may contain the other:



The value N is fixed anew every time the loop DO for N is entered. It can therefore be said that the level of definition of the variable N in the DO for I is the level of the symbol DO, i. e., 2, and this is entered as the highest level of definition so far in the DO for I Dotag, for that particular variable N. A bit is set in the symbol Dotag to indicate that each new value of the symbol must be stored.

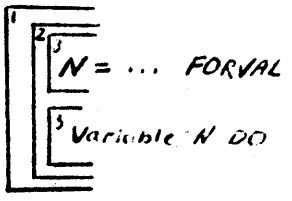
The symbol DO can not be within the variable N DO, as this would change the value of N while executing the variable N DO, and would therefore be a source program error.

ii.) The two DO's are in different subnests:



If there are transfers out of the symbol DO, these would define N. The level of definition of N is then the highest such transfer level, if it is not greater than that of the connecting DO (2 in this case). Otherwise, it is the level of that connecting DO. This level of definition for the particular N is entered in Dotag, provided that no higher level has previously been entered for this N.

Routine TS4VA L. The second routine dealing with variable N's is TS4VAL, which searches for their definition by arithmetic statements, and Read statements. The FORVAL table contains the names of variables so defined, together with the internal formula number attached to their point of definition. Only those occurring within a DO-nest are dealt with further. If such a forval entry is the same symbol as the variable N of a DO within the same nest, its level - i. e., the highest level DO containing both the forval and the variable N DO - is then the level of definition of that N, e. g.,



The level of definition of N is 2 (Not 3 even though the forval lies within a DO of level 3)

Routine RH. Within a DO for a Symbol I say, there may be a use of I, not as a subscript, but as an ordinary fixed point variable on the right hand side of an arithmetic statement, or in some IF expression. A table Forvar of such right hand side variables is searched in conjunction with the DO's, and for such DO's containing a Forvar for their symbol, a bit is set in Dotag to indicate that the current value of the DO symbol must be stored.

Routine LB. The nests of DO formulas are scanned and the bits signifying 'This DO contains a transfer in its immediate range' are used to set to one, if necessary, a bit meaning: ' This DO contains a transfer in its extended range'. This bit is used only in section 4.

Routine EB finally writes Dotag on tape, one nest per record, for processing in Block 2.

(These last 3 paragraphs were taken from P. 43 of 'the Tome. '

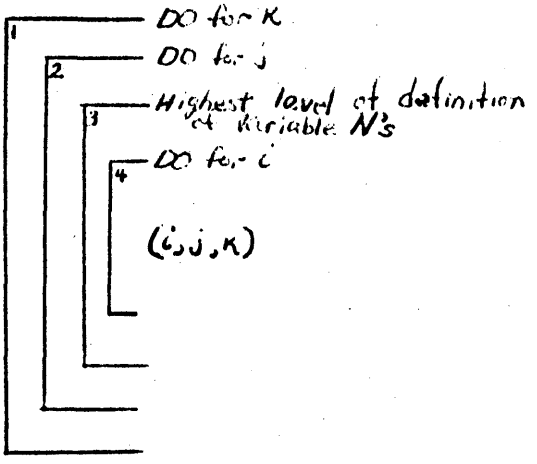
BLOCK 2. (Throughout this write-up, the symbol α refers to the beginning of a DO loop, and β to the end, unless otherwise noted.)

The Block 2 analysis is carried out for each subscript combination occurrence, at least one of whose subscripts is under the control of a DO. Only the areas within DO-nests need therefore be examined, and so the search for tags is carried out nest by nest, and, within that nest, DO by DO. The order in which the subscripted variable occurrences (i. e. , Fortag table entries) are dealt with is as follows: The last DO of the nest is selected (this is either the highest level DO in the nest or the highest in the last subnest), and any Fortag entry lying within and controlled by this DO is analyzed. Then the next back DO is taken, and Fortags controlled by this DO are dealt with, and so on until the end of the nest. Thus inner nested Fortags are always analyzed first.

On completion of the analysis of a tag, it is marked as having been dealt with, together with identical tags within the current DO.

Subroutine IDENT. Having selected a Fortag which has not been previously dealt with, certain basic information is established by IDENT. The DO's and their levels, controlling each subscript symbol are located. If such a controlling DO is not found for a subscript, it is called a relative constant or relcon. (If all the subscripts are relcons, the Fortag is ignored, as it will be dealt with by Block 3.) The subscripts controlled by DO's are now further divided into true 'dosubs, ' and DO-relcons or DORC's. The distribution between the two is made in the following manner:

The highest level of definition of variable N's of the controlling DO's is obtained. Then, all subscripts whose controlling DO's are of a lower level are termed Dorcs. The reason for this is that the index initialization values for the tag can not be computed until the highest N definition level, by which time the Dorcs have been assigned values by their DO loops, and are fixed point quantities (much the same status as a relcon defined by an arithmetic statement). An example is given:



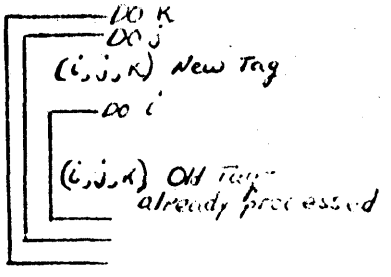
In this example, if level 3 is the highest level of definition for N's of DO's for i, j, k, then by the rules already given, k and j will be Dorcs, and i will be a true Dosub. The index loading will be done before the beginning of the i loop, and bits are set in the j and k Dotags to indicate that counters j and k must be stored, and updated as fixed point variables within those loops.

Now, all subscript symbols of the combination fall intoone of the three categories: Relcons, Dosubs, and Dorcs.

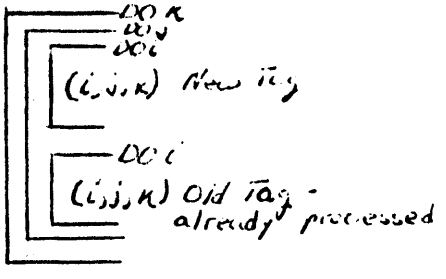
Subroutine NAME. Section One gives every subscript combination a tag name, i. e., a Tau table reference number which is a key to the entry containing the subscript symbols, coefficients, and dimensions. Thus, all identical subscript combinations wherever they occur, have been given the same tag name, and if this is left unchanged, will result in the same tag storage cell, which is updated as the tag is updated.

The NAME routine searches for situations where it would be incorrect to have the same tag, and it changes the name, where necessary. A search for an identical Fortag to the current one, but which has already been dealt with, is made throughout the nest. If such a one is found, the current tag is given a new tag name (i. e., the next new tag number free). The way in which tags are selected for processing in this block (selecting innermost tags first, and marking off identical ones within the same DO) ensures identical tags found under this routine being in different DO's, with their subscripts having different status. Two differing situations might be:

I



II



In example II, the new tag strictly need not be renamed, since no ambiguity could arise, however, this is an unimportant detail. If the search described results in the current tag name being changed, this change will also apply to identical tags within the same DO region. Thus in example I, all (i, j, k) tags in regions $\alpha_j \cdots \alpha_i$ and $\beta_i \cdots \beta_j$ must be given the new name also, and in II, (i, j, k) tags in the upper DO for i must be changed. This is not done by block 2 (which only marks off tags as having been analyzed, and has no access to the compiled instructions), but the regions over which the new name applies, together with the new and old names, are passed on to Section 3 in the Changetag table, and that section makes all the

necessary alterations in the Compiled Instruction tags. The routine which makes the Changetag entries and obtains regions is called SPC.

Finally, routine NAME makes an entry in the table Name or Namkey for the new tag. This table is merely a catalogue of new names together with their original names or Tau references.

STATE B.

Only those Fortags which are 'mixed relcons,' i. e., tags at least one of whose subscripts is a Relcon, are processed through State B and this will be dealt with next.

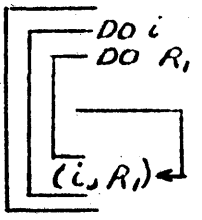
A relcon subscript can be defined in two distinct ways:

i.) By a transfer out of a DO for that symbol.
ii.) By appearing on the left hand side of an arithmetic statement or in a read statement, both of which result in entries in the Forval table, giving internal formula number and symbol.

Definition by the first method is examined by the DSDR routine.

Routine DSDR.

A DO of higher level than the Fortag-containing DO is looked for which has as its symbol the relcon (or one of the relcons) of the subscript combination. Having found such a DO, subroutine Trawrd searches for definitions of this relcon R_t, say, by transfers from the DO for R_1, which terminate above the lowest Dosub level. * For example.



The different types of mixed relcon are dealt with separately.

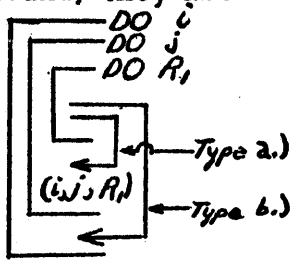
i.) 1 relcon with 1 dosub. (There may or may not be a 3rd subscript which is a dorc.)
Having found an R_1 dotag with a transfer out of its range, a search is made within the R_1 dotag for an equivalent subscript combination. If such a tag is found, the required value would be in the index register at the time of transfer. The current mixed relcon will, however, have been given a new tag name by the Name routine described earlier, so a TRASTO table entry is made to indicate to Section 3 that instructions should be compiled at the point of transfer to save the old tag (within the R_1DO in the new tag name cell for the mixed relcon.)

If no identical tag is found in the R_1DO, an added tag is created, and given the next available new name. A Trasto entry is made as before so that Section 3 will provide the necessary index saving instructions.

* Transfers which define two relcons at once are not dealt with at this point.

ii.) 1 relcon with 2 dosubs.

In this case, if transfers out of the R_1DO are found, they are divided into two categories:



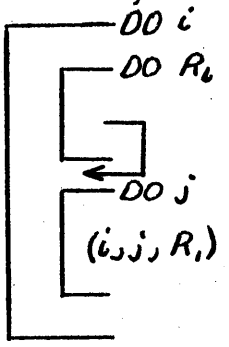
a.) Those which terminate above the DO containing this tag.

b.) Those which terminate above the lowest dosub DO level, but below the containing DO.

If type a.) transfers occur, the same procedure is carried out as for case i.) 1 relcon, 1 dosub. If type b.) transfers occur, a sense light is set to indicate their existence, but all possible type a.) relcon definitions are found and dealt with before considering type b.).

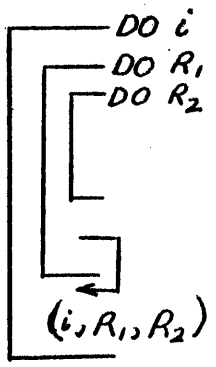
If type b.) transfers exist, the method used above would become too awkward because of the transfer out of the j range. The difficulty is therefore avoided by making i a dorc and arranging for it to be a stored counter.

DSD115. There is yet a third possibilty given in this configuration. The R is defined between the dosub DO's i and j. This is essentially the same situation as when type b.) transfers, described above, occur. The same method is used in this situation, namely the i dosub becomes a dorc and indication made in the i dotag to make i a stored counter.



iii.) 2 relcons with 1 dosub.

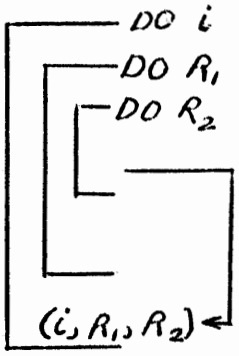
Suppose for convenience the subscript combination is (i, R_1, R_2). Having found an inner DO for, say, R_1, only those transfers which define the R_1 relcon are dealt with. Given such transfers, identical tags are searched for in those regions of transfer.



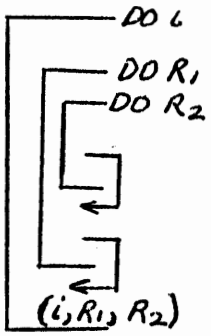
e.g., in this situation, the regions $\alpha_{R_1} \rightarrow \alpha_{R_2}$ and $\beta_{R_2} \rightarrow \beta_{R_1}$ are searched for an (i, R_1, R_2) tag. If such a tag is not found, an added tag is created (as described for the earlier cases). Trasto entries are made for both regions to save the (i, R_1, R_2) tag within the R DO on the point of transfer, in the mixed relcon tag cell.

The routine 2R is now used to search within the R_1DO for an R_2DO, looking for transfers which define both relcons. This can occur in two ways:

i)



ii)

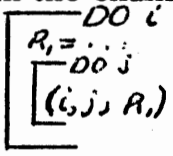


An identical tag, or added tag is looked for within the R_2DO and if there is none, an added tag is created. Trasto entries are made for either or both of the above two types of transfer situations that occur.

If there are no more R_2DO's within the R_1DO, control is returned to the single definition search routine DSDR.

Routine DS4VAL. The Forval table is now searched for the occurence of a relcon symbol within the Fortag-containing DO. If there is one, bits are set in Dotag so that all the dosubs are stored counters. A TSXCOM table entry is made so that after the relcon definition there will be a TSX to a subroutine which uses the latest values of relcons and dosubs to compute the index value. This is the method used by Block 3 for pure relcons.

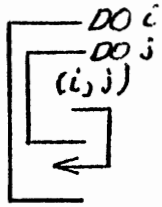
For the 2 dosubs and 1 relcon case, as well as the above search within the fortag-containing DO, a search is made in the areas between the two dosub DO's $\alpha_i \rightarrow \alpha_j$ and $\beta_j \rightarrow \beta_i$ in the example. If the R definition i is made a dorc, and the counter i is stored. The situation is reduced to 1 relcon and 1 dosub.



This is the end of State B. All types of subscript combination are then processed through State A, which is controlled by subroutine Branch. This is in three parts: INS, 2NS, 3NS, dealing with one dosub, two dosubs or three dosubs as the individual case requires. Duplicates are counted only once.

INS routine for 1 dosub. Apart from allotting a group number of 6 to the subscript combination (which indicates to Block 5 the ordering of subscript levels within a combination), this routine is mainly concerned with whether the tag can be used as a counter and as a test for the DO loop. Such is the case if the tag is a simple dosub with a coefficient of one. If this is an unstored counter, a Trasto entry is made to store it where there is a transfer out of the DO.

2NS routine for 2 dosubs. After assigning to the tag group number 1, if the first dosub level is higher than the second, and group number 6 otherwise, this routine calls upon the RESET routine, which is concerned with resetting the tag if there are transfers out of the DO's, which terminate between two dosub levels.
For example.



See write-up on Reset tag in general information part. The Reset routine is described in detail under the description of the 3NS routine for three dosubs. Of course, if there are no transfers out of the inner dosub, terminating between the two dosub levels the Reset routine is bypassed.

(If this case is really a 3 dosubs tag, reduced by duplicates to the 2 dosub case, the carry routine is executed as described later.)

3NS routine for 3 dosubs. First of all, a group number is assigned to the subscript combination according to the order of the dosub levels:

Level of subscript 1 > level of s. 2 > level of s. 3 = group no. 1
Level of subscript 2 > level of s. 3 > level of s. 1 = group no. 2
Level of subscript 2 > level of s. 1 > level of s. 3 = group no. 3
Level of subscript 3 > level of s. 1 > level of s. 2 = group no. 4
Level of subscript 1 > level of s. 3 > level of s. 2 = group no. 5
Level of subscript 3 > level of s. 2 > level of s. 1 = group no. 6

Next the possibility of carry is investigated between the left and center subscripts. If the left subscript is one level higher than the center, then the Carry routine is called upon.

CARRY routine. This routine rules that carry is possible for the left and center positions if the following conditions are satisfied:

i.) Carry of either type is indicated for the DO of the left subscript.

ii.) The "no carry transfer level" of the inner DO is lower than the lowest dosub index register initialization level (see RNC routine, Block 1).

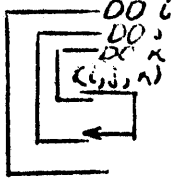
iii.) The total range through which the left subscript increases within its DO equals one increment of the center subscript. That is,

$[X]$ for left DO x coefficient of left subscript = $[N_2]$ for center xd_1 .

A bit is entered in CARWRD, (which is passed on to Block 5 in tagtag table) via bit 11 if type 1 carry, and bit 13 if type 2 carry.

A similar investigation is carried out for the center and right subscripts. If the center subscript level is one higher than the right then the Carry routine is again called upon for these two positions, changing left and center for center and right respectively. If carry is possible, bit 12 is set for type 1, in CARWRD and bit 14 for type 2.

It is now determined which subscripts will have to be reset because of transfers out of inner DO's. If there are transfers out of the extended range of any of the inner dosubs, which terminate above the index initialization level, then those particular dosubs will have to be reset (see description of Reset tag in general information section). For example:



If j and k are true dosubs, then whether i is a dosub or a dorc, k will have to be reset because

the index initializing will have been done outside the j loop.

First, subscripts to be reset because of transfers outside the innermost dosub DO are dealt with by the RESET routine, then those because of transfers out of the next innermost. Reset may therefore be called twice.

RESET routine. The resetting is done by creating another tag which measures the amount by which the subscript combination must be reset at the point of transfer. The tag created depends of course on the subscripts being reset, their coefficients and the dimensions involved. Details about each tag created are entered in the Retab table so that tags will not be duplicated if the required one already exists. The new tag details are entered in Drumtag table for Block 5 to compile the necessary indexing. (This table has the same format as the Tagtag table and is called Adtag in Block 5.) Then an entry is made in Trasto so that on each transfer, the instructions

```
SXD    *+1, Reset added tag
TIX    *+1, Tag to be reset
```

will be compiled by Section 3.

The above process is modified if the subscript to be reset is the left-most and has a coefficient of one, and if the N of the DO being transferred out of is not variable. A normal counter, if there already is one, will suffice for a reset tag. Since such a counter would start at N_1, and a reset tag must always start at zero, a Trasto entry is made which causes Section 3 to compile a TXI *+1, counter, $-N_1$, to correct this. The routine then proceeds to make the SXD, TIX Trasto entry described earlier. Both of these entries are made by the RSR subroutine. If however, no counter has been found so far, details of the tag required are entered in the Adtag table * so that RSR can be called upon later, when a counter has been established (when the nest has been completely dealt with).

Routine TAG continued. When all reset possibilities have been dealt with, the results of the whole analysis of the subscript combination are output as a Tagtag tape table entry. This provides Block 5 with information so that it can compile the appropriate initializing and indexing instructions at the appropriate points.

The only further information derived from the subscript combination is for deciding on end tests for the DO's controlling its subscripts. A bit is set in a controlling DO if the corresponding subscript does not carry over into the next left, to indicate that there must be a test on this DO.

If the subscript combination consists solely of dosubs (no relcons or Dorcs) then the tag is eligible for the end test of any of the corresponding DO's, unless there are known to be counters which would clearly be better tests. A test table number is formed from subscript position, carry bits, and group number. If the number thus derived is greater than that already held for the DO, then this is the best test found so far, and its name is stored in the dotag.

* This table is not to be confused with the Adtag table of Block 5.

The tag has now been fully processed, and all identical tags within the same DO are also marked as having been analyzed. Control then selects the nest tag within this DO for processing until all tags within and controlled by the DO have been exhausted.

Routine DOFEND. At this point, if a counter for the DO (that within which the tags have just been analyzed) is required, and one has not been found in fortag such a counter is created (in subroutine Makesc). Also if there is a possibility that no end test is needed for this DO, it is noted. (Bit 20 of word 9).

This completes the analysis for the DO, and the next back DO in the nest is selected, and those tags under its control are analyzed, as already described.

Routine Nest continued. At the end of the nest, all those tags added because of mixed relcons and counters are processed through the analysis routines like any other tag, and the results, in the same format as the Tagtag table are entered in Drumtag, (referred to as Adtag in Block 5), behind reset tag entries. The names of these added tags are catalogued in table Name. Also, those Adtag entries made in the Reset routine because a counter was required for the reset, and such a counter had not been found, can now be dealt with by the RSR routine as described under Reset routine.

Finally, if an end test for any DO in the nest is found to be unnecessary, the end test tag name is erased from the dotag.

When all the DO-nests have been processed, control passes to Block 3.

BLOCK 3. This block completes the subscript analysis by dealing with those subscript combinations not already analyzed in Block 2, namely, pure relative constants. A pure relcon is a subscript combination none of whose subscripts is under control of a DO. A relcon symbol can be defined in two different ways:

i.) By appearing on the left hand side of an arithmetic statement or in a read statement, both of which result in entries in the Forval table, giving internal formula number and symbol.

ii.) By a transfer out of a DO for the symbol.

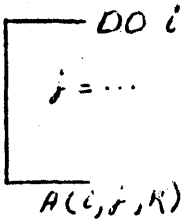
(Both these situations have been examined by Block 2, State B for 'Mixed Relcons', but that case was considerably more complicated by the existence of DO-controlled subscripts. However certain resemblances may be noted.)

VAL routine. This routine deals with method i.) above. A fortag entry is selected which has neither been analyzed in Block 2 (sign bit set), nor in this routine (bit one set).

Subroutine TABSER. The entire Forval table is now searched for ocurrences of any of the symbols of the fortag, and for each such forval, a TSXCOM table entry is made. This table enables Section 3 to compile a TSX to a subroutine which will compute the current index value for the tag. (The subroutine itself is compiled

by Block 4 of this section.) If the fortag has but one symbol with a coefficient of one, the TSXCOM entry is a special one, indicating that an LXD of the symbol into the tag suffices.

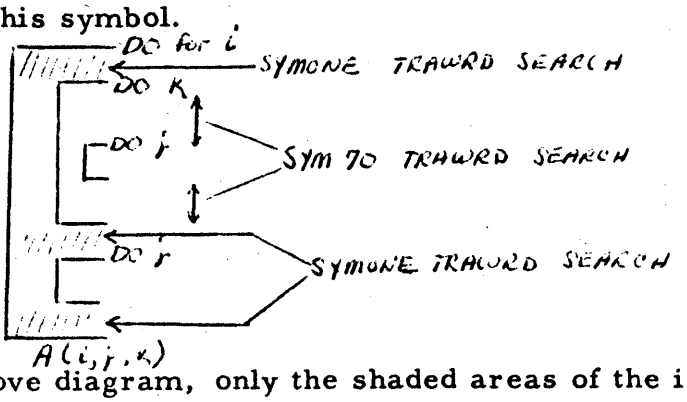
Subroutine INDO. There is however, one exception to the above rules in Tabser for dealing with forval entries. The forval is ignored if it occurs within a DO for one of the other fortag symbols. For example, in the diagram, i has strictly no value after the completion of the DO (not even the terminating value), and therefore it must be redefined before flow reaches the fortag. Thus it would be pointless to calculate the index value for (i, j, k) within the DO for i, when it must be recalculated later.



When all relevant Forvals in the TABSER search have been dealt with, equivalent fortags to the current one are marked with a bit 1. The next pure relcon is selected in the VAL routine, and the whole process is repeated until the fortag table has been exhausted.

TAGP routine. This next part of Block 3 deals with the second method of defining relcons. A new pass is made over the fortag table. A fortag is selected which has neither been analyzed in Block 2, nor in this part of Block 3, (minus sign set).

SYMONE routine. There are three levels of search for transfers out of DO's defining the fortag symbols. The routine from SYMONE to SYM70 deals with the first level search. A pass is made through Dotag looking for a DO for one of the fortag symbols. Having found such a DO (referred to hereafter as the SYMONE DO), the TRAWRD subroutine is called upon to search for transfers out of this DO which define only this symbol.



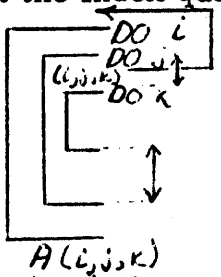
For instance in the above diagram, only the shaded areas of the i DO (i.e., those not contained within DO's for the other fortag symbols) are searched for transfers. If such transfers are found, subroutine PROCES is called upon to deal with them. This will be described later.

SYM70 - SYM170. If there are more symbols in the fortag, a second level search is commenced. A DO for one of the other fortag symbols is looked for within the Symone DO. Transfers out of areas of SYM70 DO defining only this symbol and perhaps the previous one are searched for by the TRAWRD routine. In the example, the areas marked in the DO for k would be searched. If such transfers exist, subroutine PROCES is called on as before.

SYM170-onwards. If there are three symbols in the fortag, there may be one or more DO's for the third symbol within the SYM70 DO. (In the example, the first

DO for j.) The routine looks for these, and if TRAWRD finds transfers defining the symbol, subroutine PROCES is called. After all SYM170 DO's have been dealt with, control is returned to the SYM70 routine. In the example, the other j loop would be found. After all SYM70 DO's have been processed, (calling on the SYM170 routine each time, to see if there is an inner DO), control is returned to the SYMONE routine, to find the next relevant DO.

Subroutine PROCES. This may be entered from any of the three search routines, SYMONE, SYM70, SYM170.
It is first established whether the fortag consists only of a left subscript symbol with a coefficient of one as this case is considered separately later.
The next task is to determine whether or not there exists an identical tag within the current DO, in the region of transfer, or an equivalent region. This would mean that the index quantity is already available on transfer. For instance in this diagram, where PROCES was called on after SYM70 had found a transfer shown, an identical (i, j, k) tag in the marked regions would satisfy the conditions, but an (i, j, k) tag in the k loop would have the wrong k value.



The first step is to scan the Name table*, to see if the tag occurs within the current DO. If so, the fact that its name was changed insures its occurring within the relevant areas. A Trasto entry is made to direct Section 3 to compile the storing of the new tag name contents in the old name cell, and control is then returned to the calling routine.

If the tag is not found in the Name table, it is searched for using either of two routines: TINFOR, which searches from beginning to end of a DO, or SPC, which spaces over inner DO's for other fortag symbols, the latter routine being used where such inner DO's could possibly exist. e. g. if PROCES is entered from the SYM70 search and the fortag has three subscripts (as in the previous example). If an identical tag is found, then it is in the proper index cell at the time of transfer, so control is returned to the calling routine. If not, a Trasto type 6 entry is made so that Section 3 will compile instructions to TSX to a subroutine which will compute the load value, and on return, to load the index cell with this value. Then an entry is made in table IRV, which is a list of all subroutines to be compiled by Block 4.

Case of one symbol with coefficient 1.

As with the case already discussed, the identical tag may occur within the DO. However, if it occurs as a test for the DO, in the form of a reset tag (bit 18 of word 7 in dotag indicates this), a Trasto entry type 5 must be made so that Section 3 will compile a TXI instruction for the tag with a decrement of N. This is because a reset tag is always initialized to zero instead of N at the beginning of a DO loop. If it does not occur specifically within the DO, it must equal the counter added by Block 2 (since a DO with a transfer out of range must have a counter). A Trasto entry is made so that the counter will be stored in the tag name cell.

* See write-up on Block 2, where Name table is compiled.

This completes the description of the PROCES routine.

When the end of dotag is reached in the SYM search routines, the current fortag and all identical ones are marked with a minus sign as having been processed, and the next fortag not already analyzed is selected. At the end of this pass over fortag, all subscript analysis is complete.

In the remaining part of Block 3, the tables IRV and TSXCOM are manipulated. The TSXCOM table is sorted into internal formula number order, as it is, in effect, a list of TSX instructions which must be merged into the CIT file by Section 3. Then the tag names from the TSXCOM entries are added onto table IRV (which may or may not already have entries made in the PROCES routine) and this is sorted into tag name order, duplicate entries being removed. Table IRV now consists of a list of all subroutines required for computing index values.

Block 3 is now complete.

BLOCK 4. Table IRV (sometimes known as BOB) provides a list of tag names whose index contents must be calculated from current symbol values in closed subroutines. This block compiles the instructions required to compute, for a subscript combination ($c_1 i$, $c_2 j$, $c_3 k$) with leading dimensions d_i and d_j, the quantity:

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

COMPIL routine. This routine controls the compiling, beginning with the instructions

```
A) tagname    CLA  6)+ 3
              STO  1)+ 3
```

which initialize erasable storage to 1. The symbolic location is the name given to the subroutine, and the number following the A) is the same as that following the C) in the name of the tag cell as it appears in the final listing. (It is the Tau reference number converted in a certain way by Section 6.)

The rest of the computation falls into three parts: one to compute $(c_1 i - 1)$ corresponding to the leftmost symbol, the next to compute $c_2 j d_i - d_i$ corresponding to the center symbol, and the last to compute $c_3 k d_i d_j - d_i d_j$ corresponding to the rightmost symbol. If any symbols are missing, the corresponding parts are omitted. All the different coding skeletons for the different situations are stored within the LXC subroutine, for instance if a coefficient is one, there is no need to compile a multiply instruction for it. Given the first instruction of a block, and the number required, the subroutine LXC outputs the appropriate coding for the part being considered.

Finally, Compil outputs the instruction

```
STO  C) tag name
```

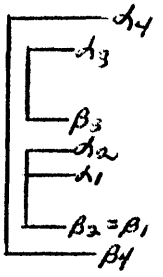
to store the result in the tag cell. Any constants required by the compilation are entered in the table of fixed point constants (Fixcon) as they are needed.

When all entries of Table IRV have been dealt with, Block 4 is complete.

BLOCK 5. This block compiles the necessary indexing instructions for the tags, using the results of subscript and flow analysis provided by Blocks 1 and 2, mainly in the Tagtag and Dotag tables.

There are two main cycles: the α cycle, which provides the loading and initial, izing instructions at the beginning of a DO, and the β cycle which compiles the incrementing, testing, and resetting instructions at the end of a DO.

MANIPULATOR routine. This routine controls the α and β cycles in the following way. It reads a nest of dotags, and then calls on subroutine Dogs to select the next α or β of a DO. The order in which α's and β's are selected is the backward order of internal formula numbers, and if DO's have equal β's then the β with the highest corresponding α takes precedence. For example, the processing order of this nest would be:



$\beta_4, \beta_1, \beta_2, \alpha_1, \alpha_2, \beta_3, \alpha_3, \alpha_4$

For any DO, clearly, the β cycle will be called upon first, thus any initializing instructions required for the instructions, because of variable parameters, will have been completely specified by the time the α cycle for the DO is reached.

RTX routine or β cycle. A special CIT entry of all one's is made to indicate the beginning of a new block of output CIT's, unless this β equals the previous one (e. g. β_1 and β_2 in the Manipulator routine examples), in which case both blocks are output as one. The location counter VCTR is initialized to the β internal formula number + 8 ready for use when a location symbol is required. (8 and subsequent multiples of 8 will be converted later into subsidiary parts of internal formula numbers 1, 2, 3, . . . etc. Resulting location symbols might be 10A2, 3A1 and so on.)

After narrowing down the search for tags in subroutine Scan, by arranging to ignore those out of range, a Tagtag entry is looked for which is modified by the current DO. Subroutine Tetg then establishes whether the tag is a test for any of the DO's controlling the tag subscripts. The β cycle compiling is divided into three parts: the TXI pass for incrementing the tag and updating end test decrements where necessary, the TXL pass, and the TIX pass for resetting the tag to its initial value ready for re-entering the DO.

TXI Pass RTX. There are six different situations which may occur, resulting in six different coding blocks - A through F. The appropriate block for the selected tag within the current DO is chosen in the following manner. Subroutine Pres forms a code word from the Tagtag information, consisting of

group no.	posind	T_L	C_{LC}	T_C	C_{CR}
X X X	X X	X	X	X	X

where the Group (1-6) is a code referring to the relative ordering of subscripts within the tag. (See Block 2 write up for full details) Posind is 3 if the current subscript is the leftmost; 2 if the current subscript is the center; 1 if the current subscript is the right.

T_L means that the tag is a test for the DO controlling the left subscript and T_C for the center subscript.
C_{LC} means that there is left to center carry, and C_{CR} means that there is center to right carry.
Comparison of this code number with a table of codes and associated block numbers establishes the TXI block for this tag in the current DO.

Block F: This means that no TXI instructions are needed, and the β state proceeds with the next tag. The reason is that there is carry for the tag, i. e., this DO is incorporated within the DO for the next inner subscript, whose TXI block will provide all the incrementing that is required.

Block A: This is the next simplest TXI block, consisting only of
TXI * + 1, tag name, decrement = $N_3 g$
where g = c_1 for leftmost subscript
$c_2 d_1$ for center subscript
$c_3 d_1 d_2$ for rightmost subscript
c and d being the coefficients and dimensions obtained from the Tau table. If there are duplicates, the $N_3 g$ is calculated for those symbols too and added to the decrement.

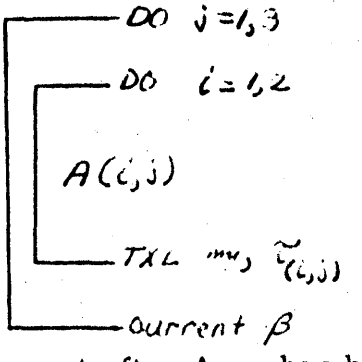
If, however, the decrement cannot be calculated because N_3 is variable, the TXI instruction is compiled with a zero decrement, together with a location symbol (using VCTR described earlier). An entry is then made in the Appended Tagtag table, a subsidiary of Tagtag itself, giving this location so that the α state can compile the necessary initializing instrustions.

Finally, for both variable and constant situations, if the tag is a simple counter, and the Dotag table indicates that the DO symbol occurs on the right hand side of an arithmetic statement or in an output statement, then the instruction

SXD Symbol , Tag name

is compiled to update the symbol itself.

Block B: This occurs when the DO for the leftmost subscript lies within that for the center (currentDO) subscript, and the tag is a test for the left subscript DO. Each time the current center subscript is incremented, the test for the inner DO must be updated. A simple 2-dimensional example will perhaps help to explain the position. In this DO configuration, the order in which members of the array A are to be selected is $A_{1,1}$ $A_{2,1}$ $A_{1,2}$ $A_{2,3}$ $A_{1,3}$ $A_{2,3}$. The first time the DO for i is satisfied, the TXL test must have 2 in its decrement, so as to drop out of the loop after $A_{2,1}$ is selected. However, on the next DO for i cycle, when j has been incremented, the decrement must be 4 so as to drop out after $A_{2,2}$ has been selected. Clearly then, the TXI block in the DO for j must contain instructions to step up the i loop TXL decrement.



The Block B instructions are:*

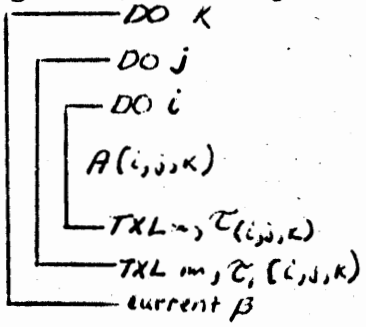
$$\begin{array}{lll} \text{TXI} & *+1, \tau_j, & \{[X]N_3g\}\ \text{left} + \{N_3g\}\ \text{center} - 1 \\ \text{SXD} & & \\ \text{TIX} & *+1, \tau_j, & \{[X]\ N_3g\}\ \text{left} - 1 \end{array}$$

(Note: the tag has already been reset by the amount through which it has been incremented in the inner loop) where $\{[X]\}N_3g$ left is the total amount through which the inner DO subscript increases, $\{N_3g\}$ center is the usual increment for a tag, as given for Block A. The SXD instruction is given the next abailable location number, and this location is saved in the Dotag of the left subscript, so that the address of the modified TXL may be inserted later. (Section3 does this.)

If the N's of the left subscript (inner) DO or N_3 of the current DO are variable, the above decrements cannot be computed. The instructions are compiled without the decrements, and the TXI instruction is given a location symbol, which is entered in the Appended Tagtag table so that the α state can compile initializing instructions.

Blocks C and D are very similar to Block B. In Block C, the current symbol is the rightmost, and the TXI block must modify the leftmost subscript test. In Block D, the current subscript may be either the leftmost or the rightmost, and in both cases the block must modify the center subscript test. Duplicate symbols in these two blocks may cause an adjustment to the decrements.

Block E: This is the most complicated and occurs when the current DO is for the right hand subscript and the relative configuration is as shown in the diagram, and the tag is a test for both the inner DO's. Both these TXL's must be updated by the current TXI block.



The Block E instructions are of the form:

$$\begin{array}{lll} \text{TXI} & *+1, \tau, & \{[X]\ N_3g\}\ \text{center} + \{N_3g\}\ \text{right} - 1 \\ \text{SXD} & \sim, \tau & \\ \text{TIX} & *+1, \tau_j & \{[X]\ N_3g\}\ \text{center} - 1 \\ \text{TXI} & *+1, \tau_j, & \{[X]\ N_3g\}\ \text{left} - 1 \\ \text{SXD} & \sim, \tau & \\ \text{TIX} & *+1, \tau, & \{[X]\ N_3g\}\ \text{left} - 1 \end{array}$$

As before, the SXD instructions are given symbolic locations, which are saved in the corresponding Dotags. If the N's for the DO's are variable, the instructions are compiled without decrements, and entries are made in Appended Tagtag so that the decrements can be initialized.

* $[X]$ is the integral part of $\dfrac{N_2 - N_1 + N_3}{N_3}$

RTX160 For all block numbers, if the tag is the best test for the current DO, its index is saved for the TXL pass later. Control now returns to select the next tag modified by the current DO and the process is repeated.

RTX180 After the Tagtags have been processed, the Adtag table of tags added by Block 2 is searched for those modified by the current DO. Instructions are computed as for ordinary Tagtags, except that block numbers can only be either A or F, since added tags are never used as tests for DO's. Corresponding to the Appended Tagtag table, there is an Appended Adtag table, for the variable N situations where decrements cannot be compiled. When all relevant Adtags have been dealt with, the TXI pass is complete.

TXL Pass RTX 200 If the current DO does not require a test, then neither the TXL nor the TIX pass is necessary. If a test is needed, the best tag is selected, and a TXL instruction for the tag is compiled as follows, whith the next free location symbol attached. If the N's of the DO are variable, a zero decrement is compiled, and a variable indicator bit is set in Dotag to inform the α state that initialization is required. If the N's are constant, then the decrement $N_2 g + \delta_1 + \delta_2$ is computed, where $g = c$ for left symbol, $c_2 d_1$ for center symbol, and $c_3 d_1 d_2$ for right symbol, and

$$\delta_1 = \left\{ c_2 N_1 d_1 - d_1 \right\}_{\text{center}} \quad \text{if the center subscript lies to the right of the current subscript}$$

$$= 0 \text{ otherwise}$$

and

$$\delta_2 = \left\{ c_3 N_1 d_1 d_2 - d_1 d_2 \right\}_{\text{right}} \quad \text{if the right subscript lies to the right of the current one}$$

$$= 0 \text{ otherwise}$$

TIX Pass RTX 222 A pass is now made over the Tagtag table, looking for tags modified by the current DO, in the same way as the TXI pass. Each of these tags must be reset by the amount through which they were incremented in the loop, ready for the next entry of the loop, unless i.) there is carry, which is not prevented by the tag's being a test for the DO, and ii.) this DO is the outermost of the controlling DO's - in which case the index will be reloaded before the next entry. In case ii.), the instruction DED with this tag, is compiled to inform Section 5 that the index is no longer needed. If the N's of the DO are constant, the instruction

$$\text{TIX} \quad * + 1, \text{ tag}, [X] \, N_3 g$$

is compiled, where $[X]$ = the integral part of $\dfrac{N_2 - N_1 + N_3}{N_3}$ and g is c, or $c_2 d_1$, or $c_3 d_1 d_2$ as explained earlier. If there are duplicate symbols, the decrement is adjusted accordingly. If this is the first TIX instruction after the TXL, then the instruction is given the next symbolic location available.

If the N's of the DO are variable, the instruction is compiled with a zero decrement and a location symbol, the latter being entered in the Appended Tagtag table.

When all relevant tagtags have been dealt with, the same is done for the Adtag table, entries being made in Appended Adtag when required.

At the end of the TIX pass the β state or RTX pass is complete. Control is now returned to the Manipulator routine to select the next α or β of a DO for processing.

α cycle - AC routine. The VCTR location symbol is initialized to the internal formula number of the α of the DO, ready for the α cycle compiled instructions, and an all ones entry is made in CIT to signify the beginning of a new block of output entries. Also, an entry is made in the SXDTX table for this DO, giving the α and β internal formula numbers and the relative location of the corresponding TXL instruction. Section 3 uses this table to fill in the addresses of the SXD instructions, compiled in the TXI pass of the β cycle, as the TXL locations were not known at the time these were compiled.

AC010 The routine looks for a tagtag entry for which the current DO is the outermost controlling DO. The next task is to compile instructions to compute and load the index for the given tag, as when this DO is reached, at object program time, all variables concerned have been defined.

If the tag is a reset added tag, all that need be compiled is

LXD 6) + 2, τ

which loads the index with zero. (Actually, this situation does not arise until the added tag pass is made, after all tagtags have been dealt with.) If the tag is a simple case of only DO-controlled subscripts, and a constant N, then the quantity:

(i) $\{c_1N_1 - 1\}$ left + $\{c_2N_1d_1 - d_1\}$ center + $\{c_3N_1d_1d_2 - d_1d_2\}$ right

is computed and entered in the Fixcon table. (The N's in the expression apply to the DO controlling the subscript indicated.) The instruction LXD 2)+n, τ is compiled, where n is the relative location of the constant in Fixcon. A location symbol derived from VCTR is given to the first instruction of the block.

If, however, the load value cannot be computed at Fortran execution time, the instructions must be compiled to complete expression (i) at object time. If any subscript symbol is a relcon, then the corresponding N is replaced by the symbol itself. If c_1, c_2, c_3, $\{N_1\}$ center $\{N_1\}$ right are all one, then the expression reduces to $\{N_1\}$ left (or S if the left symbol is a relcon), so a simple LXD of this symbol is all that is necessary. The instructions for computing the three parts of the expression are compiled in three separate routines AC049B, AC064, AC080. If any of the parts are constant, or if any coefficients are one, advantage is taken of the situation. Any constants, such as d_1, d_1d_2 that are found to be required are entered in Fixcon and their symbolic addresses are of the form 2) + n.

AC 100+2 The next part of the α cycle is concerned with the initialization of end test decrements. In front of the outermost DO loop, initializing instructions for inner DO tests may be necessary for either or both of two reasons; firstly the N's may be variable, and secondly, while the associated DO's are being carried

out, the decrement may be updated by SXD instructions (see TXI pass for Blocks B, C, D, E). If only for the latter reason, then the initial value can be computed in this routine. The quantity is $N_2 g + \delta_1 + \delta_2$

where $\delta_1 = \{c_2 N_1 d_1 - d_1\}_{center}$ if the center subscript lies to the right TXL subscript

$= 0$ otherwise

$\delta_2 = c_3 N_1 d_1 d_2 - d_1 d_2$ right if the right subscript lies to the right of the TXL subscript

$= 0$ otherwise

(see the TXL pass in the B state.)
The constant is entered in Fixcon, and the instructions

```
CLA   2) + n
STD   (TXL location)
```

are all that are required. If the TXL decrement is genuinely variable, then instructions to compute the above quantity must be compiled. These are made as simple as possible, and if for instance, N_2 is constant, then $N_2 g$ is computed in this routine. Use is made of routines for calculating the index load value, in the earlier part of the α state.

Having dealt with test decrements corresponding to all 3 symbols, then if the tag is a stored counter, the symbol must be initialized to the value N_1, so the instructions

```
PXD 0,
STO Symbol
```

are compiled.

AC 165 The appended tagtag table is now searched for an entry for the current tag indicating that there are variable TXI or TIX decrements. If not the α state is complete for this particular tag. Otherwise, each symbol of the tag is taken in turn, as in the AC100+2 routine. If the TXI block decrements associated with the corresponding DO for a symbol, are variable, then, according to the TXI block number, a routine is called upon to compile initializing instructions. It is not possible in this discussion to give full details of the compilations in the different circumstances, since they are so many and varied.

AC 200 In the same way, each symbol of the tag is taken in turn once again, this time to see whether the associated TIX instructions have variable decrements. If so, instructions are compiled using some of the routines already used in the TXI part, optimizing of course, wherever possible.

Compiling is now complete for this tag, and control is returned to the beginning of the α state, to find the next relevant tag.

When the tagtag has been exhausted, the Adtag table is processed in the same way. If no instructions have been compiled in these routines for this particular α, then

the instruction

α BSS 0

is compiled so that the internal formula number will be given a location. Control is now returned to the Manipulator routine, to obtain the next α or β for processing.

MANIPULATOR continued When instructions have been compiled for all αs and βs of a DO-nest, a zero CIT entry is made to indicate this to Block 6. After all the nests have been processed, control passes to Block 6.

BLOCK 6. The order in which Block 5 compiles DO instructions for a nest is the backward sequence of αs and βs of the nest, although within each α and β block, the instructions are in the natural order. The α and β blocks of CIT's must therefore be inverted, so that Section 3 can merge the DO file with the COMPAIL file, output by Section 1. The beginning of each block is marked by an all one's CIT entry, and, after reading a nest of CIT's (the end of a nest being marked by zeros), Block 6 searches from the end of the nest until an all one's fence is found. The instructions just scanned are output as the DO file, and would correspond to the first α of the nest. Block 6 then looks for another fence, and so on, until the whole nest is output. When the DO file is complete, control passes to Section 3.

TABLES CREATED BY SECTION TWO (Alphabetical List)

ADTAG (Block 2) - Memory table.

There are two different types of entries:

i.) 'Normal'

Relevant DOTAG index (called X1)	Current tag name (TS)
TL1 (level number)	TL2 (level number)

This entry is made by Subr. RESET, and contains information about the possible use of a counter as a RESET tag.

ii.) 'Not Normal'

Relevant DOTAG Index	FORTAG TAU reference
	New tag name for Added tag

These are details of a tag created by Block 2, in State B.

ADTAG (Block 5) - Memory table.

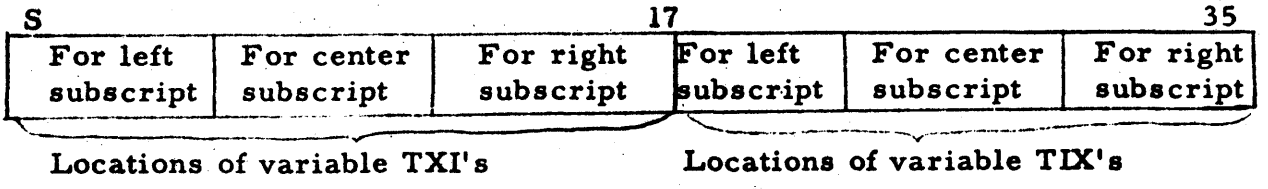
This is called DRMTAG by Block 2, and has the same format as TAGTAG. (see TAGTAG)

APPENDED ADTAG (Block 5) - Memory table.

This bears the same relation to ADTAG that APPENDED TAGTAG does to TAGTAG. (see APPENDED TAGTAG)

APPENDED TAGTAG (Block 5) - Memory table.

There is an APPENDED TAGTAG entry corresponding to each TAGTAG entry. While processing the TAGTAG in the B state of Block 5, locations of variable TXI and TIX decrements are entered in the table.



BOB see IRV

CHANGETAG (Block 2) - Memory table.

Beginning of Region	End of Region
FORTAG TAU reference (original tag name)	New tag name

The table is used by Section Three to change the tag names of all identical FORTAGS within the given region.

DOTAG Tape 2 Table

This table is derived from table TDO. Information concerning each DO is added throughout Blocks 1 and 2.

word 1)	α (Internal)	I	II	III	β (Internal)
word 2)	I				
word 3)	N1				
word 4)	N2				
word 5)	N3				
word 6)	Level # of this DO				$X = \left[\frac{N2 - N1 + N3}{N3}\right]$ N3
word 7)	N				Level of definition of variable N1
word 8)	Contains bits, the rightmost of which determines the highest level of transfer from this DO.				Level of definition of variable N1
word 9)	S → 5 Erasable	Name of tag which will be used for test.			Level of definition of variable N3

The I, II, and III in the tag of word 1) indicate whether N1, N2, or N3 are variable respectively.

Words 2), 3), 4) and 5) are left-adjusted BCD, if variable. Otherwise they are binary and in the address.

Word 6) Sign Bit: This DO requires an unstored counter.
Bit 1: This DO requires a stored counter.
Bit 2: X is not computable.
Bit 18: I is associated with a Relcon and this DO does not have Type I carry.
Bit 19: Type II carry.
Bit 20: Type I carry.

Word 7) Sign Bit: Transfer in extended range of this DO.
Bit 1: Transfer in immediate range of this DO.
Bit 2: I is formed above as a subscript in this DO.
If N $\leq$ level of γ(I), then there is no carry.
Bit 18: There is a counter to be used as a reset.
Bit 19: Nullifies effect of Bit 18.

Word 9) Bit 19: There are no tags except mixed RELCONS in this DO.
Bit 20: This DO's TXL has a variable decrement.

DRMTAG (Block 2) - Memory table.

DRMTAG entries are the results of analysis of added tags in Block 2. (This table

is called ADTAG in Block 5.) The format is the same as TAGTAG format.

IRV (or BOB) (Block 3) - Memory table.

This table consists of the second words of TSXCOM Type 2 entries, (see TSXCOM), in numerical order, with duplicates omitted. In effect, it is a list of all subscript combinations whose index values must be computed in subroutines.

NAME (Block 2) - Memory table.

DOTAG ∝	FORTAG TAU reference (old tag name)
	New tag name

This table is a list of all tags whose names have been changed from the TAU reference number to a new name.

RETAB (Block 2) - Memory table.

Entries are made in Subroutine Reset, giving details of each reset tag created to avoid duplicate reset tags.

The first word is

prefix Rebits	Current DOTAG Index	Current FORTAG TAU reference

The second and third words depend on the subscript (s) to be reset.

Left

c_1	0
0	0

c

0	c_2
d_1	0

R

0	c_3
d_1	d_2

L, c

c_1	c_2
d_1	0

L, R

c_1	c_3
d_1	d_2

c, R

c_2	c_3
d_1	d_2

TAGTAG (Block 2) Tape 4

This table contains the results of analyzing the tags in Block 2. (counters created by Block 2 are also included)

The format for both DRMTAG and TAGTAG is as follows:

Innermost controlling DO	First subscript DOTAG index
Second subscript DOTAG index	Third subscript DOTAG index
FORTAG TAU reference (original tag name) or 0, if reset or added tag	Current tag name (if reset, 'Rebits' are in 22 - 24)

Bits in this fourth word are as follows:

S bit from CARWRD, if tag is left subscript only with coefficient
8 if left subscript test
9 if center subscript test } Test indicators set in Block 5.
10 if right subscript test
11 if pos^{ns} 1 & 2, Type 1 carry
12 if pos^{ns} 1 & 2, Type 2 carry } carry bits from CARWRD
13 if pos^{ns} 2 & 3, Type 1 carry
14 if pos^{ns} 2 & 3, Type 2 carry
15 - 17 Group Number
21 - 23 Indicates which coefficient > 1
24 - 26 Duplicates
30 - 32 Relcons and DO-relcons (Dorcs)
33 - 35 Dupes and Dorcs

TRALEV (Block 1) Tape 4

Entries are of variable word length.

First word	- TIFGO α	
For each address of the TIFGO	TIFGO address	Level of address

TRASTO (Blocks 2 and 3) - Memory table.

There are 6 types of entries, all of which specify internal formula number intervals α to β level number intervals and other information. Section Three uses these entries to compile 6 types of indexing instructions in conjunction with transfer control statements, whose internal formula numbers fall within the specified formula number interval and whose transfer addresses lie within the specified levels.

The different types of entries and the resulting compilations are given:

Type I

α	β
L_1	L_2
T_1	T_2

Where level $L_1 > L_2 > 0$

T_1 and T_2 are tag names (contents of Bits 18 - 20 of word 1 may be non-zero)

Compilation:

SXD C) T_2 , T_1

LXP C) T_2 , T_2

where C)T_2 is the tag cell of T_2

Type II

α	β
BCD fixed point variable	
L_2	T_1

$L_2 > 0$

Compilation: PXD 0, T_1
STO (symbol)

Type III

$-\alpha$	β
L_1	L_2
T	n

n is a binary intefer

Compilation: TXI *+1, T, - n

Type IV

α	β
L_1	L_2
$-T_1$	T_2

$L_2 > L_1 > 0$

Compilation: SXD *+1, T_1
TIX *+1, T_2, 0

T_1 refers to a reset tag, T_2 to the tag to be reset.

Type V

$-\alpha$	β
L_1	L_2
- T	n

n is a binary integer

Compilation: TXI *+1, T, n

Type VI

α	β
-0	L_1
-0	T

Compilation:

```
SXD    6)+5, 4
TSX    A)T, 4
LXP    C)T, T
LXD    6)+5, 4
```

Where A)T is the name of the subroutine computing the index value of T, and C)T is the tag cell.

TSXCOM

There are 2 types of entries.

Type I

FORVAL α	Tag name
FORVAL symbol	

This indicates to Section Three that LXD symbol, T must be compiled following the FORVAL.

Type II

FORVAL α	
	Tag name

This indicates that a TSX to a subroutine, which will compute the index value, must be inserted after the FORVAL.

VI

SECTION THREE

The MERGE has the primary function that its name implies. That is, it must merge or collate the different files of compiled instructions (CIT's) that are available to it. There is, however, an important additional function which the MERGE serves. This is the creation of an additional file of instructions. This additional file is based on information gathered by Section Two and passed on to the MERGE in the form of tables.

The MERGE, therefore, falls naturally into three main Divisions: Merge I merges the two files passed on to it by Sections One and Two respectively; MERGE II creates the additional file of instructions; MERGE III merges the two files of instructions now existent. The two files of instructions compiled by Sections One and Two are the Compail and the Compdo files. The file created in MERGE II is called the Tifgo file. The results of the MERGE I file is called simply the Firstfile. MERGE III, of course, merges the Firstfile with the Tifgo file.

At the end of MERGE III, then, a single file of CIT's exists and is passed on to Section Four. This single file is essentially the completed compiled program. That is, it contains all the instructions necessary for the translation of the source program, on the assumption that the object machine contains as many index registers as there are symbolic tags in the single file of instructions. Therefore, the remainder of the FORTRAN Executive Program is devoted to two main tasks:

a.) Substituting absolute index registers for the symbolic index registers assumed up to this point.

b.) Inserting the load and save index instructions required by the limited number of absolute tags.

It is important, further, to point out that the additional file of instructions created in MERGE II does not result from any further analysis of the FORTRAN Source Program as such. Rather, it is compiled from tables which are themselves the result of such analyses. The MERGE, therefore, does no analytical work of its own; it simply stands at the crucial crossing point between the first part of the compiler which does the basic analysis and the latter part which handles the index register problem and the assembling problems.

Partially as a result of this critical position of the MERGE in the over-all flow of the FORTRAN Compiler, the MERGE is given certain additional subsidiary tasks to perform as it does its primary merging tasks. In this description, these subsidiary tasks will be listed and described in their appropriate place. It is only worth noting here that many of these tasks could theoretically have been done earlier; that they were not done earlier and were, instead, left to

the MERGE is, to a great extent, a matter of convenience for the earlier analyses. The fact that the MERGE must make several complete passes over all the CIT's makes it simple for the MERGE to make the insertions required by these subsidiary tasks.

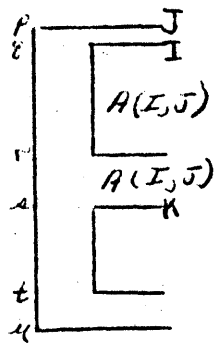
MERGE I.

A. The merge of the Compail and the Compdo files occurs by simple numerical collation. The two files are on two separate tapes, and, of course, exist in 100 word records, that is, 25 instructions per record. The first word of each instruction contains the internal formula number. As was pointed out earlier, the internal formula number is physically present only for the first instruction of the translation belonging to any unique source statement except where a source program statement gives rise to more than one internal formula number. Stated another way, the first instruction of such a block will have the internal formula number in the first word whereas all the other succeeding instructions of that block will have zero in the first word. Therefore, the instructions exist in blocks, each of which is headed by an instruction with an explicitly stated internal formula number.

Furthermore, this internal formula number may have an increment part; that is, a number in the address field as well as the decrement field. Keeping this in mind, it is easy to visualize the manner in which MERGE I goes about interrelating the instructions from the separate Compail and Compdo files. The internal formula numbers are compared and the block headed by the smaller number is compiled first. Whenever the numbers are the same, the Compail block is compiled ahead of the Compdo block.

B. Additional MERGE I functions.

1. CHTAG table. As a result of the Section Two analysis, it may be found that certain tag (subscript combination) names must be changed. To indicate the name change, an unedited change, this table (see illustration) specifies, in addition to the name of the tag and the new name to which it is to be changed, the range of the problem over which this tag is to take the new name. This range is indicated by two internal formula numbers. An example of the significance of the range is illustrated here:



This problem will give rise to three unedited change tag table entries, for the subscript combination (I, J). These three entries will cover the ranges respectively of p and q, r and s, and t and u.

If the table were unedited, it would be necessary for the MERGE to scan and test every tag field of every CIT appearing within the given ranges. In order to avoid this extended testing, the table is edited; the editing enables the exact location of the tags requiring changed names to be localized from a range of several statements to a single statement.

Therefore, the first task that MERGE I performs is the editing of the change tag table. This editing occurs with the aid of the FORTAG table which contains an association of tag names with specific internal formula numbers. The edited change tag tables, therefore, are the same as the unedited table with the exception that a range of statement has been reduced down to a single statement number. While scanning the CIT's during the merging process a test is made on statement numbers to see if they match the number in the edited change tag table. If they do, the new names are inserted in the tag field.

2. The SXDTX table. Section Two may find as a result of subscript combinations within DO nests that it is necessary to change a decrement value of a TXL instruction ending a DO. This will usually be in inner DO's where the index is the same as the leftmost subscript symbol in the subscript combination. Of course, it assumes that this subscript combination is used for testing the end of the inner DO. At the time that Section Two is compiling the SXD instruction which inserts the new test value in the TXL instruction it does not know what the internal formula number designation for the TXL instruction will be. Therefore, it makes a table indicating this internal formula number allowing the MERGE to complete, by filling in the address, the SXD instruction.

3. Open Subroutines. Whenever an open subroutine reference is encountered, during the compilation of the arithmetic instructions, a CIT is compiled which is merely a signal to the MERGE. This signal tells the MERGE not merely that an open subroutine is necessary at this point, but it also designates which open subroutine is giving information about where the argument is to be found and whether the output of the subroutine is to be left in the accumulator or MQ. Encountering this signal CIT or CIT's, the MERGE inserts the appropriate open subroutine. The designations referring to input arguments and output results, of course, pertain to the problem of arithmetic instruction linkage. With the completion of these functions the MERGE has produced a single file of instructions called the Firstfile.

MERGE II.

MERGE II of Section Three does not do any merging; it produces a new file of instructions. The tables used in producing this Tifgo file of instructions are the TIFGO and TRAD tables from Section One and TRALEV and TRASTO tables from Section Two.

The need for the Tifgo file of instructions arises in the following way. The main body of computing and indexing instructions, included in the Compdo file, are associated with the beginning and end of DO's. That is, the internal formula numbers of their CIT's have the internal formula numbers belonging to the DO statements themselves and the final statement in the range of these DO statements. However, indexing instructions belonging to DO nests may be associated with statements within the range of DO statements. The entire Section Two mechanism is set up to do compiling of the beginning and end of DO indexing instructions. It does the analysis necessary for the indexing instructions required within the range of DO's but does not compile the instructions. Instead it prepares the two tables TRALEV and TRASTO, which are a summary of this information.

All of these types of indexing instructions arise from the fact that transfers occur within DO's, specifically transfers going out of the range of a DO. In considering this problem, an entire DO nest, involving possibly many levels of DO's as well as many DO's on any given level, must be considered. Consequently, a transfer out of a DO within any DO nest may be a transfer entirely outside the nest (that is, to level zero) or to another DO within the nest (that is from level 1 to level n). Specific Tifgo instructions are caused by the fact that some indexing must occur before a transfer out of the DO is made, provided that the configuration of DO nest within the nest, subscript combinations within the nest, and the uses of DO indices are such that indexing instructions are required.

There are, in fact, six different sets of indexing instructions which may precede any individual transfer. These six sets account for six different types of TRASTO entries, as illustrated in V. Either one or a combination of these sets may be required before any transfer. The TRASTO tables are numbered: this means that the instructions corresponding to each type of TRASTO entry must occur in the sequence indicated by the number. In setting up the TRASTO table entries, Section Two determines the relevant facts with respect to both the location of the transfer instruction itself and the transfer addresses of any single source program instruction.

No detailed explanation will be given here of the specific reason for each of the six types of TRASTO entries. Briefly, however, we can note that three of them arise from the class of problems described in Section Two as "reset" problems. It has already been noted that a reset must occur before starting the recycling of an inner DO. By a simple extension of this, the same kind of resetting must occur when a transfer is made from an inner DO

to a point within the DO nest. This is, simply, so that the inner DO index will have been reset on the next repetition of the inner DO. Another TRASTO entry is required when the area of the transfer address has an arithmetic statement with the inner DO index symbol on the righthand side or, correspondingly, has that symbol in an output list. Another is required to effect a transfer, within DO's, to one of the A) subroutines. The final one is required to effect the proper saving of an index value or subscript combination tag when a transfer defines the tag value. This covers the following case:

Let a DO on I be within a DO on J. Let the subscript combination (I, J) appear in the inner DO and the outer DO. Further, let there be a transfer from the inner DO to the arithmetic statement containing the subscript combination in the outer DO. Since these two subscript combinations receive different names by the method of the Section Two analysis, provision must be made to shift the index value from one name to the other.

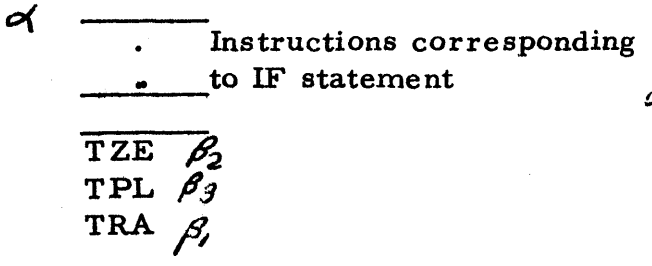
In setting up TRASTO tables, Section Two, much as in the unedited change tag table, designates a range within which the relevant Tifgo (IF or GO TO) instruction occurs. Furthermore, a range is left for the level of the particular transfer address in five of the six TRASTO cases. It is these facts, along with the necessary ordering of TRASTO instruction sets, which create whatever compensating complexity exists in the MERGE II compiling section.

The MERGE II analysis proceeds in this general manner. It uses the TIFGO table as its guide. In this connection, it must be remembered that the TRAD table is simply an extension of the TIFGO table. It simply supplements those TIFGO entries arising from computed GO TO and ASSIGN GO TO statements. When it comes across a TIFGO entry it checks to see if it is also in the TRALEV table. If it is not, there is no further concern for possible TRASTO instructions and the direct transfer addresses are compiled into the relevant transfer instruction. By direct, we mean here the number given in the source program, translated into its internal formula number. When the transfer or TIFGO entry is in TRALEV, there arises the further possibility of TRASTO entries for any of its addresses. The TRALEV table, it must be remembered, lists the levels of each of the transfer addresses. Consequently, a search is made through the TRASTO tables, first, for those entries indicating the appropriate range of the TIFGO internal formula numbers, then, if that is found, a search to see if the TRASTO entry designates the level range corresponding to the level indicated by the TRALEV entry. If these conditions are met, then MERGE II compiles the indexing instructions corresponding to this type of TRASTO.

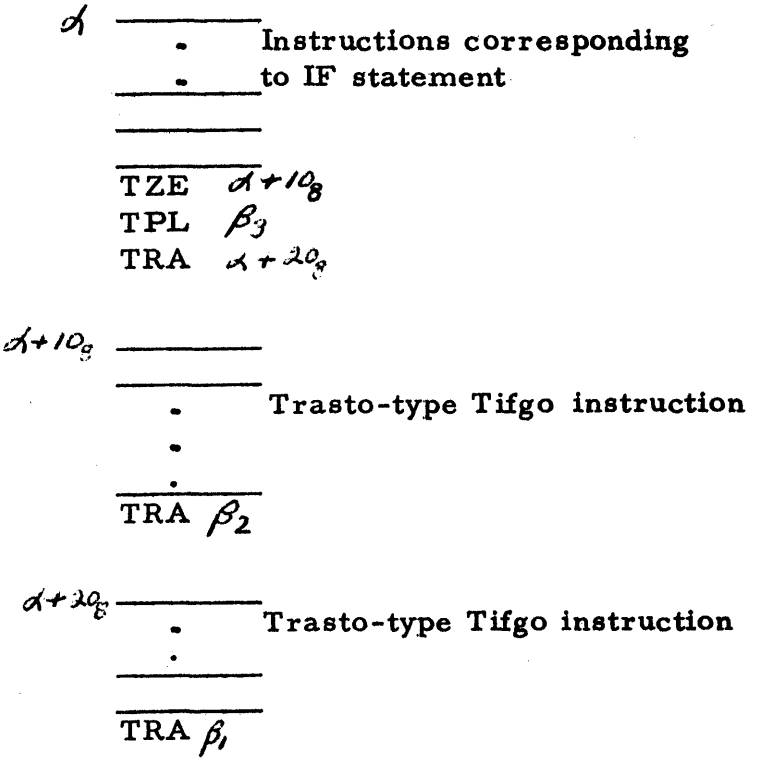
The only complexity that arises here is with the handling of the internal formula numbers. The following illustrations cover both the "no TRALEV entry" and the "TRALEV entry found" cases.

IF (...) β_1 , β_2 , β_3

No Tralev entry



Tralev entry with Trasto entries agreeing to formula numbers β_1 and β_2 (first two branches only)



The most important subsidiary task performed by this part of the MERGE is the putting together of the ASSIGN CONSTANT table. This comes about as a byproduct of the scan of the TIFGO table, which contains the ASSIGN GO TO entries. The ASSIGN CONSTANT table appears subsequently as the 5) block, containing the transfer instructions to each of the possible ASSIGN GO TO addresses.

With the completion of MERGE II, a new file of instructions exists, containing the computing and indexing instructions arising from transfers within DO's.

MERGE III.

The task of MERGE III is comparatively simple. It simply does a direct merge on the Firstfile and the Tifgo file. Here too, the principles of the numerical collation apply. It might be noted that in some cases, MERGE II will simply have supplied transfer addresses for instructions which were partially compiled in Section One. That is, the Section One instructions will be complete except for addresses. In this case, the two instructions are brought together by "oring" one over the other.

The primary subsidiary task here is the insertion of the instructions arising from the TSXCOM table. This too, provides the instruction resulting from the definition of a relative constant and there are two types of TSXCOM entries: One providing the transfer to the A) subroutine, the other, providing for a direct load into an index register from the relative constant cell.

All that remains for the MERGE to do is follow the main file of instructions with the two secondary files of instructions compiled by Section One and Section Two. These are the arithmetic statement function instructions and the A) subroutines, respectively.

At the end of the MERGE all instructions resulting from an analysis of touring the source program are complete, except for the existence of symbolic tags, rather than at the absolute tags. This provides the main task of Sections Four and Five.

SECTION FOUR

PART 1

The first task of this part is to divide the object program into basic blocks, a basic block being a stretch of program with but one entry point, and one exit point. In order to do this, a pass is made over CIT looking for transfers, tests and skip type instructions. Transfer and conditional transfer addresses, and the locations of instructions following skip type instructions or TXLs (end tests of Dos), are all entered in the BBLIST table once only, in algebraic order, by means of a binary search technique. The assigned Go To instructions are ignored for the moment.

During this pass, when a TXL is encountered, both its location and address are entered in the DOLIST table, thus providing a list of the beginning and end locations of all Dos, in end location order.

Routine Assign

A new table, TIFRD, is now formed from the Assign and Assigned Go To entries in the TIFGO table, together with the associated entries in TRAD. (Tifgo entries are of a fixed word length, and the Trad table was therefore created to accomodate all possible Assign Go To transfer addresses.) At the same time, all the transfer addresses are entered in BBLIST.

BBLIST now constitutes a list of the beginnings of all basic blocks in the program, in the order in which they occur. The basic block number which is referred to later, is the relative address of the particular basic block within the BBLIST table.

Subroutine Freq

There are two types of Frequency Statements in the Fortran language:

1) A type referring to conditional transfers, estimating the frequencies of the various branches taken.

2) A type estimating the loop counts of Dos with variable parameters.

The first type gives rise to a frequency table entry containing the internal formula number of the corresponding coding, followed by as many frequency estimates as branches. Given frequencies $(i_1, i_2, i_3, \ldots\ldots i_n)$ this routine modifies them to form quantities $\left(\frac{i_1}{\sum_1^n i}, \frac{i_1+i_2}{\sum_1^n i}, \frac{i_1+i_2+i_3}{\sum_1^n i}, \ldots\ldots \frac{\sum_1^n i}{\sum_1^n i}\right)$.
The significance of this will be explained later.

The second type gives rise to a 2 word entry containing the internal formula number corresponding to the Do followed by the estimated loop count. (The length of this type of frequency entry - 2 words - distinguishes it from the first type, which is always longer.) These Do entries are transferred to the Dofret table, and the remaining frequencies are moved up to occupy the vacated positions.

Routine SORTDO

The table Dolist, created in the first pass and ordered on the ends of the Dos, by nature of the way it was built up, is now sorted into the order of beginnings of Dos. When these are equal, the Do with the largest terminating location takes precedence. The table is now compared with BBLIST, and the internal formula numbers in Dolist are replaced by basic block numbers.

Routine LOADDO

Dos are further processed in this routine. Each Dolist entry is matched with the corresponding Dotag table entry, and if the latter indicates a transfer out of the Do range, a tag of 7 indicator is set in Dolist. Such a Do is referred to in this Section as a 'Do with an if'. Also the loopcount $\frac{N_2 - N_1 + N_3}{N_3}$ is calculated and entered in Dolist, unless the parameters are variable; in the latter case, if a Dofret entry exists for the Do, the loopcount given there is used, otherwise the arbitrary loopcount 5 is used. If the current Do is a carry case, i. e. in the Docare table written by Section 2, the loopcount is multiplied by that of the previous Do.

PART 2

A second pass is now made over Cit, producing the three principal tables with which simulation is accomplished, namely BBTABL, SET and TRATABLE. There is one 1-word entry in BBTAL for each basic block in the program, but there may be several SET and TRATABLE entries corresponding to this one BBTABL entry. At the beginning of each basic block, the next available locations in SET and TRATABLE are entered in the BBTABLE, thus providing a key to information which will be accumulated during the pass, for the basic block.

The SET table is made up of information which may influence future flowpaths taken in the object program, and is formed in the following way. On reaching a sense light instruction, this entry is made:

Address of dummy light(s) affected.	0 or 1

A zero address would indicate that the light should be turned off, a 1 that it should be turned on. (Dummy lights only are maintained during the program simulation pass, in Part 3, not the actual machine lights.)

An entry is made if the current instruction is derived from an Assign statement, (i.e. if there is an 'assign' type TIFRD entry corresponding to the current location symbol.) The appropriate transfer location is obtained from the Tifrd entry, and thence, by examining BBLIST, the basic block number is derived.

Also, the symbol of the Assign, "N" for example, is entered in a new table NLIST. The SET entry takes the following form:

NLIST address	Successor BB number

There is a further type of Set entry which will be discussed later.

The remainder of the analysis during this pass is concerned with obtaining information about basic block endings. If the instruction following the current one begins a new basic block, (i.e. its location is in BBLIST), then the current instruction clearly ends one. Also, if the following instruction does not begin a basic block, then it may be that the current one is a skip type instruction or a conditional transfer, both of which constitute basic block endings.

The ending code of the basic block is now placed in the BBTABLE, which already contains the TRATABLE and SET addresses for this basic block. The different codes are given in this list.

0 The TXL of a Do with an if.

1 Sense light ending

2 Probability ending. This is entered for such BB endings the successor of which cannot be predicted. These might be sense switches, conditional transfers, Go to vectors.

3. Certainty ending. This is entered when the block ends with a transfer, or control always passes to the following basic block.

4. The TXL of a Do without an if.

5. Assigned Go To ending.

6. Stop ending.

The transfer addresses at the basic block ending give all possible successor basic blocks, and the associated block numbers are found by looking up these addresses in BBLIST. Subroutine ENTER places each one of these in the 2nd word of a 2-word TRATABLE entry, (there being one entry for each successor basic block), together with other relevant information pertaining to the current basic block ending, e. g. location symbol of current ending, number of branches etc. See appendix for Tratable details.

There is a further type of Set entry which was not discussed earlier. During the simulation pass in Part 3, 'Dos with ifs' must be simulated so as to obtain statistical counts of the flow paths taken. In order to do this, a loop count for the Do is maintained, which must be reset to its initial value each time the Do is re-entered. To this end Subroutine Enter, mentioned in the foregoing paragraph makes the following analysis. If the successor BB which has just been entered in Tratable, begins a Do with an if, and if the Do is being entered from outside its range, then arrangements are made to reset the loopcount. However, the table entries corresponding to the Do are not available, as the Do itself may not have been reached yet in this pass. Therefore, a new table FIXDO is built up and entries made as follows:

SET entry:

BB No. of end of Do*	loopcount

*This is in effect, a table reference to the Do entry in the BBtable.

FIXDO entry:

Set entry address

At the end of the pass, the information pertaining to the Dos has been entered in the tables. Each Fixdo entry is now taken in turn, and, from the corresponding Set table entry, the Do's BBtable entry is obtained. This as described earlier, contains a reference to the Dos TRATABLE entry (containing loopcounts etc.) and this last is now stored in the above Set entry to form:

Address of Tratable entry for Do	loopcount of Do

Table Fixdo is now redundant.

Routine FIXTST

One more preliminary must be dealt with before the simulation takes place, and that concerns the frequencies and probability BB endings. For each probability BB ending found in the BBtable, the corresponding Tratable entries are obtained. Relative frequencies for the different successor BBs must be either found in the frequency table (by matching formula numbers given for the frequency and for the Tratable entry), or formed now by assuming each branch equi-probable. As mentioned in Part 1 of this discussion, the frequency table entries, originally of the form $(i_1, i_2, i_3 \ldots\ldots i_n)$, are now $\left(\frac{i_1}{\sum_1^n i}, \frac{i_1+i_2}{\sum_1^n i}, \frac{i_1+i_2+i_3}{\sum_1^n i}, \ldots\ldots \frac{\sum_1^n i}{\sum_1^n i} = 1\right)$ and the equiprobable situations must

must be in a similar form. For instance three equiprobable branches would produce probabilities of $(\frac{1}{3}, \frac{2}{3}, 1)$. The probability for each successor BB is placed in its Tratable entry.

PART 3

The object program is now simulated many times in order to obtain statistical information concerning relative frequencies of flow paths taken. The number of simulations is equal to the total number of transfers in the program multiplied by 128, which means that the more complex the program, the greater the number of simulations. The program is stepped through basic block, using the BBTABLE, starting with the first basic block. No reference is made to the complied instructions.

A BBTABLE entry is selected, and corresponding SET table entries are obtained. Settings are made according to these entries, that is, the SET address, or setting, is stored in the location given in the decrement. For instance, a Set entry turning a sense light on would cause a 1 to be stored in the dummy sense light address. A Set entry to reset the loopcount of a Do with an if would cause the maximum loopcount to be stored in the Tratable of the Do. (This is where the iterations are counted during the simulation.)

Routine DECODE

The basic block ending code in BBtable is now examined. All possible successor basic blocks to the current one are given in its Tratable entries, and the way in which the successor is chosen is described below under the different endings.

Probability. A random number less than 1, is formed by multiplying a constant 'random' number by a second number. This new random number now replaces the second number mentioned above to ensure a different random number next time. The method now used to select the successor basic block is best illustrated by an example. Suppose a 3-branch probability ending had a frequency statement (1,2,3), which was converted in Part 1 to $(\frac{1}{6}, \frac{1}{6}, 1)$. These quantities are expressible as coordinates of points of a unit line:

The part of the line from 0 to $\frac{1}{6}$ corresponds to the first successor BB, from $\frac{1}{6}$ to $\frac{3}{6}$ corresponds to the second, and from $\frac{3}{6}$ to 1 to the third. If the random number generated as described earlier lies within the first interval, then the second successor is chosen, and so on.

The flow count of the successful successor BB is now increased by 1, and control is returned to deal with this new block.

Certainty. In this case there is only one successor, so its flow count is stepped up, and simulation continues with this block.

Sense light. The successor in this case is determined by the status of the appropriate dummy sense light. If it is set at 1, then it is reset to zero and the 'light on' successor is chosen, otherwise, the 'light off' successor is chosen. The flow count is stepped up, as for other endings.

Assigned Go To. The successor of this type of ending depends on the setting made for the particular 'N', earlier in the simulation. The BB number given for N in the NLIST table (entered there by an earlier BB in the simulation) is matched with the current Tratable entries. If a match is not found, the first Tratable entry is taken. The successor flow count is stepped up as before.

Do with an if. The simulated loopcount, held in its Tratable entry, is stepped up, and if the Do is complete, then the successor BB following the Do is selected. Otherwise, that beginning the Do is chosen.

Do without an if. At this point, such an ending is treated as if it were a certainty ending. Such Dos are dealt with after the simulation pass.

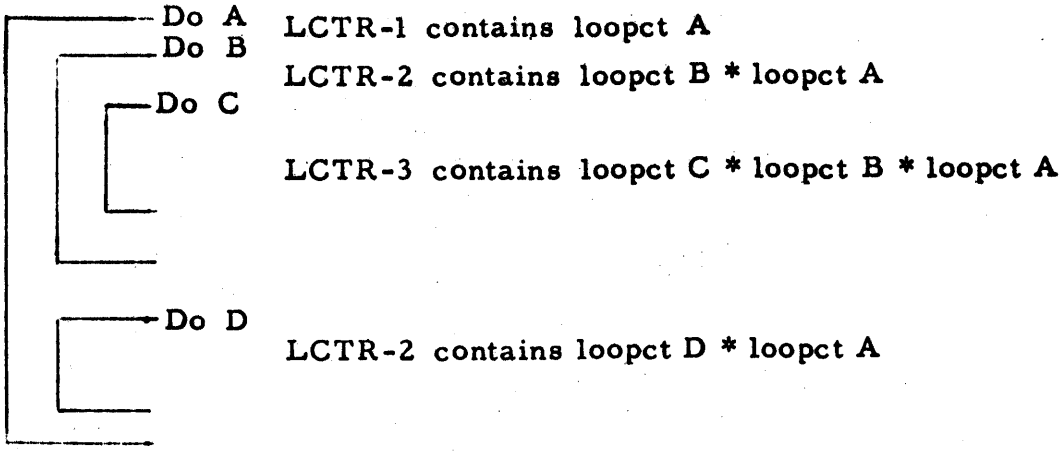
Stop. The current simulation is now complete, and it is recommended at the first basic block of the program, unless the required number of simulations has been carried out.

Routine DODO

After the simulation has been dealt with, this routine adjusts the flow counts of basic blocks which lie within Dos without ifs. If may be recalled that during the simulation, these Dos were not simulated as were the Dos with ifs and therefore the flow counts of the basic blocks within them have not taken account of the loopcounts of the Dos. This is now remedied.

If a basic block lies within several nested Dos, then clearly not one loopcount, but all loopcounts multiplied together will be involved. Dos without ifs are obtained from the DOLIST table, and a new table LCTR is devised to keep track of the loopcount nesting of the Dos as they occur. The first LCTR entry is a dummy, 1. The next always corresponds to the outer Do of a nest, and contains its loopcount. The following entry is for any second level Do and contains its loopcount the outer Do loopcount. For example, in this

configuration quantities in LCTR words are indicated



The successor flow count of each basic block between the beginning of Do_A and the beginning of Do_B is multiplied by the contents of LCTR - 1 i. e. $loopct_A$. The successor flow count of each basic block between Do_B and Do_C is multiplied by the contents of LCTR - 2 i. e. $loopct_B \times loopct_A$, and the successor flow count of each BB except the ending BB, within Do_C is multiplied by $loopct_C \times loopct_B \times loopct_A$. The successor of the ending BB in Do_C will be the beginning of Do_C for $loopct_A \times loopct_B$ $(loopct_C - 1)$ times, and will be that following the Do_C for $loopct_A \times loopct_B$ times, so these successor flow counts are adjusted accordingly. Remaining basic blocks in this nest and in other nesting configurations are dealt with similarly.

Routine DOSUCC

As described earlier, a TRATABLE entry consists of 2 words, the second word containing a successor BB number, with associated flow count, and the first word containing various information used by this section but now no longer required. The order of entries corresponds to the BBTABLE order, though there may be several successor BBs and therefore TRATABLE entries to a given BBTABLE entry. In this routine, the 2nd Tratable word is rearranged slightly and placed in a new table called SUCC. This means that for every basic block in BBTABLE, there is one or more SUCC entry containing associated flow counts.

There is another way of looking at it, namely, given a basic block, which are the preceding, or 'predecessor' basic blocks? This information is complied in the following manner. A pass is made over Tratable, together with BBTABLE, to enter in word 1 of each successor BB Tratable entry, the current BBtable number, which is of course, their predecessor. This new form of Tratable is now sorted in order of Successor BB number, and the result is that for each basic block in turn, all predecessors are grouped together. Now, word 1 of Tratable becomes the PRED table, and relative addresses of PRED BBS corresponding to the Basic Blocks are entered in BBtable. We now have three basic tables to be passed on for the Section 5 analysis:

BBTABLE

	Address of first relevant SUCC entry			Address of first relevant PRED entry		
	1	14 21	35	1	14 21	35
SUCC:	Flow count	Successor BB Number	PRED:	Flow count	Predecessor BB Number	

Routine TAGPAS

The third and final pass is now made over CIT to collect tag information for Section 5, and during this pass two new tables are built up: TAG and BBTAG. For each occurrence of a tagged instruction, an entry is made in TAG, consisting of the symbolic tag name, together with a code according to the following list.

LXA, LXD, PAX, PDX	1
LXP	2
DED	3
TIX, TXI	5
Other (passive)	6

See chapter VIII page 13

When the instruction begins a basic block, the sign of the entry is set negative.

Each time the beginning of a basic block is encountered, a 'BBTAG entry is made, containing the number of entries so far in TAG, together with octal 33, the latter being for the convenience of Section 5. The last BBTAG entry is a dummy and contains the total number of TAG entries.

Finally, details concerning table lengths are left in locations called Keys, for Section 5, and the BBtable and BBtag tables are combined to form one table - BBBTABLE, each entry of which consists of 6 words, the last four being zero.

APPENDIX TABLES CREATED DURING SECTION 4

BBLIST This is written in Part 1 and is an ordered list of the beginning locations of all basic blocks within the object program.

BBTABLE This is written during the second pass over CIT, in Part 2.

Prefix	Decrement	address
BB ending code	Starting location of corresponding Tratable entries	Starting location of corresponding Set entries

BBTAG This is written during the final pass over CIT in Part 3. An entry is made every time a new basic block is reached, and it contains the number of TAG entries made so far. The final entry contains the total number of TAG entries made.

DOFRET See FRET.

DOLIST This is written in the first pass, during Part 1, and one entry is made for each Do in the object program. The entry consists of 2 words:

Location of beginning of Do
Location of end of Do

and is replaced by:

BB No. of beginning
BB No. of end

FIXDO This is a minor table, used only in the 2nd part. When it is found necessary to reset the loopcount of a Do with an if, an entry is made here which contains the address of the Set entry described under SET in this appendix.

FRET This table originates in section 1, and is modified in Part 1. The first word always contains the internal formula number to which the frequency applies, together with a minus sign. Frequencies follows, one word per frequency. This also applies to Do frequencies. Do type frequencies are then erased from FRET and transferred to a new table DOFRET. Meanwhile, the remaining frequencies are changed from the form $(x_1, x_2, x_3, \ldots x_n)$ to $\left(\frac{x_1}{\sum x}, \frac{\sum x}{\sum x}, \frac{\sum x}{\sum x} \ldots \ldots \ldots 1\right)$.

LCTR This is an internal table. The flow counts of basic blocks lying within Dos without ifs are multiplied by its contents after the simulation.

1st entry	Dummy 1
2nd entry	Loopcount of outer Do of a nest.
3rd entry	Loopcount of any next inner Do X 2nd entry
4th entry	Loopcount of any next inner Do X 3rd entry

. . . and so on.

NLIST This is a list of symbols of Assign statements.

SET This is written in Part 2, consists of settings made during the object program, and is used in the simulation.

Sense light	Address(es) of dummy light(s)	0 if turned off 1 if turned on
Assign	Reference to NLIST where Assign symbol is stored.	Successor BB No.
Entry to reset a Do with an if	BB No. of end of Do	Maximum loopcount of Do

This is replaced later by:

Address of 1st Tra-table entry of end of Do.	Maximum loopcount

TAG This is written during the final pass over CIT, in Part 3, and an entry is made for every tagged instruction. The code depends on the type of instruction, and a full list is given in the main text.

Prefix		Address
Code		Symbolic tag

TIFRD This table consists of the Assign entries from TIFGO, which has code 6 in the first word with the internal formula number, and the Assigned Go To entries of Tifgo and Trad. The latter Tifgo entry has code 1 with the internal formula number, in the first word., and has the number of transfers (i.e. number of Trad entries) in the second word.

TRATABLE This is built up during Part 2. Each basic block ending, except a stop, has a group of one or more Tratable entries, giving the successor basic blocks. Different types of entry are shown.

TRATABLE (cont.)

Assigned Go to:

Wd. 1	Relative address of NLIST entry	
Wd. 2	Successor BB No.	flow count during simulation

followed by an entry for each other successor basic block, with only word 2 used.

Do with if:

Wd. 1	Loop count	
Wd. 2	Successor BB No. (that beginning Do)	flow count during simulation
Wd. 1	Used for simulating loop count	
Wd. 2	Successor BB No. (that ending Do)	flow count during simulation

Do without if: Exactly the same as Do with if, except that word 1 of the second entry is not needed for simulating iterations.

Conditioned transfer and Go to Vector:

Wd. 1	Location	
Wd. 2	Successor BB of 1st Branch	flow count during
Wd. 1	No. of Branches	
Wd. 2	Successor BB of 2nd Branch	flow count during simulation

Remaining branches each have one entry, and only the 2nd word is used.

All probability cases: The second word is the same as the preceding cases. The first word contains the probability quantity that that particular branch should be taken, and there are as many entries as branches.

PRED: This is derived in Part 3 from TRATABLE and BB TABLE.

Flow count	Predecessor BB No.

SUCC: Identical with Pred, except that the successor BB number is in the address portion.

VIII

SECTION FIVE

The following discussion of Section Five is divided into 4 parts corresponding to the division into records on the Fortran system tape. In addition, following the description of parts 1 and 4 is a summary of the frequently used subroutines for that part. Following part 4 there is also a description of the tables used in Section Five. Since some of the concepts used are also explained under the headings of the subroutines and tables, the reader may find it useful to refer to them while reading the main text. Also, see illustration page VIII - 12.

PART I

Section Five uses the information about basic blocks (which has been passed on from Section Four) to combine these basic blocks into larger groups called regions. The flow within a region is simulated in order to determine which symbolic tags are required and which index registers should be assigned to them. During the course of simulation, flags are set to indicate where an SXD or LXD is required. When a region has been treated it may be combined with other regions. Eventually all basic blocks will have been combined into a single region, and the complete object program will have been treated.

The most frequent paths of flow between basic blocks are handled first. Since an SXD or LXD is not inserted until necessary, this results in the most frequent paths having the least of them, and therefore a faster object program.

Region Formation

The first step of this treatment is the formation of a looplist showing the path of flow through a new region. The starting point in looplist formation is the most frequent link between basic blocks which has not yet been considered. (The PRED and SUCC tables have frequency counts which are examined to find most frequent predecessor or successor basic blocks. When a link has been treated, the entry which refers to it is marked with a minus sign so it will not be considered again.) Looplist is expanded by including as many of the most frequent unconsidered predecessors as possible and then as many of the most frequent successors as possible. If the most frequent link is to a basic block which is in a region previously treated, this whole region is included in the looplist. Thus a looplist may consist of a combination of untreated basic block and regions (or basic blocks which have already been treated).

Regions are classified as either opaque or transparent. An opaque region is one in which all three index registers are used. A transparent region has one or more index registers still available. When an opaque region is encountered while forming looplist, no more links are added to it. However, a transparent region may still be added to, since there are index registers available within it to which tags can be assigned.

The looplist table consists of one word entries for each basic block or region. A code in the prefix of the word indicates whether it refers to a basic block, a transparent region, or an opaque region. If the entry is a basic block it contains the BB number, and if the entry is a region it contains the numbers of the basic blocks at the entry and exit points of the region. The end of looplist is indicated by a word of all sevens.

From the starting point in looplist, the most frequent predecessors are added one at a time until one of the following conditions have been encountered. If an entry is already in the current looplist, this makes looplist a loop and prohibits further building. If an entry is an opaque region or if there are no unconsidered predecessors, then additions are made at the other end, and the most frequent successors are looked for. Again the same conditions apply. Basic blocks or regions are added until a loop or an opaque region is encountered, or there are no unconsidered successors to the last entry. When a looplist has been completed, it will reflect the flow in a section of the object program. It may have a loop, reflecting a loop in the object program. In such a case, if there is an end of looplist not included in the loop, that section is eliminated from looplist. Only the loop itself will remain in looplist for further treatment in this looplist. On the other hand, the looplist may be a string with no loops, having been stopped in both directions by encountering an opaque region or by finding no unconsidered links to it.

After the looplist has been formed, the path of flow indicated is ready for treatment. The next step is to prepare for simulation which is done in the 2nd LXing pass. If the looplist is a string, then the only preparation necessary is to mark the initial conditions of the IRs. If the looplist is a loop, however, the 1st LXing pass is entered.

1st LXing Pass

The index registers used by the object program are simulated in Section Five by three storage locations which are continually updated. These cells are referred to as IRs. During simulation they will contain the symbolic tags needed by the corresponding part of the object program.

The 1st LXing pass simulates the loop in order to find out the condition of the IRs when the 2nd LXing pass is begun. Each basic block in the looplist is examined to see which tags are necessary. This is done by referring to TAGLIST (which is a table containing a list of all tagged instructions in the object program.) Tags are placed in the IR cells as required. When a region is met in looplist, the previously determined exit conditions from the region are placed in the IRs. After the whole looplist has been done the IR cells contain the initial conditions for the 2nd LXing pass.

2nd LXing Pass

Simulation in the 2nd LXing pass is much more complex then the cursory treatment of the 1st LXing pass. Entries are made in tables when a tag must be loaded

into or displaced from an IR. STAG is used to record LXs and SXs within a basic block, and PRED is used for those between BBs. When a tag is displaced, its value is saved if necessary in a cell set aside for that purpose. These tag cells are thus kept up to date so that the next time a tag not already in an IR is required, an LX from the corresponding cell will be correct.

In order to determine when an SX is necessary, the concept of activity is used. When the initial value of a symbolic tag is set, or when that value is changed by an indexing instruction such as TXI, the IR becomes active. This means that the value in the storage cell corresponding to that tag is outdated. This fact is recorded in cells referred to as AC 1, 2, and 3, one for each IR. If this tag must be displaced while treating the same looplist, an SX will be introduced immediately after the active instruction, thus updating the tag cell and ending the activity. But if the tag has not been displaced from the IR after treatment of the looplist, the section of looplist is marked active from the point of the active instruction. This is done by placing activity bits in the BBB entry for each BB in that section of looplist. When flow goes through such a BB in a subsequent looplist, the activity will be noted, and if a future SX is necessary it will be placed in the link from the region containing the BB.

A tagged instruction that does not change the value of the tag, does not require this treatment. Such an instruction is called passive. A passive instruction, such as CLA or TXL, only makes it necessary to have the appropriate tag in an IR. When a tag is required that is not already in an IR, an LX from the appropriate tag cell is called for. Because of the way activity is handled, the tag cells may always be considered up to date. All that is necessary is a determination of the most desireable IR to use. If they all contain tags, this is done by searching ahead to find out which of the tags presently in the IRs will be needed last.

Treatment in the 2nd LXing pass begins with the first entry in LPLST and proceeds in sequence to the last. The three types of entries, 1) BBs, 2) transparent regions, and 3) opaque regions, are distinguished by a code number and each is treated differently.

1. Treatment of a BB

If the LPLST entry is a BB, simulation of the object program is accomplished by examining all the tagged instructions in the BB and making the necessary provisions for the tags used. The instructions in TAGLIST for this BB are taken in sequence, and the IRs are updated as necessary.

Each IR is examined to see if the symbolic tag required by the instruction already is present. If the tag is not in an IR, it is put into the most replaceable one, and the STAG entry corresponding to this instruction is marked to show that an LX from the tag cell is necessary. The LX is also recorded in the region table entry. If the instruction is an active instruction, an active indicator is also stored. If a tag had to be displaced from an IR, and that IR was active from a previous instruction, then an SX is necessary. This is recorded in STAG if the activity was caused by an instruction in a BB in this LPLST, or in PRED if the active instruction was in an already treated region. A PRED entry is necessary because

once a region has been treated, nothing is changed within it. Thus the SX will appear in the link from that region to the current LPLST. The STAG entry, on the other hand, calls for compilation of an SX immediately following the active instruction.

The appearance of an active instruction using a tag already present in an active IR will cause the section of LPLST to be marked active.

When a DED pseudo instruction, compiled to tell Section Five that a tag is now valueless, is encountered and the tag is not in an IR, nothing need be done. An IR containing that tag will be loaded with a "hash" symbol, indicating its contents are no longer of any value, and if the IR was active, the section of LPLST will be marked active.

For each taglist instruction an entry is made in STAG to record which IR to use. After all the tagged instructions within a BB have been examined and proper table entries made, the entrance requirements (tags needed) and exit conditions for the three IRs are stored in the BBB table. BBB also will have bits indicating which IRs are active within the BB, and has information passed on from Section Four about how the basic block ends and the numbers of the SUCC and PRED entries referring to this BB. Thus BBB is a summary of the basic block, and the individual instructions need not be looked at again.

If the LPLST entry just treated ended with an Assigned GO TO, some extra treatment is required. If there are any active IRs an SX is recorded as necessary.

2. Treatment of a Transparent Region

Entries in LPLST which are regions have had all the BBs in that region simulated at the time that region was formed. Therefore, it is not necessary to go through its tagged instructions again. However, it is necessary to take care of the links to and from the region. The best match possible is made between the current IRs and the entrance requirements of the entry BB in the region. This may require permuting index register assignment in the region. For example, if a tag T_1 is in IR1, when a region's entrance requirement is T_1 in IR2, then the region's index register assignment may profitably be changed to have T_1 in IR1. The STAG, PRED, and BBB tables become obsolete by this change and must be updated. The tables are not actually changed, however, since the tables are read through permutation numbers in BBB, thus only these permutations are changed. The numbers were originally set to 1, 2, 3, (octal 33) by Section Four, meaning entry 1 is IR1, etc. If they were changed to 2, 1, 3, they would mean entry 1 is now IR2, entry 2 is now IR1, and entry 3 is IR3. LX and SX bits in PRED will take care of problems not solved by permutations.

Processing a transparent region entry in LPLST begins after matching the region's entrance requirements. A pass is made over the BBB entries for the basic blocks in the region. If the region's entrance requirement for a particular IR is empty (there must be at least one of these, since by definition a transparent region has one or more empty IRs), the current tag for that IR may be carried through the

region. The new entrance and exit conditions of the IR are stored for each BB. On the other hand, if the entrance requirement is not empty and does not match the current tag in the corresponding IR, and if the IR is active, it becomes necessary to examine the exit conditions of the BB. When the BB exit condition does not match the corresponding IR but does match a different IR, an SX is recorded necessary. If the BB exit does match and the IR is either active or does not match at the region exit, the IR is marked active in this BB. After each BB has been treated in this fashion, the new permutation numbers and active indicators are stored in BBB.

When all the BBs in the region have been examined the region is considered as a whole. If the entrance condition for an IR is empty but the IR matches a different region exit, a "hash" symbol is put in the IR since the same tag should not appear in more than one IR simultaneously. When the region entrance condition is "hash" and the IR is active, an SX is called for. If the region entrance requirement matches the IR but the region exit is "hash", empty, or active, then the section of LPLST is recorded active by marking the necessary BBB entries active for the corresponding IR. When the region entrance requirement is a symbolic tag which is not already in an IR, an LX is called for in the link to the region, and if the IR was active, an SX is also indicated. At the conclusion of this processing the region's exit conditions are in the IRs, the active indicators are set, the region is permuted for the best possible match to the preceding section of LPLST, and any remaining empty IRs are noted.

3. Treatment of an Opaque Region

The processing of an opaque region in LPLST is simpler than that of a transparent region. This is true because there is no possibility of carrying a tag through the region, since there are no empty IRs. A match of the region's entrance requirement is made if possible, and the permutation numbers updated. When the entrance matches the IR but the IR is not active, nothing further need be done. But when under the same conditions, the IR is active, either because of an LX within the region or, if the region is the same one that started LPLST (a loop condition), the IR was not active at the start of LPLST, then an SX is necessary. If there was no LX for the IR in the region, and the region is not the same one that started LPLST or the IR was not active at the start of LPLST, then an SX is not yet necessary, but the BBs in the region and the section of LPLST to this point are marked active.

When the contents of an IR do not match the tag required at the entrance of an opaque region an LX is recorded as necessary, and if the IR was active, an SX is of course also indicated.

After the 2nd LXing pass has been finished and all the LPLST entries dealt with, any remaining activity must be taken care of. If the LPLST was a string this is done very simply. It is only necessary to mark a section of LPLST active for any remaining active IRs. In the case of a loop, however, the problem is more complex. The entrance requirements at the beginning of the loop are examined. If a requirement is not a real tag (it is either "hash" or empty) and the IR is

active, then an SX is recorded. If the entrance requirement is a tag different than the contents of the IR, an SX and LX are recorded. At this point if the SXs have taken care of the activity and there are no more active IRs, the problem is solved.

Active Pass

If there are still active IRs remaining, just as the 1st LXing pass was required, another pass, the active pass, is executed. LPLST entries are examined and treated again in a manner similar to that of the 2nd LXing pass, with SXs called for where necessary. After each LPLST entry has been dealt with, a test is made to see if there is still an active IR. Eventually they will have all been taken care of and the active pass finished.

Table Updating

It only remains to bring the appropriate tables up to date. The PRED and SUCC entries that have been treated are flagged negative. BBB has the new region references entered. And finally the region table is updated by wiping out obsolete entries (regions absorbed into the new one) and making the entry for the new region.

Part 1 repeats the cycle of looplist formation and treatment, with new, large regions absorbing old ones, until all links between basic blocks have been treated and the object program consists of a single, all encompassing region.

SUBROUTINES USED IN PART I

SE GROUP

These routines compute the correct references to the STAG, SUCC, PRED, and BBB tables for a desired entry. They are entered with the item number in the AC and return with index register 1 loaded appropriately.

S1

This routine selects the most replaceable IR by scanning ahead through LPLST and noting how long it will be before the present tags will be required again. That IR whose contents will be needed last is the most replaceable. If this routine is entered at S111 it does the reverse, that is search for the least replaceable IR, the one whose tag will be needed first. This routine also uses S2 as a subroutine.

S3

This routine can exist in two states, "Feed LPLST" or "Feed Tag". In the "Feed LPLST" state it will feed the next item in LPLST and take the LPLST Feed exit. It switches to the "Feed Tag" state when the last LPLST item it fed was a BB and not a region. It then will feed the next item in TAGLIST and take the tag feed exit until it has fed the last TAGLIST item for that BB, when it returns to

the "Feed LPLST" state. When the end LPLST sentinel is fed it re-initializes itself to the beginning of LPLST and to the "Feed LPLST" state. The routine uses the S4 subroutine for handling the taglist tape.

S5

This routine will specify the permutation of index registers which will provide the best match between the IRs and the entrance requirements of a region.

During treatment of a LPLST, the object program index registers are simulated by the IR cells, which contain a symbolic tag, the empty symbol (octal 777777), or the hash symbol (777776). Entrance requirements for a region will be placed by S5 into the En 1, 2, 3, cells which S5 will try to match against the IRs. The optimal match for the IRs will be in the IN cells, and similarly, the best match for EN 1, 2, 3, will be in EN 4, 5, 6. For example, IN 1 is set to 3, 2, or 1 depending upon whether the correspondent of IR1 is EN 1, 2, or 3. Also EN 4 will be set to 3, 2, or 1, depending upon whether the correspondent of EN 1 is IR1, 2, or 3. Thus if IR1 matches EN 1, IN 1 will contain 3 and EN 4 will contain 3.

S5 uses S1, S6, S7 and S9 as subroutines.

S9

This routine is used by S5 to load EN 1, 2, and 3 with the tags needed by the first BB in a region as entrance conditions for the index registers.

SA

This routine loads the EX 1, 2, 3 and ACT 1, 2, 3 cells from the exit conditions and activity bits in the BBB table of the exit BB in a region.

SB

This routine enters the PRED or STAG bits to record that an SX is to be compiled. The appropriate activity cell, AC1, 2, or 3, is examined. These cells describe the status of IR 1, 2, 3. They may contain plus 0, indicating that the IR is not active; plus activity, meaning that the active instruction occurred in a BB which has not been treated until this LPLST; or minus activity, meaning that the active instruction was in a BB which is in an already treated region. If SB finds plus activity, the SX bit is placed in the STAG entry for the active instruction. If it finds minus activity, the SX bit is entered in PRED in the link from the region.

SC

This routine is used to mark a section of LPLST active.

An index register becomes active when, in the simulation of a new BB in the 2nd LXing pass, an active instruction (LX, TXI, or TIX) is encountered. The activity produced is plus activity. If while treating the same LPLST the contents

of the IR must be displaced, SB is entered and will record an SX necessary in STAG. This SX completely takes care of the activity problem, and the activity is ended.

But if at the end of LPLST an IR is still active and the need for an SX has not yet arisen, the compiling of the SX may be postponed. However, it is not safe to destroy all record of the activity, for an SX may be needed in treating a later LPLST. In such a case SC is entered and will transfer the activity from AC 1, 2, 3 to the BBB table for all BBs during which the IR is active. When entered, SC examines the designated AC cell. If it is not active, nothing is done. If it is active, an entry is made in the prefix of word 2 of BBB for every BB between the origin of the activity and the present point of LPLST, and the AC is turned off.

The activity has now become minus activity, and can never be ended. The appearance of such a BB is a subsequent LPLST will cause the appropriate AC to contain minus activity, and whenever the contents of the corresponding IR must be displaced, the SB routine will put an SX bit into the PRED link.

This scheme of postponing the compilation of an SX whenever possible, has the general property of producing a larger number of SXs than strictly necessary. However, sincd the high frequency paths are treated first, the SXs will appear in the lower frequency paths. Thus to save object program time, Section Five will trade object program space.

The SC routine uses SD as a subroutine.

SD

This routine is used to mark the BBs in any one region active.

SF

This routine forms the appropriate AC 1, 2, 3 entry when an active instruction is encountered.

SG

This routine does the permutation of a REG entry as indicated by EN 4, 5, 6.

F1

This routine finds the highest frequency unconsidered PRED entry for a given BB.

F30

This routine finds the highest frequency unconsidered SUCC entry for a given BB.

PART 2

In part 1, tags were continually reassigned to index registers on the basis of the optimal match that could be achieved. This reassignment was done by changing the permutation numbers in the 2nd word of the BBB table. Part 2 makes the actual changes in the appropriate tables on the basis of the final permutation numbers. It also combines BBLIST (and some information about Assigned GO TO statements), with BBB for convenience later on.

Each basic block is examined in sequence. The location word of CIT for the first instruction in each BB (which has been put in BBLIST by Section Four) is placed in word 6 of BBB. Then the LX and SX bits in the PRED entries are changed according to the permutation numbers. Next, the STAG entries are similarly updated. Then, for each BB which ends with an Assigned GO TO, the BB number of the last assigned GO TO is stored in word 2 of BBB. This is done in order that part 3 may find all GO TO N BBs easily. Finally, the entrance and exit conditions in words 3, 4 and 5 of BBB are reentered in accordance with the permutation numbers.

PART 3

Section Five may insert SXDs and LXDs at points in the object program which are transferred to by an Assigned GO TO. It may therefore happen that the transfer should no longer go to its original address, but to one of the SXs or LXs. Part 3 handles this by making the necessary changes in the assign constants.

For each BB which ends with an Assigned GO TO, part 3 finds the successor BBs and their appropriate PRED entries. From the SX and LX bits in PRED, the correct transfer address is prepared. The Assign Constants are then compared to the first instruction of each successor BB, and when a match is found the assign is replaced by the hew symbol. The SX bits are also stored in the prefix of word 2 in BBB for use by part 4.

When all the Assigned GO TO BBs have been treated, the altered assign constants are written back on tape for Section Six, and part 3 is finished.

PART 4

Part 4 does the actual compilation of instructions on the basis of the information passed on by the previous parts of Section Five. The bits in PRED indicate when inter-block SX and LX instructions are required. STAG has the necessary information about when to compile an LX or SX immediately preceding or following a tagged instruction in CIT. The real index register assignment for each tag is also indicated by bits in STAG. Part 4 follows these directions while compiling. In addition, some minor optimizing is done.

A pass over CIT is made, and the method used to bring in blocks of instructions and scan them for tagged instructions and endings of BBs is similar to that used by Section Four. This is the only time that Section Five looks at the CIT. The

instructions are brought in from tape and examined in groups, and when the necessary modifications have been made, they are rewritten on tape for Section Six.

First, part 4 considers a basic block as a whole. By referring to the BBB and PRED entries for the BB, a list of the necessary LXs in the links to the BB is formed. Then a list of the necessary SXs in the link is formed. When the SX lists are compiled for the various PREDs, it may happen that two or more of these are the same. The symbolic locations of these SX lists will be different, however, because the number of the PRED entry is contained in the location symbol. A SYN pseudo instruction is compiled in this case.

A "sequential transfer", which is one from the last instruction in the previous BB to the first instruction in this BB, is compiled if necessary. The transfer may be around one or more lists of LXs and SXs associated with other PREDs for this BB. On the other hand, the transfer may be dropped if no instructions had to be inserted between the BBs.

After the inter-block SXs and LXs have been taken care of for each BB, all the instructions within the BB are handled. All CIT entries without tags are, of course, kept. A CIT entry which already has a real tag of 4 is checked to see if it is an SXD or LXD which has been placed around a subroutine calling sequence. If such is the case and if IR4 is not necessary for Section Five assignment of a symbolic tag at this point, the SXD or LXD will be deleted. The SXD location will be compiled as a BSS 0 since it may be referred to elsewhere in the program. When an LXD after a subroutine calling sequence cannot be deleted because IR4 is necessary, if the following instruction is a similar SXD, both are deleted. As a result, a series of TSX instructions will have the unnecessary SXDs and LXDs removed.

When an instruction with a symbolic tag is encountered in CIT, the STAG entry referring to it is examined. If STAG requests it, an LX from the tag cell will now be compiled. Then the instruction itself is compiled and next an SX to the tag cell if so indicated. Each of these instructions will have had the real tag assigned also on the basis of the STAG entry. The LXP pseudo instruction is deleted when it occurs, as is a DED. These instructions were put in as signals to part 1 and are no longer required.

After an instruction has gone through the foregoing treatment, it is checked to see if this is the end of the BB. If it is not, the next CIT entry is examined and treated. When the ending is found, any transfer addresses are examined to see if the transfer is to a BB with SXs or LXs in the PRED link. If it is, the address is changed to the location of the proper SX or LX. Any "sequential transfers" are not compiled at this time, however. An indicator is stored if there is one, and the deletion or insertion of this transfer is left up to the analysis of the PRED link when the next BB is treated. The case of an Assigned GO TO ending is treated differently. The SX bits placed in word 2 of BBB by part 3 are examined and SXs compiled where necessary. Then the transfer to N is compiled.

When all the instructions in the BB have been treated and the ending taken care of, the next BB is dealt with as before. The process continues until the end of CIT is reached. Finally the relative constant routines are copied at the end of CIT and control passes to the Section Five Prime of Fortran.

SUBROUTINES USED IN PART 4

SAD

This routine determines the correct address of a transfer instruction. It is entered with the BB number in the AC and the successor BB number in ARG1. It returns with the address in the AC. The PRED entry for this link is found, and the LX and SX bits used to determine the SX or LX case and form the symbolic address.

SCMI

This routine compiles an instruction if CPIND indicates it should be compiled. CPIND is a cell which is negative when the instruction should be compiled, but is made positive when an instruction has been compiled while treating a tag. This is done so that a tagged instruction at the end of a BB, when SCMI is entered, will not be compiled again.

SH

This routine determines an SXD case. It uses the SX bits in PRED and the exit conditions of the predecessor BB from BBB to determine which tags have to be saved in this PRED link.

SI

This routine compiles an SXD case, and if it is not associated with an LXD list, compiles the appropriate transfer to an LXD case.

SJ

This routine determines if the SXD case is associated with an LXD list.

SK

This routines makes the actual entries in CIT when an instruction is compiled.

SL

This routine compiles a transfer to an LXD case.

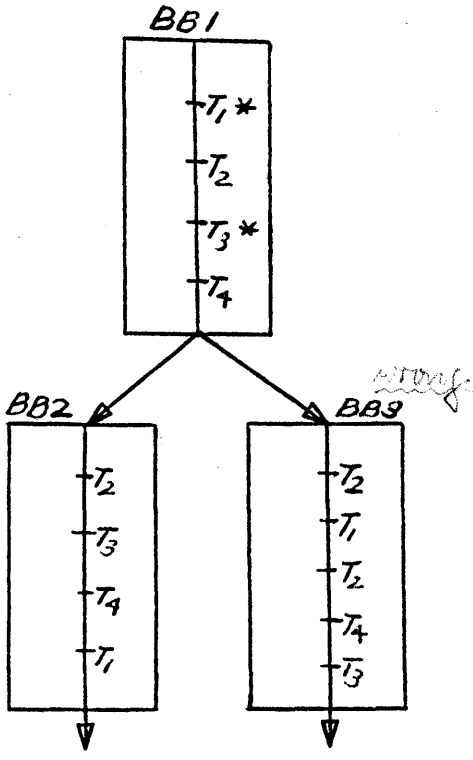
SL1

This routine compiles a hanging transfer if necessary.

SM

This routine compiles an LXD list.

As an example of SXD and LXD placement by part 1, consider the following illustration.



In this diagram BB1 has two possible successors, that is a conditional transfer may go either to BB2 or BB3. The tagged instructions in each BB are shown with tags T_1, T_2 etc. The asterisk next to 2 tags in BB1 indicates an active instruction. Let us say that the link between BB1 and BB2 is the most frequent. It will therefore be in the first LPLST and incorporated into a region. Let us also say that the IRs are all empty when BB1 is treated. Then LXs will be indicated in STAG as T_1, T_2, and T_3 are encountered. An LX will also be necessary for T_4, but this invloves displacing a tag in one of the IRs. Since T_2 and T_3 will be needed again first, T_1 is chosen to be displaced, and because it is active an SX will be indicated in STAG. When BB2 is treated and T_2, T_3, and T_4 are encountered, nothing need be done, as they are already in the IRs. For T_1 an LX is necessary and will be called for in STAG. The LX will be correct since the tag cell was updated by the SX in BB1 after the active instruction. Later, when the link from the region (containing BB1) to BB3 is treated in another LPLST, the exit conditions will be T_2, T_3, T_4. Then in BB3 when T_2 is needed it is available, but T_1 will cause an LX bit in STAG. Since T_2 and T_4 will be required next, T_3 will be displaced. The IR containing T_3 is active, and since the activity was present from an already treated region, an SX bit will be placed in PRED for the LX to be compiled in the link between BB1 and BB3. Then when T_3 is again required an LX will load the proper value.

From this example it may be seen that in one case an active instruction was handled by an SX immediately following it. In the other case, however, there was no need for an SX in the most frequent path of flow, and the compilation of the SX was postponed until a less frequent path was treated.

SECTION FIVE TABLES

LPLST (looplist). This table is used only in section 5, part 1. A LPLST is formed defining each new region to be treated. The table has one word entries which may be either a BB or an already treated region. An entry for a BB has the BB number in the decrement. An entry for a region has the entry BB in the decrement, and exit BB in the address. If the region is at the beginning of LPLST it will have all sevens (octal) in that portion of the word normally occupied by the entry BB (decrement), and if at the end of LPLST will have all sevens in place of the exit BB (address). An example of a LPLST is shown below.

	P	Dec.					T	Add.				
Wd. 1	1	7	7	7	7	7	0	0	0	0	1	4
Wd. 2	0	0	0	0	2	3	0	0	0	0	0	0
Wd. 3	1	0	0	0	2	6	0	0	0	0	3	2
Wd. 4	2	0	0	0	0	3	0	7	7	7	7	7
Wd. 5	7	7	7	7	7	7	7	7	7	7	7	7

Code in Prefix:
0 - Basic block
1 - Transparent regio
2 - Opaque region

The first word shows that LPLST begins at the exit of a transparent region with BB 14. Then BB 23 (word 2), and then a transparent region (word 3), with entry BB 26 and exit BB 32. Next BB 3 (word 4) entering an opaque region which ends LPLST. The word of all sevens is the flag marking the end of LPLST.

REG (region). This table is used only in section 5, part 1. The table contains a one word entry for each already treated region. When an old region is incorporated into a new one the old entry is wiped out. Bits in the prefix indicate the presence of an LX for that IR within the region. Tag bits indicate that the IR is empty in the region. The last bit in the address is present to distinguish a possible real entry (BB 0 with no LXs and no empty IRs) from the absence of an entry.

Pre.	Decrement	Tag	Address
xxx	No. of 1st. BB in reg.	xxx	1

CMTAG (taglist). This table is read from tape 3, file 3, in 15 word records. There is a one word entry for each tagged instruction in CIT. The sign is negative if the instruction begins a basic block.

Pre.	Decrement	Tag	Address
x	code		Symbolic tag

Codes in decrement:
1 - LXD, LXA, PAX, PDX
2 - LXP
3 - DED
4 - TNX
5 - TIX or TXI not used to transfer
6 - Passive instruction
7 - TIX used to transfer
8 - TXI used to transfer

STAG. This table is formed in section 5, part 1, and used by section 5, parts 2-4. Each STAG word refers to 9 taglist instructions. The first nine bits call for an SX following the corresponding tagged instruction, and the next nine bits for an LX preceding the instruction. Bits 18-35 in pairs specify the index register each tagged instruction is to use. In the following illustration, the numbers refer to the entry within the STAG word.

S	9	18 35
123456789	123456789	11 22 33 44 55 66 77 88 99

BBB. This table is passed on from section 4 in its initial form with words 3-6 set to zero. There is a 6 word entry for each BB. During section 5, part 1, the permutation numbers in bits 12-17 of word 2 may be changed, the IR activity in this BB is indicated in the prefix of word 2, and words 3-6 are filled in. Word 2 is changed in parts 2 and 3 to contain information about Assigned GO TOs.

	Pre.	Decrement	Tag	Address
Wd. 1	Code	Number of 1st SUCC referring to this BB		Number of 1st PRED referring to this BB
Wd. 2	IR active	perm. nos.		No. of 1st taglist entry belonging to this BB
Wd. 3		Entry Requirement IR4		Exit condition IR4
Wd. 4		" " IR2		" " IR2
Wd. 5		" " IR1		" " IR1
Wd. 6		index of region this BB is in		no. of the next BB in the region

Code in word 1 prefix describing ending of BB:

0 - Do with an if
1 - MSE
2 - Probability branch
3 - Certainty
4 - Do without an if
5 - Go to N
6 - Stop

Word 6 is changed in section 5, part 2, to the location of the first CIT instruction in the BB from BBLIST.

PRED. This table is passed on from section 4. There is a one word entry for each predecessor BB. Thus each BB has as many entries in PRED as it has predecessor BBs. Section 5, part 1, uses bits 15-17 to call for SXs and bits 18-20 to call for LXs for the 3 IRs in this link. Section 5, parts 2-4 use this information. The sign is made negative after the link is treated in part 1.

S	15	18	21 35
x Frequency of link	xxx	xxx	No. of predecessor BB

SUCC. Similar to PRED, except entries refer to successor BB

S	14	21
x Frequency of link		No. of successor BB

BLIST (BBLIST). This table is read from tape 3, 4th file, record 1 by section 5, part 2. There is a one word entry for each BB. The entry contains the first word (location) from the CIT instruction which begins the BB.

	Decrement	Address	32
	Internal formula no.	Instruction no.	

ASCON (assign constants). This table is read from tape 3, 8th file as one record by section 5, part 3. There is a one word entry for each assign constant consisting of the location word of the CIT (see format of BLIST above) which is assigned as the transfer address. If section 5 has inserted SXs or LXs in the link to this BB, it will change the ASCON entry to the new transfer address. An example of an LX location is shown below. The octal 15 in the first 6 bits is translated by section 6 as D). After making any necessary changes, ASCON is written back on tape 2 as the 10th file.

	23-25	
15 0 0 0 0 0	Combined LXD case	BB no. of successor

SECTION FIVE-PRIME

The purpose of section 5 prime is to add to the CIT file all constants and source program data appearing in the symbolic listing, except for the B) and 9) constants, for use of section six.

At the end of section 5 the CIT file contains the entire working program, the arithmetic statement function definition subroutines, and the relcon computation subroutines A). Available to 5 prime are tables of the values of assign constants 5), fixed point constants 2), floating point constants 3), format BCD words 8).

Assign constants are in the ASSIGN table, one record of file 10, tape 2. The table format is

word	1	number of assign constants N
word	2	constant 1
word	3	constant 2
word	N+1	constant N

Each assign constant is a one word binary number in the decrement field 0IIIII000000, where IIIII is some internal formula number used in the program.

Fixed point constants are in the FIXCON table, one record of file 9, tape 2. The table format is

word	1	number of fixed point constants N
word	2	constant 1
word	3	constant 2
word	N+1	constant N

Each fixed point constant is a one word signed binary integer in the decrement field 0YYYYY000000.

Floating point constants are in the FLOCON table, record 1 of file 4, tape 2. The table format is

word	1	number of floating point constants N
word	2	constant 1
word	3	constant 2
word	N+1	constant N

Each floating point constant is a one word signed binary number of 8-bit exponent and 27-bit mantissa PPPMMMMMMMMM.

Format BCD words are in the FORMAT table, record 2 of file 4, tape 2. The table format is

word	1	identification number 10
word	2	number of words in table N
word	3	symbol 8)K

word	4	BCD word
word	5	BCD word
word	P	flag 777777777777
word	P+1	symbol 8)L
word	N+2	flag 777777777777

A format statement gives rise to one internal symbol, an indeterminate number of BCD words, all followed by an end of statement flag.

To initialize 5 prime, the last CIT record previously compiled is read from tape 3. To this record, and to additional records as required as each record is filled and written off on tape 3, is added a four-word CIT for each word in each of the tables.

The first assign constant yields a CIT entry of the form

word	1	050000000000
word	2	636121000000
word	3	0IIIII000000
word	4	000000000000

which appears in the symbolic listing as

```
5)   TRA   IA
```

Subsequent assign constants are compiled with word 1 zero.

The first fixed point constant yields a CIT entry of the form

word	1	020000000000
word	2	462363000000
word	3	0YYYYY000000
word	4	000000000000

which appears in the symbolic listing as

```
2) OCT  0YYYYY000000
```

Subsequent fixed point constants are compiled with word 1 zero.

The first floating point constant yields a CIT entry of the form

word	1	030000000000
word	2	462363000000
word	3	PPPMMMMMMMMM
word	4	000000000000

which appears in the symbolic listing as

```
3) OCT  PPPMMMMMMMMM
```

Subsequent floating point constants are compiled with word 1 zero.

Universal constants are compiled for all programs, as certain subroutines assume that they be present. The first compiled CIT is

word	1	060000000000
word	2	462363000000
word	3	233000000000
word	4	000000000000

which appears in the symbolic listing as

```
6) OCT  233000000000
```

Subsequent compiled CIT's include word 1 zero and word 3.

```
000000077777
000000000000
000001000000
000000000000
```

These will appear in the symbolic listing as

```
6) OCT 233000000000
   OCT 000000077777
   OCT 000000000000
   OCT 000001000000
   OCT 000000000000
```

The last cell is used to store the contents of index register 4 whenever a transfer is to be made to a closed function or subroutine subprogram or to a library subroutine.

The first word of a format specification yields a CIT entry of the form

word	1	1000000KKKKK
word	2	222324000000
word	3	BBBBBBBBBBBB
word	4	000000000000

Subsequent format specification words are compiled with word 1 zero. When a 777777777777 flag is encountered, marking the end of a specification, the next word in the table, the symbol, is inserted in word 1. Format specifications are stored backwards in memory, i. e., a specification will give rise to a block of CIT's in which the symbol appears in the last compiled instruction.

These will appear in the symbolic listing as

```
   BCD 1MENT)
   BCD 1 STATE
   BCD 1FORMAT
8)K BCD 1  (12H
```

When all constants have been compiled, the partially filled final CIT record, if any, is written off on tape 3, an end file is written to mark the end of the CIT file, tape 3 is rewound, and control passes to 1TOCS to bring in the next record of the compiler.

SECTION PRE-SIX

The purpose of section pre-six is to complete the CIT file by compiling certain preparatory instructions inserted at the beginning of the program, adding to the CIT file Hollerith data B), and computing and adding to the CIT file initialization addend constants 9). These instructions are required for the use of the FUNCTION, SUBROUTINE, and CALL statements, and by the BSS loader.

Pre-six reads into memory all tables required by section 6, including FORSUB, SIZ, END, SUBDEF, COMMON, HOLARG (for pre-six), EIFN, EQUIV, and CLOSUB tables.

Each word in CLOSUB table yields a CIT entry of the form

word	1	AAAAAAAAAAAA
word	2	222324000000
word	3	AAAAAAAAAAAA
word	4	000000000000

which appear in the symbolic listing as

NAME BCD 1NAME

where NAME is a symbolic entry point to a subprogram mentioned in a CALL or arithmetic statement. This is the transfer vector. During the Section 6 assembly, any TSX NAME, 4 will be assembled as a TSX to the BCD name in the Transfer Vector, but during the loading process, the BSS loader will replace this BCD word by a trap transfer to wherever the required subprogram has been loaded.

The length of the transfer vector is inserted in 8L decrement of the program card. The relocatable entry point is tentatively assigned the location immediately following the transfer vector, and is inserted in 7R address.

Three cells are reserved to store the contents of index registers so that later they may be restored to their original states before control is returned to the calling program.

The first cell yields a CIT entry of the form

word	1	536000000000
word	2	306351000000
word	3	000000000000
word	4	000000000000

Subsequent storage cell instructions are compiled with word 1 zero. These will appear in the symbolic listing as $ HTR
The name of the subprogram yields a CIT entry of the form

word	1	000000000000
word	2	222324000000
word	3	NNNNNNNNNNNN
word	4	000000000000

This will appear in the symbolic listing as

```
BCD 1NAME
```

Three index saving instructions are compiled to store the contents of the index registers upon entry into the subroutine. The first of these is the entry point into the subroutine. These yield CIT entries of the form

```
word   1   000000000000
word   2   626724000000
word   3   536000000000
word   4   00000R00000T
```

These will appear in the third file as

```
SXD  $+R, T
```

If the subprogram being compiled has an argument list, the CIT file is searched for instructions referring to any of these arguments. The arguments referred to throughout the subprogram are merely dummies for the actual variables to be used at object time, the latter being listed in each calling sequence to this program. Instructions must therefore be compiled to initialize the addresses of all argument occurrences according to the requirements of any such calling sequence.

CIT records are brought into memory one at a time from tape 3 and when completely scanned, are replaced by the next subsequent record. A count of the memory location of each CIT relative to the first program instruction ($$ if an IFN is not assigned) is maintained.

A CIT is scanned first for its op-code (18 leading bits word 2). If the op-code is SYN, the CIT is ignored. If the op-code is BSS, the relative counter is increased by the length of the block reserved. Otherwise the relative counter is increased by 1. If the op-code is BCD, BCI, OCT or QPR the address field is not examined. If the op-code is anything else, the symbolic address is compared with each of the entries in the SUBDEF table. If the symbolic address is an argument, a two word entry is made in an initialization table. The table format is

```
word   1   0NNNNNN0MMMMM
word   2        n        τ              (CIT word 4)
```

Where N is the argument number, M is the corresponding relative count.

After the entire CIT tape has been scanned, it is rewound in preparation for its second pass.

Initialization CIT's are compiled using the information in the table just prepared.

Each argument yields a CIT of the form

```
word   1   000000000000
word   2   234321000000
word   3   000000000000
word   4   0NNNNN000004
```

where N is the argument number (the relative order of the dummy variable name in the SUBDEF table). This will appear in the symbolic listing as

```
CLA   N,4
```

The initialization table is searched for any entry containing a corresponding argument number N. If such an entry is found, the relative address is compared with the prior relative address. If the relative address differs from the prior relative address, an addend is required. The 9) table is searched for such an addend, and if not redundant this addend is inserted in the 9) table. The table format is

```
0000000AAAAA
```

where A is the required addend.

The addend yields a CIT of the form

```
word   1   000000000000
word   2   212424000000
word   3   110000000000
word   4   0BBBBB000000
```

where B is the relative position of this addend in the 9) table. This will appear in the symbolic listing as

```
ADD   9)+B
```

The initialization yields a CIT of the form

```
word   1   000000000000
word   2   626321000000
word   3   535360000000
word   4   0MMMMM000000
```

where M is the relative count of the instruction to be initialized. This will appear in the symbolic listing as

```
STA   $$+M
```

The initial symbolic location is $$ only if no internal formula number is assigned to the first CIT following the prologue. Each entry in the initialization table of which the argument number is the same N will yield one or both of the last two CIT's. Each argument number N will yield a string of CIT's consisting of a CLA N, 4 followed by as many ADD 9)+B and STA $$+M as is required.

For example, instructions for initializing an argument occuring as, say, MPY ARG + 5 might be:

```
CLA   2,4
ADD   9)+2
STA   3A+201
```

given that the argument is 2nd in the argument list, that 9) +2 contains the constant 5, that 3A is the location attached to the first instruction of the program, and that MPY ARG+5 is the 201st instruction of the program.

The prologue CIT's are written on tape 4 as records become filled. The final partialprologue record, if any, is written on tape 4.

For the second pass, CIT records are brought into memory, one at a time from tape 3, and when completely scanned, are replaced by the next subsequent record. A count of the memory location of each CIT relative to the first program instruction ($$ if an IFN is not assigned) is maintained. If no IFN is assigned to the first program instruction, the word 53536000000 is inserted in word 1 of the first CIT following the prologue.

A CIT is scanned first for its op-code. If the op-code is SXQ, it has been compiled by section 1 immediately prior to a TSX to double precision routine, to prevent deletion by section 5. The op-code is replaced by 626724000000 in word 2 (SXD). If the op-code is QXD, it has been compiled by section 1 immediately prior to a return from a subroutine, to prevent deletion by section 5. The op-code is replaced by 436724000000 in word 2 (LXD). If the op-code is QPR, it has been compiled by section 1 as the return instruction in a subroutine. It, followed by TRA NA, acts as an end of path of flow signal for section 4. The op-code is replaced by 635121000000 in word 2 (TRA). The symbolic location NA of this CIT is required only by the following instruction, which will be deleted. This internal formula number is deleted in word 1. A flag is set to delete the following instruction (TRA NA), which will not be copied into the tape 4 buffer. All other CIT's are written on tape 4 as records become filled.

After the end of tape 3 CIT file is encountered, the 9) constants are added to the CIT file. The first 9) constant yields a CIT of the form

word	1	110000000000
word	2	432363000000
word	3	0000000AAAAA
word	4	000000000000

where A is the value of the addend. This will appear in the symbolic listing as

```
9) OCT  A
```

Subsequent addend constants are compiled with word 1 zero.

The first word in the HOLARG table yields a CIT entry of the form

word	1	130000000000
word	2	222324000000
word	3	HHHHHHHHHHHH
word	4	000000000000

which will appear in the symbolic listing as

```
B) BCD  1HHHHHH
```

Subsequent HOLARG table words are compiled with word 1 zero.

An end of argument flag yields a CIT entry of the form

word	1	000000000000
word	2	432363000000
word	3	777777777777
word	4	000000000000

which will appear in the symbolic listing as

OCT 777777777777

The complete B) area will include but one symbolic location, but as many sets of one or more BCD words followed by the OCT flag as there are Hollerith arguments.

The final partial CIT record, if any, is written on tape 4, and the end of the CIT file is marked.

The compiled instructions now correspond exactly to the symbolic listing. Tape 3 is rewound to prepare for binary output.

If a CLOSUB table exists, it is saved as second file on tape 4 for later use by section 6 record P. If an EIFN table exists, it is saved as first record on tape 3 for later use by section 6 record N. Control is passed to 1TOCS to bring in the next record of the compiler.

SECTION SIX

Since the object program is symbolically complete, all that remains is to assemble the compiled instructions, producing a relocatable binary program ready for loading and running, and a listing of certain information concerning the program being compiled. Section 6 is primarily an assembler, differing little from any standard assembler. It builds a table of symbol names and (relocatable) locations, translates BCD operation codes to binary instructions, replaces symbolic locations with (relocatable) locations, and assembles the binary operation code, decrement, tag, and address into one word which shall occupy one location in memory during object time. In addition, options are available to include in the binary deck library subroutines for use at object time; to punch on line a row-binary deck, preceded by the BSS loader if a main program; to punch on-line a column-binary deck (709 only); to produce a third file of SAP-like symbolic listing of the compiled program; to print on line a listing of the source deck, the storage map, and, if produced, the SAP-like symbolic listing of the compiled program; to produce a binary symbol table (32K 709 only).

The discussion which follows is a general introduction to the logic of the assembler, followed by a detailed discussion of each of its processors.

5 Prime - adds to the CIT file, which on tape 3 now includes all executable instructions in the source program, certain constants and program data appearing in the symbolic listing: assign constants, fixed point constants, floating point constants, universal constants, and FORMAT BCD statements, using information in the ASSIGN, FIXCON, FLOCON, and FORMAT tables.

6A - completes the CIT file. It uses information in the CLOSUB, SUBDEF, and HOL-ARG tables, and scans during prefirst pass the entire CIT file on tape 3 for those instructions referring to arguments which require initialization. It writes the transfer vector, and if a subprogram, prolog, and initialization on tape 4; copies during presecond pass the CIT file from tape 3 to tape 4, changing certain pseudo op codes used internally in FORTRAN to machine op-codes; and adds to the end of the CIT file Hollerith arguments, and initilization addend constants. It also reads into memory tables required by section 6.

6B - is a common binary search routine which remains in memory for use of subsequent processing.

6C-E - builds that portion of the dictionary which is defined by COMMON, EQUIVALENCE, DIMENSION, CALL, SUBROUTINE, and FUNCTION statements, and any statement referring to a library subprogram. It uses information in the COMMON, EQUIV, SIZ, SUBDEF, and CLOSUB tables. Variables appearing in COMMON, EQUIVALENCE and DIMENSION statements are mapped. The map appears following the source program on tape 2. The order of processing differs between the 704 and the 709; the latter having a more thorough diagnostic procedure. The variable names are entered into the DEV table, while the locations in upper memory (which may be relocated later) are entered into the DEA table.

6F - adds to the dictionary those names which arise from arithmetic statement function definitions from the FORSUB table. Locations are tentatively set to zero, to be inserted into DEA later.

6G - is the first compiler pass over the CIT file. During this pass external variables not appearing in COMMON, DIMENSION or EQUIVALENCE statements are inserted in the TEV table, the location being determined by the order of their appearance in TEV, internal formula numbers, and internal symbols appearing in the symbolic listing are defined by the then current contents of a program counter and inserted in IFN and TIV tables, respectively. SYNs to names in the transfer vector, format statement symbols, subsidiary internal formula numbers, location symbols for subroutines to compute relcons, for section 5 LXD and SXD instructions,* (program counter), and the special symbols $ and $$, if they have been defined prior to the appearance of the SYN, are defined. Blocks of required length of storage are reserved for non subscripted variables in TEV, and for internal symbols not appearing in the symbolic listing. Reference is made to the COMPILED INSTRUCTION file, and the DEV table prepared earlier in section 6.

At this point, every symbol appearing in the CIT file has been entered into one of the tables DEV, IFN, TEV or TIV.

6H - assigns locations for names of arithmetic statement function subroutines, and maps them. It uses information in the FORSUB table and the IFN table prepared earlier in section 6. The location assigned to the internal formula number corresponding to the subroutine name is inserted into FORSUB. The name, internal formula number, and location of each subroutine is mapped.

6I - maps external formula numbers and corresponding internal formula numbers, with the relative locations assigned to the internal formula numbers. It uses information in the EIFN table, and the IFN table prepared earlier in section 6.

6J - relocates storage not in common downwards adjacent to program constants. The limits of storage not used in the program are mapped. It operates upon the DEA, TEV and TIV tables prepared earlier in section 6.

6K - maps the transfer vector, program variables not in common, and internal symbols. It uses information in the DEV, DEA, TEV, and TIV tables.

6L - writes the program card on the binary output tape.

6M - is a table of operation codes which remains in memory during the processing of the second pass over the CIT file.

6N - is the second compiler pass over the CIT file, now on tape 4. During this pass each CIT is converted to a binary machine instruction. The location for each symbol is found in one of the tables DEA, IFN, TEV or TIV; this is combined with the relative address and relocation bits computed. The relocation bits are inserted in 8 row, while the address, tag, decrement, and the binary op code are combined to form an instruction which is inserted in the next available 7-12 row of a card image. Column binary bits, word count and load address (relative to zero) are inserted in 9L, the checksum computed and inserted in 9R. The card images are written on tape 3. During this pass, SYNs are defined, and the locations assigned to internal formula numbers and internal symbols appearing in the location field are checked against the current program counter for inconsistent definition. If the symbol *, an external symbol in DEV (name in transfer vector), or the special symbol $ or $$, appears in the location field, it is ignored. At this point the entire binary output for the source program is complete.

6P - begins processing the options which the programmer has instructed the FORTRAN compiler to provide. If a library search is required (sense switch 5 down) and a transfer vector exists, every program card in the library file on tape 1 is scanned for a primary entry point the name of which is in the transfer vector. If at least one such entry point exists, the names of all entry points on the program card are added to the LIBF table, and any matching names are deleted from the transfer vector.

The library subroutine transfer vector is then examined to determine if any entry points exist which are not yet in the LIBF table or in the object program transfer vector. Such names are added to the object program transfer vector, and the subroutine program card and the entire binary subroutine are added to the binary output tape 3. The library search is discontinued when the entire transfer vector is exhausted, or no subroutines are found in one complete pass over the library file. The names of entry points in the LIBF table and the names of entry points remaining in the transfer vector are written in the storage map, tape 2, and the storage map is complete. If the object program is not a subprogram, a transfer card is written on the binary output tape 3, and the binary output is complete. At this point the compiled output of FORTRAN is complete.

6Q - provides binary cards on line if sense switch 1 is down. If sense switch 4 is up, and the object program is not a subprogram, the BSS loader is punched on line. Each card image is read from tape 3, the column binary bits deleted, and the row binary card punched on line. If sense switch 4 is down, each card image is read from tape 3, rotated to column binary image, and punched on line.

6R - provides a machine language listing if sense switch 2 is down. An additional pass over the CIT tape 4 is made, each CIT being converted into the standard form, SYMBOL OPC ADDRESS + RA, TAG, DECREMENT and written in three columns on tape 2 following the storage map.

6S - provides on line listings if sense switch 3 is down. The entire contents of the BCD output tape 2, source program, storage map, and machine language listing, if any, are converted to card images, one record at a time, and printed on line one line at a time.

At the end of section 6 the FORTRAN compiler has completed processing of the source program. The results of the FORTRAN compilation are on two tapes: BCD tape 2, the source program, storage map, and, if requested, the machine language listing; and binary tape 3, the program card, object program, if requested, the library program cards and subroutines, and, if a main program, the transfer card. If on line output has been requested, the row- or column-binary cards can be found in the on line punch, and the listing in the on line printer.

The job is returned to the FORTRAN monitor.

DETAILED DISCUSSION

In the discussion that follows, portions of the assembler are labeled as to parts. 704 FORTRAN II has one record for each part. 709 FORTRAN II has one or more parts in a record.

Part A
Part A is Presix coding and has been discussed in Chapter IX.

Part B
Part B is a common binary search routine which remains in memory for use of subsequent parts. The maximum table length which can be searched by this routine is 16383 words, which is the effective limit to the length of any table which must be searched.

Part C
Part C builds that portion of the dictionary which is defined by COMMON, DIMENSION, EQUIVALENCE, CALL, SUBROUTINE, and FUNCTION statements, and any statement referring to a library subprogram, such as PRINT or X = SQRTF(B). The names of variables, dummy variables (arguments), or subroutine or subprogram entry points are entered into the DEV table, while the relocatable address assigned to each is entered into the associated DEA table.

First processed are variable names appearing both in COMMON and EQUIVALENCE statements. A variable name is selected from the EQUIV table. It is compared with the names appearing in the COMMON table. If it appears in both, the entire sentence in the EQUIV table in which this variable appears is assigned to upper memory.

An equivalence sentence, assembled by Section I prime (see chapter III) contains all variable names, the relative locations of which to each other have been fixed by EQUIVALENCE statements. The sentence contains no redundancies or inconsistancies. The sentence is made up of two-word entries, the BCD variable name, and the relative location (subscript) to each other. The end of each sentence is marked by a flag (negative sign) in the final subscript.

The equivalence sentence is scanned for the greatest subscript. The current value of the location counter, initially at -206 in the 704, -207 in the 709, is reduced by the greatest subscript. This is the base from which the location assigned to each of the variable names is computed. The equivalence sentence is scanned again for any variables which are names of arrays. If a variable appears in the SIZ table, the overhang of the array length over the base location(array length-subscript) is computed, and the maximum of these is found. The equivalence sentence is scanned again. Each subscript is added to the base address, in effect creating an array stored backwards in memory, and the variable or array name is entered into DEV with its corresponding location in DEA. The array name with the greatest subscript will be assigned the value of the location counter before it was reduced, in effect locating the most precedent array name in the first available memory location. The value of the location counter is reduced by the maximum overhang, which is not less than 1, reserving memory for the overlapping array extending farthest into memory, and reserving for the next variable name the next lower cell.

Suppose there are common symbols E, D, X, which are related by EQUIVALENCE (E(5), D(2), X), and that E and D occur in dimension statements giving their total size as E(6) and D(5), X being a nonsubscripted variable. The first variable to be defined is the one with the largest element number in the equivalence group, E in this case, and the 1st element of E is given the highest free location, ie. LCTR. D and X are immediately defined by their equivalence relationship with E: $E(5) \equiv D(2) \equiv X$

or $D = E-3$
$= LCTR-3$

and $X = E-4$
$= LCTR-4$

It must also be determined how much space these variables occupy. Since the array E has 6 elements, the last of these would be in LCTR-5, and similarly D has 5 elements, the last of which would be in LCTR-7. Clearly then, the first free location is the one following array D, namely LCTR-8, which then becomes the new LCTR for the next set of assignments.

After all equivalence sentences in common have been assigned, storage is assigned for all other variables appearing in COMMON statements. The COMMON table, assembled by section 1 (see chapter II), is made up of one-word entries, the BCD name of a variable appearing in a COMMON statement. Each variable name is checked against DEV to determine if it had appeared in an equivalence sentence. If it is not so redundant, it is entered into DEV with the contents of the location counter as the corresponding location in DEA. The SIZ table is checked to determine if this is an array name, and the value of the location counter is reduced by the length of this array; or if not an array, by 1. This, in effect, creates an array stored backwards in memory, reserving for the next variable name the next lower cell.

When all of common has been assigned, the current value of the location counter, the cell next below the last cell in common, is entered into the program card 8R address break.

Next to be processed are equivalence sentences not assigned to upper storage. The first symbol of each equivalence sentence is checked against DEV to determine if any symbol in this sentence had appeared in a COMMON statement. If it is not so redundant, the entire sentence is assigned storage locations, identically as described above. The array name with the greatest subscript in the first equivalence sentence will be assigned the location stored in the common break, the cell next below the last cell in common. This, and all subsequent storage assignments later will be relocated downwards in memory.

At this point processing in the 704 and 709 differ in order of tables processed. 709 processing will be described, as this results in more accurate diagnostic analysis of the source deck.

Next to be processed, in 709 FORTRAN II, is the SUBDEF table. If this program is a FORTRAN subprogram, defined by a SUBROUTINE or FUNCTION statement, the name of the subprogram and the argument list are assembled

into the SUBDEF table by section 1 (see chapter II). Each entry is a one-word BCD name of a dummy variable used as an argument. Each argument name is compared with the subprogram name. If it is multiply defined, a diagnostic message results. Entry is made into DEV to prevent assignment of a storage location for this dummy variable if it appears in a DIMENSION statement not in common, or as the symbolic address (word 3) of a CIT. The corresponding address in DEA is the flag 77777. If this dummy name is already in DEV, it has appeared in a COMMON or EQUIVALENCE statement, and a diagnostic message results. The name of the subprogram is not entered into DEV, as it may properly appear in the source program in a COMMON, DIMENSION, or EQUIVALENCE statement, and as a subscripted or nonsubscripted variable.

The SIZ table, assembled by section I prime (see chapter III), is made up of two word entries, the BCD name of the array, and the length of the array (the product of its dimensions as stated in a DIMENSION statement). Each array name in the SIZ table is checked against the DEV table to determine if it has appeared in a COMMON or EQUIVALENCE sentence or is a dummy variable name of an argument. If it is not so redundant, it is entered into DEV with the current contents of the location counter as the corresponding location in DEA. The value of the location counter is reduced by the length of this array. This, in effect, creates an array stored backwards in memory, reserving for the next variable name the next lower cell. A dummy variable name of an argument may appear in a DIMENSION statement in order that a proper relative address may be computed for reference to a specific element in an array, but no storage will be allocated to this dummy variable.

The storage for variables appearing in common statements is now mapped. The variable name, right adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeros suppressed, inserted in the third word; and the octal location, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings , each line as completed, and the final partial line if any, are written on tape 2 immediately following the internal end of file marking the end of the source program listing.

Next to be processed is the transfer vector. If the source program refers to other subprograms through a CALL statement or an arithmetic statement in which a function name appears, or if a library subroutine is called, section 1 (see chapter II) assembles the BCD name of the entry point to each such subprogram as one- word entries in the CLOSUB table. The transfer vector, made up of N such names, occupies (relocatable) storage locations 0 thru N-1 of the object program. Each subprogram name is entered into DEV with the corresponding lower storage locations entered into DEA. If the name is already in DEV, it has appeared in a COMMON, EQUIVALENCE, or DIMENSION statement, and a diagnostic message results.

Finally, the names of arithmetic statement functions are processed. If such a statement appears in the source program, section 1 (see chapter II) assembles the BCD name of the function so defined, and the internal formula

number assigned to the subroutine, in a two word FORSUB entry. Each name is entered into the DEV table with location zero (to be entered later) entered into DEA. If the name is already in DEV, it has appeared in a COMMON, EQUIVALENCE, or DIMENSION statement, or has been referred to in a CALL statement or an arithmetic statement including an argument list with the terminal F omitted from the name, or as a dummy variable name, and a diagnostic message results. The improper use of the name with the terminal F omitted and with no argument list will compile; however improperly.

On the 704, the processing is similar, but in a different order. Hence the diagnostic proceedure is not as comprehensive. After common storage has been assigned, part D includes lower storage equivalence assignment of variables in EQUIVALENCE statements, of arrays not in EQUIVALENCE statements, and of names in the transfer vector. Part E maps common storage. Part F enters subroutine arguments and arithmetic statement function definitions into DEV. There is no diagnostic procedure for multiply defined names of dummy variables (arguments). Each of these is a separate record.

The DEV and DEA tables are now complete. All other variable names in the source program are nonsubscripted, requiring one storage location each.

Control is passed to 1 - CS, to bring in the next record.

Part G
Part G includes the first pass over the complete CIT file, to define all internal formula numbers, source program symbols not in DEV, and internal symbols.

The DEA table is moved up in memory and packed against the end of DEV. The IFN table will share memory with the DEA table, the former occupying the decrement portion of each word, while the latter occupies the address. The TEV table will follow the longer of the two.

CIT records are brought into memory from tape 4, and are replaced with the next subsequent record when completely scanned.

Each CIT is scanned first for its op code. If it is OCT or BCD the address portion is ignored.

For other codes the symbolic address is scanned next. If the address is an internal formula number, the address is ignored. A SYN to an IFN is undefined. If the address is a subsidiary internal formula number (nAm), the symbol is assembled into TIV form (see chapter X) and TIV is searched to define a possible SYN to this symbol. If it is not in TIV, it is entered, undefined. If the address is *, the contents of the program counter are used to define a possible SYN to this symbol.

If the address is

2)	Fixed point constant
3)	Floating point constant
5)	Assign constant

6)	Universal constant
8)N	Format specification word
9)	Initilization addend constant
B)	Hollerith subroutine argument

it is in the symbolic listing, and the address is ignored. A SYN to one of these symbols is undefined.

If the address is

1)N	Arithmetic eraseable
4)N	Arithmetic statement function argument storage
7)N	Arithmetic statement function index register eraseable
C)N	Index register eraseable

it is not in the symbolic listing, and is entered into TIV with greatest level of storage (decrement of word 4 CIT) as the address. A SYN to one of these symbols is undefined.

If the address is

A)N	Location symbol for subroutine to compute relative constants
D)N	Location symbol for a section 5 LXD instruction
E)N	Location symbol for a section 5 SXD instruction

it is in the symbolic listing, but TIV is searched to define a possible SYN to one of these symbols.

If the address is $ or $$, the location assigned to each of these is used to define a possible SYN. If the address is an external variable, DEV and TEV are searched to define a possible SYN to one of these symbols. If this variable name is not in DEV or TEV, it is entered into TEV, the location to be defined later.

The opcode again is scanned for SYN. The symbol D)N or E)N in the symbolic location can be synonymous with another symbol D)N or E)N, compiled by section 5. If the SYN is undefined, a diagnostic message results. For all op-codes other than BSS or SYN, the location counter is bumped by 1. If it is BSS, the length of block reserved is assumed to be zero. If it is SYN, no location is reserved.

Next to be scanned is the symbolic location. If the location symbol is an internal formula number, the contents of the program counter are entered into the IFN table (decrement portion of the joint IFN-DEA table), ordered as to internal formula numbers. The test for an internal formula number is such that it may not extend over more than 12 bits in the decrement field, a maximum of 4095. If any internal number is greater, it will appear to be an internal symbol, and will miscompile. No diagnostic message results. If the location symbol is a subsidiary internal formula number (nAm), TIV is searched to determine if there had been a prior reference to the symbol. If such a reference had been made, the contents of the program counter are entered into TIV to define this symbol. If no prior reference had been made, the symbol remains undefined. This is to optimize entries into the TIV table. If the reference to the subsidiary internal formula number is prior to the appearance of the number in the location field, it will have been entered into TIV, and defined in pass 1. If such reference is subsequent to such appearance, the TIV entry will be made, but the symbol will remain undefined until pass 2. During pass 2, this symbol will be defined prior to such subsequent reference. Hence, any subsidiary internal formula number to which a reference is made will eventually appear defined in TIV, while such a symbol to which no reference is made will not be entered into TIV. If the location symbol is *, it is ignored. For all other internal symbols appearing in the symbolic listing, a TIV entry is made, the contents of the program counter defining this symbol. If the location symbol is $ or $$, each of these is defined by the contents of the program counter. If the location symbol is an external symbol (transfer vector name), it is ignored.

At the end of the first pass over the complete CIT tape, all symbols appearing in compiled instructions have been entered into one of the tables, DEV, IFN, TEV or TIV. The upper location counter is one cell below the lowest cell reserved for a DEV entry. The location counter is reduced by the length of the TEV table, and each variable in TEV is implicitly defined as the current contents of the location counter plus its ordered location in the TEV table. Later, these locations will be relocated downwards in memory.

Assignment of storage locations for eraseable cells in TIV is made next. Each TIV entry is examined to determine if it is an eraseable cell (1)N, 4)N, 7)N, C)N). If it is, the location counter is reduced by the largest value of the block required, the address portion of the TIV entry, and this location defines the symbol. This, in effect, creates an array stored forwards in memory. The location counter is reduced by one moreto reserve the next lower cell for the next symbol. The symbol 4), eraseable for library subroutines, is defined as the location of top of memory, 77777.

The storage assignments at this point are as in the following diagram.

Control is passed to 1 - CS, to bring in the next record.

Location Symbol		Table entries	
NAME	TRANSFER VECTOR	DEV	relocatable zero
$	PROLOG		subprograms only
	INITIALIZATION		
$$ nA nAm D)N E)N	OBJECT PROGRAM	IFN TIV TIV TIV	
nA	ARITHMETIC SUBROUTINES	(Name in DEV) IFN	
A)N	RELCON SUBROUTINES	TIV	
5) 2) 3) 6) 8)N B)	PROGRAM CONSTANTS	TIV	
	NOT ASSIGNED		end of symbolic listing contents of program counter contents of location counter
7) 4)N 1) C)N	ERASEABLE STORAGE	TIV	
NAME	NON SUBSCRIPTED VARIABLES	TEV	
NAME	DIMENSION VARIABLES	DEV	
NAME	DIMENSION EQUIV VARIABLES	DEV	common break
NAME	COMMON DIMENSION VAR.	DEV	
NAME	COMMON DIM. EQUIV. VAR.	DEV	-207 (709) -206 (704)
4)	LIBRARY SUBROUTINE ERASEABLE	TIV	top of memory

Note: argument dummy names (in subprograms) are entered into DEV, flagged 77777 in DEA

Part H

Part H assigns locations for arithmetic statement function subroutines and maps them. The DEV table is scanned for the name of each subroutine (word 1 of each FORSUB table entry). If it is not found, a machine error has occurred, and a diagnostic message results. The location of the internal formula number assigned to this subroutine name (decrement of word 2 of FORSUB table entry) is found in the IFN table, and inserted in the address of word 2 of the FORSUB table entry, and, to define this symbolic location, in the DEA table.

The location of each subroutine is now mapped. The subroutine name, right adjusted, is inserted in the second word of a tetrad; the decimal internal formula number, right adjusted, inserted in the third word; and the octal location of this internal formula number, right adjusted with leading zeroes included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line if any are written on tape 2, following the mapping of common storage assignment (if any).

Control is passed to 1 to CS, to bring in the next record.

Part I

Part I maps external formula numbers with corresponding internal formula numbers and relative locations.

Each decimal external formula number (address portion of one-word entry) in EIFN table, right adjusted, is inserted in the second word of a tetrad; the decimal internal formula number (decrement portion of entry), right adjusted, is inserted in the third word; and the octal location of this internal formula number, found in the IFN table, right adjusted with leading zeroes included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line if any are written on tape 2, following the mapping of arithmetic statement function subroutines (if any).

Control is passed to 1 - CS, to bring in the next record.

Part J

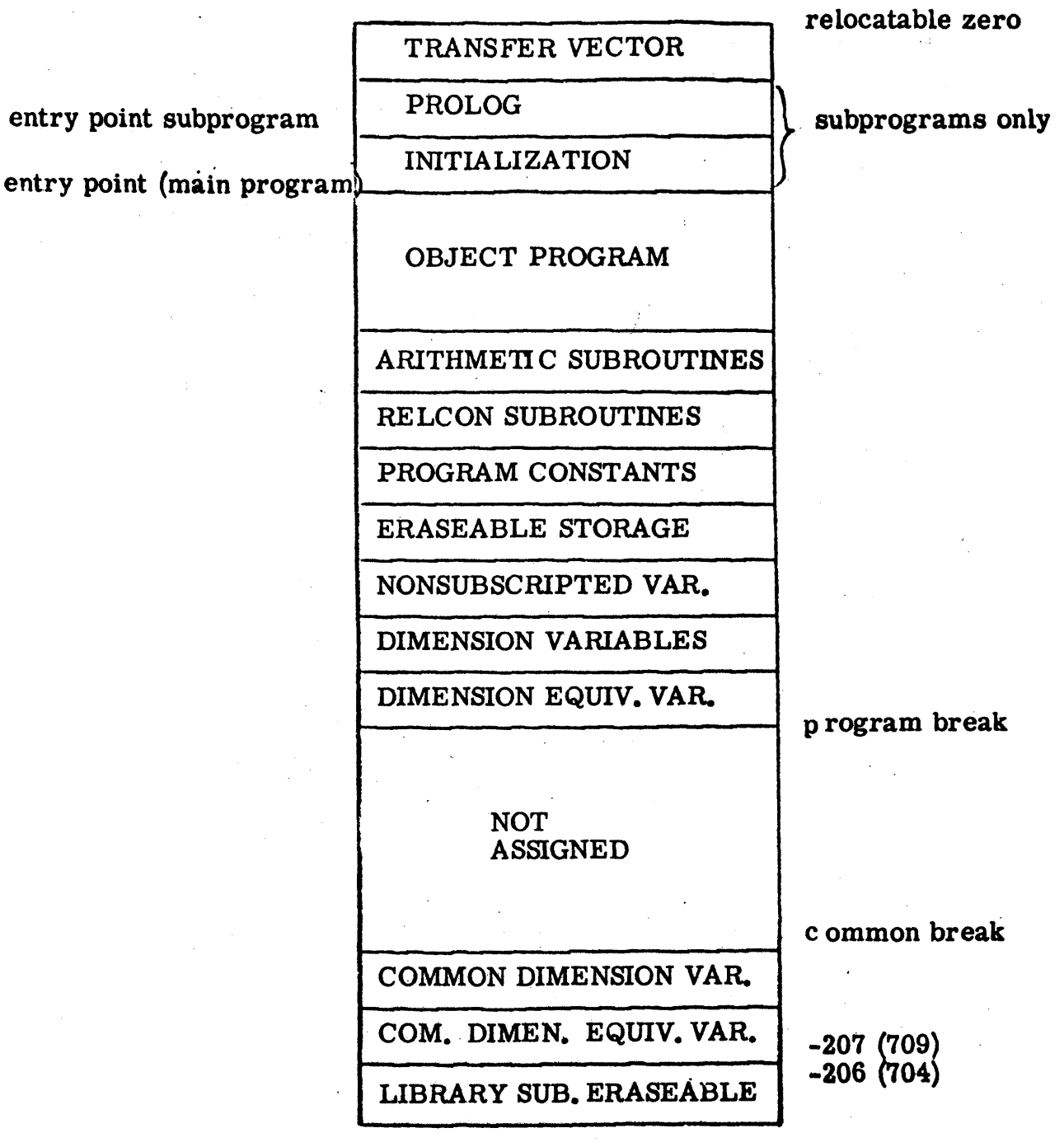
Part J relocates storage not in common downwards packed against program constants.

The length of unassigned memory is computed (contents of location counter less contents of program counter, plus one), and is the extent of relocation. The position of the program break is computed (location of common break less contents of the location counter, number of variables to be relocated, added to contents of program counter), and inserted in the program card 8L address.

Each location in DEA is compared against the common break (highest cell in storage to be relocated) and against the program break (lowest cell in storage to be relocated). If it is not in common, a transfer vector name, a subprogram

argument dummy variable (flagged 77777), or an arithmetic subroutine, the location is reduced by the extent of relocation. The base location for TEV is so relocated; in effect, relocating each variable in TEV. Each location in TIV is compared against the common break and the program break. If it is not 4) (location 77777), program data in the symbolic listing, or an instruction location symbol, it is an eraseable cell and is so relocated.

The final storage assignments are as in the following diagram.


relocatable zero
TRANSFER VECTOR
PROLOG
INITIALIZATION
entry point subprogram
entry point (main program)
subprograms only
OBJECT PROGRAM
ARITHMETIC SUBROUTINES
RELCON SUBROUTINES
PROGRAM CONSTANTS
ERASEABLE STORAGE
NONSUBSCRIPTED VAR.
DIMENSION VARIABLES
DIMENSION EQUIV. VAR.
program break
NOT
ASSIGNED
common break
COMMON DIMENSION VAR.
COM. DIMEN. EQUIV. VAR.
-207 (709)
-206 (704)
LIBRARY SUB. ERASEABLE

The limits of storage not used by program (program break and common break), converted to decimal, right adjusted with leading zeroes suppressed, are inserted in the third word of a tetrad; converted to octal, right adjusted with leading zeroes included, inserted in the fourth word. The first and second word of this tetrad are blank. The title, column headings, and this line are written on tape 2, following the mapping of external-internal formula numbers (if any).

Part K
On the 704, part K is a separate record.
Part K maps the transfer vector, program variables not in common, and internal symbols. The number of entries in the transfer vector is one location greater than that of the last name in the transfer vector. Each location in DEA is compared against the location of the first instruction following the transfer vector, and if in the transfer vector, the corresponding transfer vector name in DEV, right adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeroes suppressed, inserted in the third word; and the octal location, right adjusted with leading zeroes included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line if any, are written on tape 2 following the mapping of the storage limits.

If any arithmetic subroutines exist, the location following them is the first location in which a variable may appear. If not, the location following the transfer vector is this location. Each location in DEA is compared against the first location following either the transfer vector or arithmetic subroutines, and against the common break. If it has not been listed previously as a transfer vector or arithmetic subroutine name, as a variable in common, or is not a subprogram argument dummy variable name, it is a subscripted variable not in common, and the corresponding name in DEV, right adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeroes suppressed, inserted in the third word; and the octal location, right adjusted with leading zeroes included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line, if any, are written on tape 2 following the mapping of the transfer vector (if any).

Each entry in TEV (nonsubscripted variable not in common), right adjusted, is inserted in the second word of a tetrad; the decimal location, the sum of base location for TEV and the relative location of this variable in TEV, right adjusted with leading zeroes suppressed, inserted in the third word; and the octal location, right adjusted with leading zeroes included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line, if any, are written on tape 2 following the mapping of the subscripted variables not in common (if any).

Each entry in TIV is then mapped. A TIV entry consists of a symbol in bits S, 1, 2, 3; bits 4 and 5 zero; sub symbol, if any, in bits 6-20; and the location in bits 21-35. A subsidiary internal formula number consists of bits S, 1, 2, 3, 20 zero, the internal formula number in bits 4-14 (maximum size 2047), the subsidiary number in bits 15-19; and the location in bits 21-35.

If the TIV entry is a sub internal formula number, it is ignored. If it is an internal symbol for a storage cell, an alpha numeric character from the set 1 through 9, A through E is assigned to the 4-bit pseudo symbol, followed by a right parenthesis. The 15 bit subsymbol, if any, is converted five bits at a time to 3 alpha numeric characters from the set 1 through 9, A through W. The pseudo symbol, left adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeroes suppressed, inserted in the third word; and the octal location, right adjusted with leading zeroes included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line, if any, are written on tape 2 following the mapping of nonsubscripted variables not in common (if any).

Part L
On the 704, Part L is a separate record.
Part L writes the program card on binary output tape 3.
Program card 9L includes a 4 punch in the prefix and a word count of 4 in the decrement.
8L contains the length of transfer vector in the decrement and program break in the address.
8R contains the common break in the address.
7L contains the BCD subprogram name, if any.
7R contains the entry point, relative to zero in the address.
The computed checksum of the card is inserted in 9R.

Column binary bits, 7-9 punch in column 1, not included in the checksum, are inserted in 9L, and the program card is written as the first record on binary output tape B3.

Part M
On the 704, Part M is a table of operation codes which is brought into memory as a separate record for the use of Part N which will follow.

Control is passed to 1 - CS to bring in the next record.

Part N
Part N is the second pass over the CIT tape to define each of the symbols used in each CIT, construct a binary instruction for each CIT, and write the compiled program on binary output tape 3.

CIT records are brought into memory from tape 4, and are replaced with the next subsequent record when completely scanned.

Relocation bit patterns are of three types. Type 00 indicates that address portion of the instruction is not relocatable. Type 010 indicates that the address portion is relocatable as data on the proper side of the program break. Type 011 indicates that the instruction is complement relocatable: the address refers to a cell in an array the base symbol of which is on the opposite side of the program break, and should be relocated as its base symbol would be. The decrement of an instruction is not relocatable in a FORTRAN object program.

The relocation bits are initially reset to not relocatable. First to be scanned is the opcode. If it is OCT or BCD, the address portion is not relocatable. For all other opcodes, the symbolic address is scanned next. If the symbolic address is zero, it is not relocatable. If the symbolic address is an internal formula number, the location is obtained from the IFN table. If the symbolic address is a subsidiary internal formula number or an internal symbol, the location is obtained from the TIV table. If the symbolic address is *, the location is the current contents of the program counter. If the symbolic address is $ or $$, the location is as assigned to either of these. If the symbolic address is an external symbol, the location is obtained from TEV or DEA. If any symbol has as yet not been defined, a diagnostic message results. For each of these, the address is tentatively set directly relocatable.

The opcode is again scanned. If it is SYN, the definition is saved to be checked. No binary output results. If it is BSS, the length of the block reserved is assumed to be zero. No binary output results. For all op-codes other than BCD, OCT, BSS or SYN, the binary machine code is found in the SOPR table. If the op-code is not found in the table, a diagnostic message results.

The relative address is added to the location for the symbolic address to determine the absolute address for the symbol. If negative, it is complemented. The base symbol (symbolic address) is examined to determine if both the base symbol and the absolute address are on the same side of the program break. If they are not, the address is set complement relocatable. The binary decrement, absolute tag, and absolute address are combined with the operation code. For BCD or OCT, the binary word (symbolic address) is used. The program counter is bumped one location.

The relocation bits are packed left adjusted against any prior relocation bits already in the 8 row of the card image. The binary instruction is inserted in the next available half row of the card image. When the card image is full, the word count is inserted in 9L decrement, the load address is inserted in 9L address, the checksum is computed and inserted in 9R, column binary bits added to 9L, the card is written on tape 3, and the load address is updated to the program counter for the next instruction.

For all CIT's the symbolic location is scanned. If it is a subsidiary internal formula number and is not in TIV, it has been omitted as no reference to it was made in the symbolic address, and it is ignored. If it is in TIV and is not yet defined, the reference to it was later in the CIT file than its appearance in the location field. It is here defined. If it is defined, the location assigned to this symbol is checked against the program counter. If it is inconsistent, a diagnostic message results. If the symbolic location is an internal formula number, it is checked for inconsistent definition. If the symbolic location is an internal symbol, it is the symbol for program data appearing in the symbolic listing, or the symbol assigned to a section 5 LXD or SXD instruction or a relcon subroutine. If the symbol appears in TIV, it is checked for inconsistent definition. If it does not appear in TIV, it is a machine error, but no diagnostic message will result. If it is $, $$, *, or an external symbol

in DEV (transfer vector name) it is ignored. No other external symbol in DEV or any in TEV should appear in a location field.

After the entire CIT file has been scanned, the final partial card image, if any, is written on tape 3. Processing is now complete, except for the transfer card, and for options.

Control is passed to 1 - CS to bring in the next record.

Part P

Part P processes the options which the programmer has instructed the FORTRAN compiler to provide. Available are the following options:

End Card Setting	Physical sense switch	704	709
1	1 (up)	punch cards on line	
2	2 (down)	add symbolic listing	
3	3 (down)	list on line	
4	4 (down)	N.A.	punch column binary
5	5 (down)	add library subroutines to binary output	
6	-	N.A.	add symbol table binary output

The transfer vector, which has been stored as one record following the CIT file on tape 4, is brought back into memory. End card setting and/or physical sense switch 5 is tested to determine if a library search is required. If the transfer vector is not empty, and if a library search is required, a flag for subroutines found in each pass over the library file on the FORTRAN system tape is reset.

The next record in the library file is read into memory. If 9L prefix has a 4 punch, it is a program card; if not the next record is read in.

After a program card has been found, the next record is brought into memory with rows 8 through 12 packed against the earlier card image. Row 9L prefix is again tested to determine if the program card continues over more than one card. When a card other than a program card is encountered, the tape is backspaced over the card image, and a consolidated program card exists in memory. The word count of the consolidated program card is found in the decrement of 9L, while the length of the subroutine transfer vector is found in the decrement of 8L. Each right row (entry point relative to zero corresponding to entry point name) is scanned to determine if it is flagged by a sign bit punch as a secondary entry point. If it is not so flagged, the left row (name of primary entry point) is compared against the transfer vector to determine if this subroutine is required to complete the object program. If no such name is found, the remainder of the subroutine in the library file is passed over to find the first program card of the following subroutine.

If a primary entry point to a subroutine is found in the transfer vector, the name is transferred from the transfer vector to a list of entry points to subroutines output from the library. A flag is set that at least one subroutine has been found on this pass over the library file.

The names of all entry points to subroutines output are added to the found list, and if any of these are in the transfer vector, they are deleted from the transfer vector.

The consolidated program card is converted back into card images, and written on tape 3 following the object program, or the last library subroutine output, and the next record read from the library file. If the library subroutine includes a transfer vector, each name in the subroutine transfer vector is compared against the found list and the object program transfer vector. If it is in neither, it is added to the object program transfer vector. The card image is written on tape 3 following the library program card, or last subroutine card. If the subroutine transfer vector extends over more than one card, this is repeated until the subroutine transfer vector is exhausted. If a program card is encountered before the subroutine transfer vector is exhausted, a diagnostic message results.

After the subroutine transfer vector is exhausted, the remaining cards in the library subroutine are copied from the library file to tape 3, until the next program card is encountered. If the object program transfer vector is exhausted, the search is completed. If not, the search continues until the end of the library file is sensed.

After the end of the library file, the flag for subroutines found is examined. If any subroutines have been found on this pass, the subroutine transfer vector may require another pass over the library file. If not, the search is completed. If the object program transfer vector is exhausted, the search is completed. If not, the system tape is backspaced to the beginning of the library file, the flag reset, and another pass over the library file is made.

After the library search is completed, the system tape is repositioned at the end of this record, and if any names of entry points to library subroutines are on the found list, these names are written on the storage map. Each BCD name is right adjusted and inserted in the second word of a pair. The first word is blank. The title, each line as completed, and the final partial line, if any, are written on tape 2 following the mapping of internal symbols.

After this mapping, or if the library search was not required, the transfer vector is examined to determine if any subprograms exist which are not library subroutines. Each BCD name remaining in the transfer vector is right adjusted and inserted in the second word of a pair. The first word is blank. The title, each line as completed, and the final partial line, if any, are written on tape 2 following the mapping of names of entry points to library subroutines (if any).

The storage map is now complete and marked with an end of file.

If the object program is not a subprogram, a transfer card is written on tape 3. The end of binary output is marked with an end of file, and the tape is rewound.

Part Q
On the 704, Part Q is a separate record.
End card setting and/or physical sense switch 1 is tested to determine if cards are required on line. If cards are required on line on the 709 end card setting and/or physical sense switch 4 is tested to determine if cards should be row binary or column binary.

If switch 4 is up, cards are to be row binary, and if the object program is not a subprogram, the BSS loader is punched on line. The column binary bits are deleted from 9L of each card image, and the card punched on line.

If switch 4 is down, cards are to be column binary, the column binary bits are added into the checksum, 9R, of each card image, the row binary image rotated to a column binary image, and the card punched on line.

If no column binary cards have been punched on line, sense light 1 is turned on to so flag the monitor.

Part R
On the 704, Part R is a separate record.
End card setting and/or physical sense switch 2 is tested to determine if a machine language listing is required. If it is so requested, sense light 2 is turned on to flag monitor that a third file exists on the BCD output tape. An additional pass is made over the CIT tape to accomplish this.

CIT records are brought into memory from tape 4, and are replaced with the next subsequent record when completely scanned.

First the symbolic location is processed. If the symbolic location is an internal formula number or a subsidiary internal formula number, the main number is converted to decimal, the character A appended, and the subsidiary number converted to decimal. The largest internal formula number which can be stored in TIV is 2047, and a subsidiary number can be one character only. Hence this symbol cannot exceed six characters. If the symbolic location is *, it is deleted. If the symbolic location is an internal symbol, a pseudo symbol is constructed (see chapter X) which can not exceed five characters. If the symbol is $, $$, or a transfer vector name, these characters are used. The BCD symbol, so constructed, right adjusted, is inserted in the third word of a hexad.

The BCD opcode, preceded and followed by a blank, is inserted in the first five characters of the fourth word.

If the opcode is BCD, and the symbolic address is a 777777777777 flag, the code is replaced by OCT, and processing continues as an octal symbolic address.

If the symbolic address is not a flag, the numeral 1 is inserted in the sixth character of the fourth word, and the six character BCD word in the fifth. The sixth word is blank.

If the opcode is OCT, the first bit is interpreted as a sign, inserted in the sixth character of the fourth word, and the 35 bit binary number, converted to 12 BCD octal digits, is inserted in fifth and sixth words.

For all opcodes other than BCD or OCT, the symbolic address is processed as follows. If the symbolic address is an internal formula number or a subsidiary internal formula number, the main number is converted to decimal, the character A appended, and the subsidiary number converted to decimal. The first BCD character of the internal formula number is inserted in the sixth character of the fourth word. The remaining BCD characters (five or fewer) of the symbol, followed by blanks, are saved. If the symbol address is an internal symbol, a pseudo symbol is constructed (see chapter X). The first BCD character of the pseudo symbol is inserted in the sixth character of the fourth word. The remaining BCD characters (four or fewer) of the pseudo symbol, followed by blanks are saved. If the symbolic address is an *, $, $$, or any external symbol, the first BCD character of the symbol is inserted in the sixth character of the fourth word. The remaining BCD characters (five or fewer) of the symbol followed by blanks are saved.

The remaining characters in the symbol (five or fewer) followed by blanks are examined one at a time for the first blank character. The non blank characters are packed left adjusted into the fifth word of the hexad, extending no further than the fifth character of the fifth word. The relative address is isolated from the decrement of the CIT word 4. If it exists, it is converted to five or fewer BCD decimal digits. The BCD sign is inserted packed against the symbol, no farther than the sixth character of the fifth word. The BCD relative address is packed against the sign, extending no farther than the fifth character of the sixth word. The tag, is isolated from the address of the CIT word 4. If the tag is greater than four, the flag T is inserted in following the tag. No diagnostic message results. A comma is inserted packed against the symbol or relative address, no farther than the sixth character of the sixth word, followed by the tag, no farther than the first character of the seventh word. The CIT decrement is isolated from the address portion of CIT word 1. If it exists, it is converted to 5 or fewer BCD decimal digits. A comma is inserted packed against the symbol or relative address, no farther than the second character of the seventh word, followed by the decrement, packed against the comma, no farther than the second character of the eighth word.

If following the symbolic address (and, if it exists, the relative address) no tag exists, the CIT decrement is isolated from the address portion of CIT word 1. If it exists, a zero is selected as the tag field, and processing continues as before.

If no symbolic address exists, the CIT fourth word is tested for a relative address and/or tag . If either or both exist, the relative address is isolated. If it exists, it is converted to 5 or fewer BCD decimal digits. If it is negative,

the sign is inserted in the sixth character of the fourth word. If positive, the first BCD numeral is inserted in the sixth character of the fourth word. The remaining characters are inserted left adjusted in the fifth word, and the tag and decrement are processed as before. If no symbolic address or relative address exist, a zero is inserted in the sixth character of the fourth word, and the tag and decrement are processed as before.

If no symbolic address, relative address or tag exist, the CIT decrement is isolated from the address portion of CIT word 1. If it exists, a zero is inserted in the sixth character of the fourth word, and the tag and decrement processed as before. Processing of the null tag is necessary to insert the nonredundant comma and zero tag field.

After the variable field has been processed, the final word is filled with blanks. If no variable field exists, a blank is inserted in the sixth character of the fourth word.

All processing converges at this point. If the op-code is not SYN or BSS, the relative counter is converted to 5 BCD octal digits, left adjusted, followed by a blank, and inserted in the second word of the hexad. The relative counter is bumped by one. If the op-code is BSS, the block length is assumed to be zero, hence for either BSS or SYN word two is blank, and the relative counter is unchanged. Word one of every hexad is blank. The CIT is now in the standard form

SYMBOL OPC ADDRESS+RA, TAG, DECREMENT

The six words of every hexad are transferred to a page image buffer. In this process, overflow of the machine language image to the seventh and first two characters of the eighth word are truncated. As the FORTRAN processor compiles TIX, TXI and TXL only in a DO loop, the only machine language instructions which may contain decrement fields are TXI *+1, 4, 32767 TIX *+1, 4, 32767 and TXL 4095A, 4, 32767. None of these will overflow.

A count is kept of the hexad entries made in the page image buffer. The first 58 entries are made in column one, the next 58 entries in column 2, and the next 58 entries in column 3. When 174 entries have been made, the page image is written on tape 2 following the end of second file mark. The page image is followed by a page restore.

When the end of the CIT file is sensed, the buffer is checked for a partial page image. If a partial image exists, it is written on tape 2. An end of file mark is written following the machine language listing, and tapes 2 and 4 are rewound. The information on tape 4 is no longer of significance.

Part S
On the 704, Part S is a separate record.
End card setting and/or physical sense switch 3 is tested to determine if on-line output of the source program, storage map, and machine language listing (if any) is required. If it is so requested, the page is restored so that each file begins on a new page.

One record (one printed line) is read from tape 2, is converted to a card image, and the line is printed.

When the end of the source program file is sensed, the page is restored, and the map is printed line by line.

When the end of the storage map file is sensed, sense light 2 is tested to determine if a third file, the machine language listing, exists on tape 2. If it does, this flag is restored for monitor, the page is restored and the listing is printed line by line.

When the end of the listing file is sensed, tape 2 is rewound. The FORTRAN compiler has completed processing of the source program. The results of the FORTRAN compilation are on two tapes: tape 2, the BCD source program, storage map, and symbolic listing if requested; and tape 3, the binary program card, the object program, library subroutines including their program cards if requested, and transfer card if a main program. If on line output has been requested, cards have been punched and listings have been printed. Control is passed to 1 - CS to return this job to the monitor.

XI LIBRARY AND LIBRARIAN

INPUT-OUTPUT LIBRARY

Control Routines

IOS / Input-Output Supervisor
IOU/ Input-Output Channel-Unit Table
SLO/ Short-List Output
SLI/ Short-List Input
WER/ Tape Write Error
RER / Tape Read Error

Hollerith Input-Output

IOH/ Input-Output Hollerith
STH/ Storage to Tape Hollerith
TSH/ Tape to Storage Hollerith
CSH/ Card to Storage Hollerith
SCH/ Storage to Card Hollerith
SPH/ Storage to Printer Hollerith

Binary Input-Output

IOB/ Input-Output Binary
STB/ Storage to Tape Binary
TSB/ Tape to Storage Binary
DRM/ Write Drum and Read Drum

Tape Non-Transmission

BST/ Backspace Tape
EFT/ Endfile Tape
RWT/ Rewind Tape

MATH LIBRARY

XP1/ Exponential - FXPT Base - FXPT Exp.
XP2/ Exponential - FLPT Base - FXPT Exp.
XP3/ Exponential - FLPT Base - FLPT Exp.
ATN/ Floating Point Arctangent
XPF/ Floating Point Exponential Function
LOG/ Floating Point Natural Logarithm
SCN/ Floating Point Sine and Cosine
SQR/ Floating Point Square Root
TNH/ Floating Point Hyperbolic Tangent

MONITOR LIBRARY

CHN / Chain
DMP / Dump
XIT / Exit

OTHER LIBRARY ROUTINES

FPT / Floating Point Trap
TES / Test Last Write
XLO / Relocated Location Function

THE LIBRARY EDITOR

LIB / Librarian

INPUT/OUTPUT LIBRARY

The 709 FORTRAN I/O LIBRARY was designed as a simple, generalized and flexible method for handling the input-output and conversion of data required by Fortran - compiled programs at object-time under Monitor or non-monitor operation. The I/O Library (IOL) consists of hand-coded, FAP-assembled, relocatable subroutines, which communicate with Fortran programs by means of linkage compiled by the I/O Translator (IOT) in Section One.

Most of the analysis done by the IOT concerns the items in the List. When indexing instructions are necessary for the List, entries are made in TDO table, which cause Section Two to compile the necessary instructions for the treatment of arrays conforming to standard Fortran usage, e.g. the first element is assigned the highest location of the array. The remainder of IOT's task is simple: the communication of the minimum amount of information necessary to the IOL. This could be: The unit designation, type designation, location of Format specification, and the termination of the List.

The simplicity of this scheme will become apparent during the following description. Its flexibility and generality provide the obvious advantages of easy modification, and a continuing opportunity for improvement. This partly explains the reason for the fragmentation of the IOL into about twenty different routines. Generally, in systems design, the linkage cost of keeping functions separate and distinct, is repaid both in memory space and in the ease with which additions and improvements may be made.

The IOL contains four types of routines:

1) for initialization and control:
   IOS, IOU, SLO, SLI, WER, RER;

2) for the transmission of information to and from each TYPE of I/O unit:
   STH, TSH, CSH, SCH, SPH, STB, TSB, DRM;

3) for the conversion of data, and/or its transmission to and from the data area, according to MODE:
   IOH, IOB;

4) and for non-transmission TYPE tape handling:
   BST, EFT, RWT.

In the following writeup, the mode routines (IOH,IOB) will be described in conjunction with the unit routines.

The general overall flow can be outlined as follows:

1) The logical unit designation, if necessary, is picked up, and control exits from the calling sequence to the indicated TYPE routine.

2) if this is a non-transmission TYPE routine, control passes directly to the control routine, IOS, for initialization. If a transmission type, except DRM, the TYPE routine furnishes the correct switch setting for input or output to the appropriate MODE routine. Then the MODE routine conveys the logical unit designation, along with the correct mode indication, to IOS.

3) IOS turns to the IOU table for the logical-actual unit correspondence, after having checked for the correct completion of a previous write statement. When all I/O commands have been initialized, control returns to the MODE routine (or to the non-transmission caller).

4) the MODE routine now controls transmission, and/or conversion, of data according to the Format specification and the List of items indicated by the calling sequence. A return is made to the TYPE unit routine for each record of input or output.

5) When the List is satisfied a final return is made to the MODE routine to make sure the last record is read or written, and to restore conditions.

TABLE OF USAGE

TYPE UNIT	MODE	SHORT-LIST CONTROL	TAPE ERROR CONTROL	CHANNEL-UNIT CONTROL
TSH	IOH	SLI	RER	IOS
STH	IOH	SLO	WER	IOS
CSH	IOH	SLI	-	IOS
SCH	IOH	SLO	-	IOS
SPH	IOH	SLO	-	IOS
STB	IOB	SLO	WER	IOS
TSB	IOB	SLI	RER	IOS
BST	-	-	-	IOS
EFT	-	-	-	IOS
RWT	-	-	-	IOS
DRM	-	-	-	-

CONTROL ROUTINES

IOS/INPUT-OUTPUT SUPERVISOR

Purpose:

To initialize all input-output instructions for a given logical unit designation according to logical-actual correspondences in IOU. These instructions may then be executed through a transfer vector.

Calling Sequences: used by all I/O routines except DRM.

```
    CAL     C(AC)              C(AC) = MODE,,UNIT
    TSX     $(IOS),4           where: MODE = 0 for BCD, 20(8) for BIN
    return                             UNIT = logical designation
    - - -
or  XEC*    $(XXX)             C(XR4) may = - ADDR.
```

where XXX may be: RDS, WRS, BSR, WEF, REW, ETT, RCH, TEF, TCO, TRC

Transfer Vector

(TES), (IOU)

Stop	Reason
HPR 0,6	Actual unit designation not found in IOU.

Storage Requirement (for 6 channels)

$124_{(8)}$ locations.

Description

IOS first makes sure any previous tape write is correct. Then, the current unit is compared with the last; if they are the same, then IOS exits to caller. If different, then if the current unit designation is either zero, or greater than the number of entries in IOU, IOS stops. Otherwise, IOS examines the indicated entry in IOU, and stops if the entry is zero. Otherwise, all unit instructions are initialized. Then, the current channel designation is checked against the last. If the same, then IOS exits to caller. If different, the channel instructions are initialized before IOS returns.

The instructions (RCH), (TEF), (TCO), (TRC) have an address and tag = 0,4. The two's complement of their effective address is contained in XR4.

IOU/Input-Output CHANNEL-UNIT TABLE

Purpose

To establish logical-actual channel-unit correspondences at object time.

Calling Sequence

Table referenced by IOS.

No Transfer Vector

No Stops

Storage Requirement

$14_{(8)}$ locations (as distributed). The number of locations required equals the number of units, plus one which contains the total number of tape unit addresses.

Description

(Only the address field of locations in IOU is presently used.)

(IOU)-3 contains the channel-unit address of the Printer.
(IOU)-2 contains the channel-unit address of the Punch.
(IOU)-1 contains the channel-unit address of the Reader.
(IOU) contains the number of tape unit addresses in the table.

The following N actual tape unit addresses correspond to the logical tape unit addresses from 1 to N. If there is no actual tape unit available for a logical unit address from 1 to N, that particular entry should be zero, to cause the stop HPR 0,6 in IOS.

SLO/SHORT LIST OUTPUT

Purpose

To provide list indexing for the output of non-subscripted arrays.

Calling Sequence

```
    TSX     $(SLO),4            - used in List
    PZE     SYMBOL+1
    PZE     N
    return
```

where: SYMBOL = Location of the array, and
N = Number of elements.

No Transfer Vector

No Stops

Storage Requirement

$15_{(8)}$ locations.

Description

SLO will initialize the instructions:

```
    AXT     1,4
    LDQ     SYMBOL+1,4
    STR
    TXI     *+1,4,1
    TXL     *-3,4,N
```

and trap back and forth on the STR between IOH or IOB, until each element of the array has been output in the normal Fortran order. To handle arrays in reverse of the normal order, it is possible to change the above instructions to:

```
    AXT     N,4
    LDQ     SYMBOL+1,4
    STR
    TIX     *-2,4,1
```

SLI/SHORT LIST INPUT

Purpose

To provide list indexing for the input of non-subscripted arrays.

Calling Sequence

```
    TSX     $(SLI),4            - used in List
    PZE     SYMBOL+1
    PZE     N
    return
```

where: SYMBOL = Location of the array, and
N = Number of elements.

No Transfer Vector

No Stops

Storage Requirement

$15_{(8)}$ locations.

Description

SLI will initialize the instructions:

```
    AXT     1,4
    STR
    STQ     SYMBOL+1,4
    TXI     *+1,4,1
    TXL     *-3,4,N
```

and trap back and forth on the STR between IOH or IOB, until each element of the array has been input in the normal Fortran order. To handle arrays in reverse of the normal order, it is possible to change the above instructions to:

```
    AXT     N,4
    STR
    STQ     SYMBOL+1,4
    TIX     *-2,4,1
```

WER/TAPE WRITE ERROR

Purpose

To check tape output.

Calling Sequence

```
    TSX     $(WER),4 - used by STH and STB
    return
or: STA*    $(WTC)   - to save the last command address.
```

Transfer Vector

(TCO), (ETT), (TRC), (TES), (BSR), (WRS), (RCH), (WEF), (REW).

Stops	Reason
HPR 0,5	I/O check light is on.
HPR 1,5	Fifth redundancy while writing.
HPR 2,5	First redundancy while erasing.
HPR 3,5	End of tape indicator is on.

Storage Requirement

$60_{(8)}$ locations.

Description

WER delays if the channel is in operation. Then tests are made for end of tape, I/O check, and redundancy check. If no tests fail, TES is reset to NOP, and control returns to caller. On end-of-tape, WER backspaces a record, writes end file mark, rewinds, loads the unit address into the MQ, and stops. If the start key is depressed, WER then rewrites the last record on the new tape and repeats the tests. On an I/O check, WER simply stops. On a redundancy check after writing, WER tests the error count and stops it if it is exhausted. If it is not exhausted, or if the start key is pressed, WER then backspaces, erases the previous record and checks the erase. If the erase failed, WER stops. If not, WER then rewrites the previous record and repeats all tests.

RER/TAPE READ ERROR

Purpose

To check tape input.

Calling Sequence

```
    TSX     $(RER),4        - used by TSH and TSB
    return
or: STA*    $(RDC)          - to save the last command address.
```

Transfer Vector

(TCO), (TRC), (TEF), (BSR), (RDS), (RCH).

Stops	Reason
HPR 0,3	I/O check light is on.
HPR 1,3	Tenth redundancy while reading.
HPR 2,3	End of file indicator is on.

Storage Requirement

$36_{(8)}$ locations.

Description

RER delays if the channel is in operation. Then tests are made for I/O check, redundancy, and end-of-file. If no tests fail, control returns to caller. On an I/O check, RER simply stops. On a redundancy check after reading, RER tests the error count and stops if it is exhausted. If it is not exhausted, or if the start key is pressed, RER then backspaces, rereads the previous record, and repeats all tests. on end-of-file RER stops, and if the start key is pressed, reads the first record beyond the file mark, and repeats all tests.

HOLLERITH INPUT-OUTPUT

IOH/INPUT-OUTPUT HOLLERITH

Purpose

To handle the transmission and conversion of BCD data according to List and Format specifications.

Calling Sequences

```
        LDQ      C(MQ)          - used by TSH,STH,CSH,SCH, and SPH.
        TRA*     $(IOH)         1,4 = L(FORMAT) and 2,4 = L (LIST)
```

where: C(MQ) = NOP XXX for input,
and C(MQ) = TRA XXX for output,
where XXX = the re-entry address to the TYPE unit routine,
when IOH is entered for initialization. Control returns to List.

```
then:   STR                    - used by the List
        STQ Symbol, TAG          for input
or:     LDQ Symbol, TAG        - used by the List
        STR                      for output.
```

when location 2 has been set to re-enter IOH for data conversion.

```
then:   TSX (RTN), 4           - used by the List
        return                   when input is terminated
or:     TSX (FIL), 4           - used by the List
        return                   when output is terminated.
```

Transfer Vector

(IOS)

Stops

Stops	Reason
HPR 0,1	Illegal format statement.
HPR 1,1	Illegal data.
HPR 2,1	Illegal data.
HPR 3,1	Illegal data.
HPR 4,1	Illegal data.
HPR 5,1	Illegal data.

Storage Requirement

$1553_{(8)}$ locations + $242_{(8)}$ erasable.

Description

When IOH is entered initially from one of the TYPE unit routines, switches are set for either input or output. The exit is set to return to the List. Various conditions are saved. Location 2 is set so that an STR will cause a return to the conversion part of IOH. Various indicators are reset. And IOS is called to initialize the I/O instructions for the indicated logical unit. If input, a record is read by the TYPE routine.

Description (cont'd)

Then the Format specifications, in their original BCD form, are scanned. If Hollerith, Blank, or Skip specifications are encountered, control remains in IOH and the specified number of BCD characters are taken from, skipped, or placed into the REC buffer. Then the Format Scan resumes. If data field specifications are encountered, the appropriate switches and counters are set for the indicated type of conversion fixed, floating or integer. If input, the specified number of characters are obtained from the BCD REC buffer, converted to binary, placed in the MQ, and control returns to the List. If output, control returns to the List immediately, to obtain a List item in the MQ. When the STR is executed, the List item is converted from binary to BCD and packed into the REC buffer. This continues until the count for repetition of this particular Format field specification is exhausted, or until the List is satisfied. When the field count is exhausted, the Format Scan resumes.

When a Slash is encountered by the Format Scan, a new record is either input or output by the current TYPE unit routine. This will also occur on the final right parenthesis of the Format, if the list has not been exhausted. However, if the List has been satisfied, a final return of control to IOH causes the output of the last record, and/or the restoration of the various saved conditions, before the final exit to the caller of the TYPE unit routine.

STH/STORAGE TO TAPE HOLLERITH

Purpose

To write one BCD tape record.

Calling Sequence

```
CAL     N           - unit designation.
TSX     $(STH),4    - or (STHM) under Monitor.
PZE     FMT         - location of Format specification.
. . .   . .         - indexing
LDQ     SYMBOL,TAG  - output list.
STR                 - trap to IOH.
. . .   . .         - indexing
TSX     $(FIL),4    - fill out record.
```

for: WRITE OUTPUT TAPE N,FMT,List.

Transfer Vector

(IOH), (WER), (TES), (WRS), (WTC), (RCH)

No Stops

Storage Requirement

$72_{(8)}$ locations + $25_{(8)}$ erasable.

Description

STH loads the MQ with the output switch setting and re-entry address, and exits to IOH for initialization. Then, whenever IOH is ready to output a record, it re-enters STH. If (STH) has been changed to (STHM) by the Monitor, then the line count is increased by 1. STH then exits to WER to check any previous write. On return from WER, the word count for the write command is obtained from 1,4, and the contents of the REC buffer in IOH are moved into the output buffer. TES is set to check the current write. The command address is saved in (WTC) for use by WER. STH then executes the (WRS) and (RCH) commands in IOS, to initiate the writing of the contents of the output buffer onto tape, and returns to IOH at 2,4.

TSH/TAPE TO STORAGE HOLLERITH

Purpose

To read one BCD tape record.

Calling Sequence

		for: READ INPUT TAPE N,FMT,List.
CAL	N	- unit designation.
TSX	$(TSH),4	- or (TSHM) under Monitor.
PZE	FMT	- location of Format specification.
. . .	. .	- indexing.
STR		- trap to IOH.
STQ	SYMBOL,TAG	- input List.
. . .	. .	- indexing.
TSX	$(RTN),4	- return to restore conditions.

Transfer Vector

(IOH), (RDS), (RDC), (RCH), (RER), EXIT

No Stops

Storage Requirement

$30_{(8)}$ locations + $25_{(8)}$ erasable.

Description

TSH loads the MQ with the input switch setting and re-entry address, and exits to IOH for initialization. Then, whenever IOH requires an input record, it re-enters TSH. TSH then executes the (RDS) and (RCH) commands in IOS, causing a 20-word record to be read into the IOH input buffer. The command address is saved in (RDC) for use by RER. If (TSH) has been changed to (TSHM) by the Monitor, then an end of file on channel A causes an exit to the Monitor. Otherwise, TSH exits to RER to check the read, and returns to IOH directly from RER.

CSH/CARD TO STORAGE HOLLERITH

Purpose

To read one Hollerith card and convert to BCD.

Calling Sequence

for: READ FMT, List.

```
TSX     $(CSH),4
PZE     FMT          - location of Format specification.
. . .   . .          - indexing
STR                  - trap to IOH
STQ     SYMBOL,TAG   - input List.
. . .   . .          - indexing
TSX     $(RTN),4     - return to restore conditions.
```

Transfer Vector

(IOH), (TCO), (TEF), (RDS), (RCH)

Stops

	Reason
HPR 0,2	- Illegal card character.
HPR 1,2	- End of file.

Storage Requirements

$173_{(8)}$ locations + $141_{(8)}$ erasable.

Description

CSH sets the AC to -1, for the IOU table reference, loads the MQ with the input switch setting and re-entry address, and exits to IOH for initialization. Then, whenever IOH requires an input record, it re-enters CSH. CSH then executes the (TCO), (RDS), (RCH), and (TEF) commands in IOS. On an end-of-file, CSH stops until the card can be readied. When a card has been read, CSH then converts it from Hollerith and places the BCD in the IOH input buffer. If an illegal character is encountered, CSH stops until the corrected card can be readied. When the card has been converted, control returns to IOH.

SCH/STORAGE TO CARD HOLLERITH

Purpose

To convert BCD to Hollerith and punch one card.

Calling Sequence

for: PUNCH FMT, List.

```
TSX       $(SCH),4
PZE       FMT          - location of Format specification.
. . .     . .          - indexing
LDQ       SYMBOL,TAG   - output list.
STR                    - trap to IOH.
. . .     . .          - indexing
TSX       $(FIL),4     - fill out card.
```

Transfer Vector

(IOH), (TCO), (WRS), (RCH).

No Stops

Storage Requirements

$140_{(8)}$ locations + $148_{(8)}$ erasable.

Description

SCH sets the AC to -2, for the IOU table reference, loads the MQ with the output switch setting and re-entry address, and exits to IOH for initialization. Then, whenever IOH is ready to output a record, it re-enters SCH. SCH then obtains the output word count from 1,4, and converts the contents of the REC buffer in IOH from BCD to Hollerith and places it in the output buffer. When the image conversion is complete, SCH executes the (TCO), (WRS), (RCH) commands in IOS, causing the Hollerith card to be punched. When punching is complete, SCH returns to IOH at 2,4.

SPH/STORAGE TO PRINTER HOLLERITH

Purpose

To convert BCD to line image and print a line.

Calling Sequence

```
                                    for: PRINT FMT, List.
TSX     $(SPH),4
PZE     FMT         - location of Format specification.
. . .   . .         - indexing
LDQ     SYMBOL,TAG  - output List.
STR                 - trap to IOH.
. . .   . .         - indexing
TSX     $(FIL),4    - fill out line.
```

Transfer Vector

(IOH), (WRS), (TCO), (RCH).

No Stops

Storage Requirements

$270_{(8)}$ locations + $163_{(8)}$ erasable.

Description

SPH sets the AC to -3, for the IOU table reference, loads the MQ with the output switch setting and re-entry address, and exits to IOH for initialization. Then, whenever IOH is ready to output a record, it re-enters SPH. SPH then obtains the output word count from 1,4, and converts the contents of the REC buffer in IOH from BCD to line image and places it in the output buffer. Then according to program control characters, SPH senses the hubs, and executes the (TCO), (WRS), (RCH) commands in IOS, to print left and right halves of the line respectively. When printing is complete, SPH returns to IOH at 2,4.

BINARY INPUT-OUTPUT

IOB/INPUT-OUTPUT BINARY

Purpose

To handle the transmission of binary between storage and tape buffers according to List specifications.

Calling Sequences

```
        LDQ       C(MQ)              -used by STB and TSB.
        TRA*      $(IOB)             1,4= L(LIST)
```

where C(MQ) = STQ XXX,4 for output,
and C(MQ) = LDQ XXX,4 for input,
where XXX = the re-entry address to the TYPE unit routine,
when IOB is entered for initialization. Control returns to List.

```
then:   LDQ Symbol, TAG            -used by the List
        STR                        for output.
or:     STR                        -used by the List
        STQ Symbol, TAG            for input.
```

when location 2 has been set to re-enter IOB for transmission.

```
then:   TRA* $(EXB)                -used by STB and TSB
```

when the last physical record has been written or read.

Transfer Vector

(IOS)

No Stops

Storage Requirement

$463_{(8)}$ locations + 3 erasable.

Description

When IOB is entered initially from one of the TYPE unit routines, switches are set for either input or output. The binary indicator is added to the AC, and IOS is called to initialize the I/O instructions for the indicated logical unit. If input, a record is read by the TYPE routine. Various indicators are reset, and the contents of locations 0 and 2 are saved. Location 2 is set so that an STR will cause a return to the transmission part of IOB. And control returns to the List.

When an STR is executed, one word is put into, or taken from, one of two 127-word buffers, and the count is decreased by 1. Meanwhile, whenever there is more than one physical record in a logical record, the other buffer is either being written out, or being read into. This is accomplished by switching the address of the working buffer and the address of the I/O command whenever the count is exhausted. To initiate the tape read or write, the indicated TYPE routine is called.

Description (continued)

When the List has been satisfied, and the last physical record has been written or read, control returns finally to IOB to restore the contents of locations 0 and 2, before the exit to the caller.

STB/STORAGE TO TAPE BINARY

Purpose

To write one physical record and its appropriate label onto tape.

Calling Sequence

```
                                        for:  WRITE TAPE N, List
CAL     N                               -unit designation
TSX     $(STB),4
...     ..                              -indexing.
LDQ     Symbol, TAG                     -output List
STR                                     -trap to IOB.
...     ..                              -indexing.
TSX     $(WLR),4                        -write last physical record.
```

Transfer Vector

(IOB), (WER), (WRS), (WTC), (RCH), (TES), (EXB)

No Stops

Storage Requirement

$63_{(8)}$ locations + 1 erasable.

Description

STB loads the MQ with one output switch setting and re-entry address, and exits to IOB for initialization. Then, whenever IOB is ready to output a record, it re-enters STB. STB first exits to WER to check any previous write. On return from WER, the physical record count is increased by 1, the current command address is saved in (WTC) for use by WER, and the commands (WRS) and (RCH) in IOS are executed to initiate the writing of the contents of the indicated buffer, preceded by a zero label, onto tape. STB then returns to IOB at 2,4.

When the list has been satisfied, STB is entered at (WLR). Any previous write is checked by calling WER. The write command is set with the current buffer address and the current word count obtained from IOB. The physical record count is increased by 1, and placed in the address portion of the label. The PRC counter is reset to zero. The writing of the partial buffer-load and the non-zero label are initiated. The command address is saved in (WTC). TES is set to check the current write. And STB exits to IOB at (EXB).

TSB/ TAPE TO STORAGE BINARY

Purpose

To read one physical record and its appropriate label from tape.

Calling Sequence

```
                                        for: READ TAPE N, List.
CAL     N                               -unit designation
TSX     $(TSB),4
...     ..                              -indexing.
STR                                     -trap to IOB.
STQ     Symbol, TAG                     -input list.
...     ..                              -indexing.
TSX     (RLR),4                         -read last physical record.
```

Transfer Vector

(IOB), (RER), (RDS), (RDC), (RCH), (EXB)

No Stops

Storage Requirement

$42_{(8)}$ locations

Description

TSB loads the MQ with the input switch setting and re-entry address, and exits to IOB for initialization. Then, whenever IOB requires an input record, it re-enters TSB. TSB first exits to RER to check any previous read. On return from RER, the last label read is examined. If non-zero, control returns to IOB at 2,4. If zero, the commands (RDS) and (RCH) in IOS are executed to initiate reading the next physical record into the indicated buffer. The address of the read command is saved in (RDC) for use by RER. And control returns to IOB at 2,4.

When the list has been satisfied, TSB is entered at (RLR). Any previous read is checked by calling RER. If the label is zero, TSB then continues to read physical records until a non-zero label is encountered, which signifies the last physical record of this logical record. Then TSB exits to IOB at (EXB).

DRM/WRITE AND READ DRUM

Purpose

To transfer arrays to and from a Drum.

Calling Sequence

```
                                    for: WRITE (or READ) DRUM, N, J, List
CAL     N                           -Drum designation.
TSX     $(XXX), 4                   (XXX= SDR for output, DRS for input)
CAL     J                           -Drum address.
LDA     ..                          -set by DRM.
LXD     2)K, TAG                    -indexing.
CPY     Symbol, TAG                 -array Symbol
TIX     *-1, TAG, 1                 -indexing.
CPY     Symbol                      -last element.
```

No Transfer Vector

No Stops

Storage Requirement

$17_{(8)}$ locations.

Description

The drum designation is used to initialize the write or read select. The drum is selected. The drum address is moved into DRM, and its location is stored in the address of the LDA instruction. Then DRM exits to the LDA at 2,4.

TAPE NON-TRANSMISSION

BST/BACKSPACE TAPE

Purpose

To backspace the indicated tape one logical record.

Calling Sequence

```
CAL     N          for: BACKSPACE N
TSX     $(BST),4   -unit designation
return
```

Transfer Vector

(IOS), (BSR), (RDS), (RCH), (TCO), (TRC), (TEF)

No Stops

Storage Requirement

$34_{(8)}$ locations + 1 erasable.

Description

The binary indicator is added to the AC, and IOS is called to initialize the I/O instructions for the indicated logical unit. Then BST attempts to read the previous physical record in the binary mode, by executing the instructions: (BSR), (RDS), (RCH), and (TCO). Then, if a (TRC) or a (TEF) cause a transfer when executed, only one backspace is required. Otherwise, BSR backspaces the number of physical records specified by the last binary record label. Control then returns to the caller at 1,4.

EFT/ENDFILE TAPE

Purpose

To write end-of-file on the indicated tape.

Calling Sequence

```
CAL     N           for: ENDFILE N.
TSX     $(EFT),4    -unit designation
return
```

Transfer Vector

(IOS), (WEF)

No Stops

Storage Requirement

$7_{(8)}$ locations.

Description

IOS is called to initialize the I/O instructions for the indicated logical unit. Then EFT simply executes (WEF), causing an end-of-file to be written. Control returns to the caller at 1,4.

RWT/REWIND TAPE

Purpose

To rewind the indicated tape.

Calling Sequence

```
CAL     N           for: REWIND N
TSX     $(RWT),4    -unit designation
return
```

Transfer Vector

(IOS), (REW)

No Stops

Storage Requirement

$7_{(8)}$ locations.

Description

IOS is called to initialize the I/O instructions for the indicated logical unit. Then, RWT simply executes (REW), causing the tape to be rewound. Control returns to the caller at 1,4.

MATH LIBRARY

The 709 FORTRAN MATH LIBRARY consists of modified SHARE Library routines, which have been FAP-assembled, and which communicate with Fortran programs by means of linkage compiled by the Arithmetic Translator in Section One.

XP1/EXPONENTIAL - FXPT BASE - FXPT EXP.

Purpose

To compute I^J, where I and J are fixed point variables.

Calling Sequence

```
        CLA     I       for: I**J in an arithmetic statement
        LDQ     J       -fixed point base
        TSX     $EXP(1,4 -fixed point exponent
        return
```

No Transfer Vector

No Stops

Storage Requirement

$43_{(8)}$ locations + 2 erasable.

Description

(If I=0, then $I^J = 0$; if J= 0, then $I^J = 1$)

XP2/EXPONENTIAL - FLPT BASE - FXPT EXP.

Purpose

To compute A^K, where A is a floating point variable and K is a fixed point variable.

Calling Sequence

```
        CLA     A       for: A**K in an arithmetic statement
        LDQ     K       -floating point base
        TSX     $EXP(2,4 -fixed point exponent
        return
```

No Transfer Vector

No Stops

Storage Requirement

$46_{(8)}$ locations + 2 erasable

Description

(If A = 0, then A^K = 0; if K = 0, then A^K = 1.0)

XP3/EXPONENTIAL - FLPT BASE - FLPT EXP.

Purpose

To compute B^C, where B and C are floating point variables.

Calling Sequence

```
CLA     B       for: B**C in an arithmetic statement
LDQ     C       -floating point base
TSX     $EXP(3,4,  -floating point exponent
return
```

No Transfer Vector

No Stops

Storage Requirement

$160_{(8)}$ locations + $10_{(8)}$ erasable

Description

(If B= 0, then B^C= 0; if C= 0, then B^C= 1.0)

ATN/ FLOATING POINT ARCTANGENT

Purpose

To compute the principal value of arctan (X) where X is a floating point, single precision, argument in radians.

Calling Sequence

```
CLA     X
TSX     $ATAN,4
return
```

for: ATAN F(X) in an arithmetic statement
-floating point argument

No Transfer Vector

No Stops

Storage Requirement

$115_{(8)}$ locations + 3 erasable

Timing

$\sim$ 1.98 milliseconds

Accuracy

Error $\leq 1 \times 10^{-8}$

Description

(If $|X| > 2^{27}$, then arctan $|X| = \pi/2$; if $|X| < 2^{27}$, then arctan $|X| = |X|$)

Adapted from SHARE Routine IBATN1, Distribution Number 507.

XPF/FLOATING POINT EXPONENTIAL FUNCTION

Purpose

To compute e^x for a floating point, single precision, argument.

Calling Sequence

```
CLA     X
TSX     $EXP,4
return
```

for: EXPF(X) in an arithmetic statement
-floating point argument.

No Transfer Vector

No Stops

Storage Requirement

$56_{(8)}$ locations + 4 erasable

Timing

$\sim$ 1.7 milliseconds

Accuracy

Error $< 1 \times 10^{-8}$

Description

(If $X > 88.028$, then $e^x = x$; if $X < -88.028$, then $e^x = 0$)
Adapted from SHARE Routine IB FXP, Distribution Number 507.

LOG/FLOATING POINT NATURAL LOGARITHM

Purpose

To compute $\log_e x$, natural logarithm, for a floating point, single precision, argument in normalized form.

Calling Sequence

```
CLA     X
TSX     $LOG,4
return
```

for: LOGF (X) in an arithmetic statement
-floating point argument

No Transfer Vector

No Stops

Storage Requirement

$56_{(8)}$ locations + 3 erasable

Timing

1.848 milliseconds

Accuracy

Error $\leq 3 \times 10^{-8}$

Description

(If x = 0, then log x = 0; if x<0, then LOG computes log | x|)
Adapted from SHARE Routine IB LOG 3, Distribution Number 665.

SCN/FLOATING POINT SINE AND COSINE

Purpose

To compute the sine or cosine of a floating point, single precision, normalized argument, in radians.

Calling Sequence

```
        CLA     X          for: SINF (X) in an arithmetic statement
        TSX     $ SIN,4    -floating point argument
        return
or:                        for: COSF (Y) in an arithmetic statement
        CLA     Y          -floating point argument
        TSX     $ COS,4
        return
```

No Transfer Vector

No Stops

Storage Requirement

$151_{(8)}$ locations + 4 erasable

Timing in Milliseconds

SIN: 1.38 to 2.03; COS: 1.45 to 2.11

Accuracy

Error $\leq 1 \times 10^{-8}$

Description

(Cos x = Sin x + $\pi/2$)
Adapted from SHARE Routine IB SIN 1, Distribution Number 507.

SQR/FLOATING POINT POINT SQUARE ROOT

Purpose

To compute the square root of a floating point, single precision, argument.

Calling Sequence

```
CLA     X
TSX     $ SQRT,4
return
```

for: SQRTF (X) in an arithmetic statement
-floating point argument

No Transfer Vector

No Stops

Storage Requirement

$54_{(8)}$ locations + 4 erasable

Timing

1.062 milliseconds

Accuracy

Error $\leq 1 \times 10^{-8}$

Description

(If $x = 0$, then $\sqrt{x} = 0$; if $x<0$, then $\sqrt{x} = x$)
Adapted from SHARE Routine IB SQ1, Distribution Number 721.

TNH/FLOATING POINT HYPERBOLIC TANGENT

Purpose

To compute tanh (x) for a floating point, single precision, argument, in radians.

Calling Sequence — for: TANHF (X) in an arithmetic statement

```
CLA     X
TSX     $ TANH,4
return
```

No Transfer Vector

No Stops

Storage Requirement

$126_{(8)}$ locations + 4 erasable

Timing

2.09 to 2.64 milliseconds

Accuracy

Error $\leq 3 \times 10^{-8}$ for $X \rightarrow .00034$ or $x \rightarrow .17$, and $\leq 1 \times 10^{-8}$ elsewhere.

Description

(If $|X| < .00034$, then tanh $(X) = X$; if $|X| > 12$, then tanh $(X) = \pm 1$)
Adapted from SHARE Routine IB TANH, Distribution Number 507.

MONITOR LIBRARY

The 709 FORTRAN MONITOR LIBRARY consists of hand-coded, FAP assembled, relocatable subroutines, which use linkage compiled by Section One to communicate with Fortran programs that are executed under Monitor control.

CHN/ CHAIN

Purpose

To load the indicated chain link from tape into cores, and pass control to it.

Calling Sequence

for: CALL CHAIN (R,T)

```
TSX     $CHAIN,4
TSX     L(R)        -record identification
TSX     L(T)        -actual channel B tape designation (1,2,or3)
```

Transfer Vector

(TES), EXIT

No Stops

Storage Requirement

$236_{(8)}$ locations

Description

After checking any previous write through TES, CHN searchs the channel B tape specified by the argument: T, for the record beginning with the control word: PZE 0, T, R. If an EOF is encountered before the record is found, the tape is rewound and the search continues. A second EOF causes an error message, and the job is deleted by calling EXIT. The same will occur, if five attempts to read the tape fail. If the search is successful, CHN then moves the Execution Loader on top of The Diagnostic Caller, and transfers control to it.

DMP/DUMP

Purpose

To control the dumping of cores and panel according to Argument specifications, during execution.

Calling Sequences

for: CALL DUMP (A, B, F . . .)

```
TSX     $DUMP, 4
TSX     A           - first limit
TSX     B           - second limit
TSX     L(F) or F   - dump format
. . .   . .
```

Control returns to the Monitor through EXIT.

or: for: CALL PDUMP (A, B, F . . .)

```
TSX     $PDUMP, 4
(arguments as above)
```

Control returns into execution, after restoring memory.

Transfer Vector

(TES), EXIT

No Stops

Storage Requirement

$245_{(8)}$ locations.

Description

After checking any previous write through TES, DMP saves $3500_{(8)}$ words on tape B2. Then a table is prepared of no more than 20 sets of arguments, and is written onto tape B2. The Monitor Dump Record is called into memory, and control remains with it until the argument table is exhausted. Then memory is restored and control re-enters DMP. If any arguments remain to be processed, another table is formed, and the Dump Record is called again. This continues until no arguments remain. Then, if DMP was entered at DUMP, control returns to the Monitor through EXIT. Otherwise, execution is resumed, if memory was correctly restored. Redundancy while reading memory from tape B2 will cause an error message, and execution will be terminated. Control then returns to the Monitor through EXIT.

XIT/ EXIT

Purpose

To terminate execution, and transfer control to the Sign-on record of the Monitor.

Calling Sequence

for: CALL EXIT

```
TSX     $EXIT,4
```

Transfer Vector

(TES)

Stop

Stop	Reason
HTR *-12	5 failures while reading System Tape

Storage Requirement

$23_{(8)}$ locations

Description

EXIT first checks the completion of any previous write by executing (TES). Then the System Tape is rewound, 1 TOCS is restored without destroying the current line count, and the System Tape is positioned to the Sign-on Record of the Monitor. If a redundancy stop occurs, pressing the start key will cause 5 new attempts. If there was no redundancy, control passes to 1 TOCS and the Monitor Sign-on is entered.

FPT/FLOATING POINT TRAP

Purpose

To handle the floating point trap feature during execution.

Calling Sequence

for: any FORTRAN Main Programs

```
$$      CLA     $(FPT)
        STO     8               -trap cell
        STZ     4)-205          -overflow data cell
```

The trap cell, location 8, is set to: TTR (FPT). Control returns indirectly through location 0.

No Transfer Vector

No Stops

Storage Requirement

$22_{(8)}$ locations + 2 erasable.

Description

When control traps into FPT, overflow and underflow data is saved in 4)-205. The AC and MQ are set according to overflow and underflow conditions. Then control exits to the address in location 0.

TES/ TEST LAST WRITE

Purpose

To test the correct completion of any previously initiated tape write.

Calling Sequence

for: FAP-coded programs

```
XEC*    $(TES)
return
```

No Transfer Vector

No Stops

Storage Requirement

1 location

Description

TES consists of one NOP instruction, which is set to TSX (WER),4 by STH and STB at execution time, and reset to NOP by WER after the writing of any previous tape record has been checked. This instruction, XEC* $(TES), should be used in FAP-coded programs that contain input-output instructions to make sure the execution of FORTRAN I/O statements is complete.

XLO/RELOCATED LOCATION FUNCTION

Purpose

To return the relocated location of its argument to the AC as a FORTRAN fixed point constant.

Calling Sequence

for: XLOCF (N) in an arithmetic statement

```
TSX     $XLOC,4
return
```

The instruction, CLA N, precedes the calling sequence at some point.

No Transfer Vector

No Stops

Storage Requirement

$14_{(8)}$ locations.

Description

XLO searchs for the last CLA N preceding the calling sequence, and obtains the location of the argument N. This is placed in the decrement of the AC, and control returns at 1,4.

THE LIBRARY EDITOR

LIB / LIBRARIAN

Purpose

To write the 709 FORTRAN LIBRARY onto tape in card record form.

Binary Deck

LIB consists of 15 row binary cards: the first of which is a one-card loader, the following 11 cards are the FAP-assembled Librarian in absolute, the 13th card is the one to be removed to cause the Library to be written on B5, the 14th is the TRA card, and the 15th card is blank to denote the END of the Library.

Stops		Reason
HPR	$777_{(8)}$	Checksum error.
HPR	$77777_{(8)}$	Final stop.
HPR	$2221_{(8)}$	RTT while reading B1.
HPR	$1221_{(8)}$	RTT while writing A1.
HPR	$2225_{(8)}$	RTT while writing B5.

Description

LIB will copy two files from B1 to A1, unless sense light 1 has been turned on by the Fortran Editor Program. It will then write the Library as a third file onto A1, either from cards (row-binary or column-binary), or from B1, if the first card is blank (the LIB END card). Then, if the card labeled LIB B5 has been removed, LIB will also write the Library onto B5. If no cards follow the LIB END card in the card reader, LIB will copy the fourth file from B1 onto A1, and halt. Otherwise, it will load the self-loading program which follows the LIB END card. LIB will buffer its input-output if run on a 32K 709, and will run non-buffered on a 8K 709.

XII

MONITOR ROUTINES FOR 709/7090 FORTRAN

The Monitor exists in separate records on the system tape. No monitor record remains in core after another record has been called in. In fact, the only core communication between monitor programs is 1-CS, which contains four erasable cells used by the monitor. The monitor records, and their system tape numbers are as follows:

Record No.	Name
1	Card-Tape Simulator
2	Dump
3	Sign on
4	FAP Pass 1
5	FAP Pass 2
6	Monitor Scan
7	BSS 1
8	Machine Error
9	Source Error
42	Tape Mover
43	BSS 2

Records 10-41 are the FORTRAN compiler. There is an EOF mark after record 9, and another after record 43. The 1-CS record is self loading and precedes record 1. 1-CS remains in memory except when an object program has control during execution.

I. START card and record 1 (card-tape-simulator)

System operation for a series of jobs followed by an end tape card begins by the START card, which installs 1-CS in memory and transfers control to record 1. This is the one and only time record 1 is called. Record 1 tests a flag left by the START card signalling that operation is with the monitor (rather than single compile), and then tests the card reader. If the card reader is empty, the assumption is made that input is off-line, and so record 2 is skipped and control is passed to record 3. If the card reader is not empty, a card to tape operation is simulated from the card reader to tape A2. The following specifications are observed. Cards with a 7-9 punch in column 1 are treated as column binary. Card with an 8-9 punch in column 1 are not transcribed onto tape, but cause an EOF to be written.

An EOF in the card reader causes an EOF on tape A2, and termination of the operation. The program is double buffered, card reader speed. Illegal Hollerith punches cause a stop with a restart procedure analagous to that for off-line equipment. At the termination of operation, record 2 is skipped and control is passed to record 3 (sign on). (Note: Since this program is also used in the single compile mode, if the monitor flag is off, card-tape simulation is done onto B2 and control passed directly to FORTRAN at record 10.)

II. Record 3 (Sign on)

Record 3 is called when and only when a job is begun (or ended). A test is made to determine if the input tape (A2) is positioned at the beginning of a file (job). If it is not, the tape is skipped to the next file. The number of lines output from the last job (kept in a cell in 1-CS), if greater than zero, is converted to decimal and reported on-line and off-line. Then the first card of the file is read and scanned to see if it is an End Tape card. If it is, an EOF is written on the BCD output tape, 2 EOF's are written on the peripheral punch tape, the End Tape card is printed on and off line, and a load card button sequence is simulated to end monitor operation. If the 1st card is not an End Tape card, it is assumed to be an I.D. card. It is in this area that a sizable space has been reserved for an installation to insert coding for accounting or other purposes. The distributed treatment merely writes an EOF on the peripheral punch tape and prints the I.D. card on line and off line. After treating the I.D. card, control is passed to record 6 (monitor scan). (Note: Record 3 has its own diagnostic messages and prints then on & off line).

III Record 6 (monitor scan)

Record 6 is the primary monitor record in that it interprets the control cards which specify different system programs to be called. It also scans FORTRAN programs and prepares a single-compile input tape for the compiler. Control is passed to this record in the following circumstances:

a) From record 3 (sign on) after processing an I.D. card at the beginning of a job.

b) From record 5 (FAP) after completing an assembly not for execution.

c) From record 7 or 43 (BSS) after relocating a series of binary programs when there are more symbolic programs remaining in the job.

d) From record 8 (machine error) or record 9 (source error) or record 2 (dump) when it has been determined that the job should be continued after an error.

e) From the restart card "CONTINUE".

Operation is as follows: All input is from A2. Records are read double buffered and scanned first for an asterisk in column 1. If this is found, the mnemonics on the card are scanned and compared with a dictionary of control card mnemonics. If no asterisk is found, the card is assumed to be part of a FORTRAN program and a routine called SP is used. If the card is column binary, and an XEQ control card has been encountered earlier in the job, control is passed to record 7 (BSS 1). If the XEQ flag is off, column binary cards are ignored. Asterisk cards not in the dictionary are printed on and off line as remarks and then ignored. FORTRAN source program cards are scanned and then transcribed onto tape B2 (FORTRAN input). A FORTRAN source card with a CALL CHAIN (N, Bn) will be changed to CALL CHAIN (N,n). Upon encountering an END card, another END card is simulated onto B2 containing output options as indicated by control cards. Programmer's END card options will be preserved if not in conflict with control cards, which have precedence. Asterisk (control cards) found in the dictionary, are treated as follows:

a) XEQ - a flag in 1-CS is set indicating execution is desired. A word of zeros is written on the beginning of tape B1 to indicate that there is no snapshot (See record 7).

b) CHAIN ( ) - If the execution flag is off, this is treated as a remark card. If on, the parameters are examined and a unique control word is written on B1 (in front of the zero word) and stored in a cell (curchn) in 1-CS. If this is the 1st link, it is stored in a different cell (1st chn). A chain flag is set in 1-CS (FLGBX)

c) FAP - An END card is simulated onto B2 containing control card output options and control is passed to FAP Pass 1 (record 4).

d) DATA - This should be encountered only if there was no execution flag (or if execution has been deleted). Control is passed to Sign-on unless the execution flag is on, in which case an error message (incorrect deck set up) is printed and control is passed to record 9 (source error).

e) CARDS ROW, LIBE, etc. - A flag is set for the end card routine to set the appropriate END card options.

In summary, control is then passed as follows:

Upon Recognizing:	Go to:
a) Fortran END card -	Record 10
b) Column binary card -	Record 7
c) FAP control card -	Record 4
d) Deck error -	Record 9
e) Machine error -	Record 8

NOTE (Record 6 has its own diagnostic messages and prints them on and off line).

IV. Record 7 or 43 (BSS Control)

Records 7 and 43 are identical except for tape positioning. In fact, they are identical except for the first word. This record is duplicated in order to be quickly accessible either from the second file (after a FORTRAN compilation) or from the first file (for binary input from record 6 or for a just-completed FAP assembly). BSS accepts card image input from A2, B3, or A1 using a generalized double buffered read routine. The BSS program itself is located in the top of memory, occupying the standard Common region. It relocates binary card images into locations 144_8 to 73000_8. 73000_8 to 74456_8 is used for a table of BCD program names, a missing subroutine table, and a Transfer Vector table. These tables, together with several loading counts are referred to as the SNAPSHOT.

Upon entry, the SNAPSHOT (from previous relocations in the same job) is read from B1. (If this is the first pass through BSS for the job a zero word will have been written on B1 indicating this). A number left by the calling record in the indicators specifies which tape BSS should first take as input. If an assembly or compilation has just been completed, this is B3, otherwise A2. This input tape is then read in binary. Transfer Vectors are peeled off and stored in the transfer vector table, and the binary cards are relocated into 144_8 - 73000_8. When a new set or Transfer Vectors is met, the relocated block is saved as a record on B1, the first word being a control word specifying its length and memory assignment and whether it has a transfer vector, If it has a transfer vector, there is a second control word written specifying how many TTR's exist for this block. When an EOF is encountered on B3 (if that is the 1st input tape) the input tape is changed to A2. If overlap with BSS occurs (i.e. program relocation over 73000) the non-overlapping block is saved as before on Bland relocation continues as before, but the relocation buffer is shifted back to 144_8. Overlap with Common is of course not allowed and results is an error message. If a BCD Record is enconntered, it is scanned and compared with a dictionary of control words. CHAIN and DATA are accepted. XEQ is ignored (as is obviously extraneous). Any other BCD cards result in the SNAPSHOT being written on B1 and control returned to record 6.

If DATA or CHAIN is recognized, the table of Transfer Vectors is searched against the table of BCD names to form a table of missing subroutines, (MISUB). The input tape is then changed to A1, which becomes positioned at the library, and the read and relocation routines are modified to search for and relocate missing subroutines from the library. MISUB is updated with lower level missing subroutines and the search continues until the MISUB count is zero or until two passes have been made over the library. If subroutines are still missing, they are listed with an error message on and off line and the Job is deleted. When all subroutines have been relocated, the Transfer Vector table is changed to TTR's with their proper relocated addresses and the blocks are read in from B1 and written onto A4 preceded by their appropriate TTR's.

At this point a test is made to see if this is a Chain Job. If not, a small execution loader is moved over 1-CS, the word "execution" is printed, and control is passed to the execution loader, which reads the absolute programs from A4 into memory. The last record on A4 is a transfer word to the program.

If it is a Chain Job, and the data card has not been encountered, B1 is backspaced to the chain I.D. word and the link is stacked on B1. BSS is refreshed, and the process begins a new, reading A2. If it is a Chain job and the Data card has been encountered, the chain links are edited from B1 onto their apecified tapes and the 1st link is located. Then the small execution loader is placed over 1-CSand the link is read in as in a single job, except that it is from B1, 2, or 3 instead of A4. This program carries a sizable set of diagnostic messages that it prints on and off link and then returns control to the machine errror record or soucce error record as appropriate.

V. Monitor Control During Execution, via Libray Routines.

This is effected by two Monitor Routines, (EXIT and CHAIN), and by the modification of two I/O routines, ((TSH) and (STH)). EXIT restores 1-CS and calls in record 3 (Sign on). CHAIN searches for a specified chain link and, upon finding it, stores a tape reading program over 1-CS exactly as is done in BSS. The chain tape is searched from whatever point it is positioned, and if an EOF is obtained, it is rewound and the entire link file is searched once. Failure to find the link results in an error message and the job is deleted. Transfer Vectors to (STH) and (TSH) are modified at loading time (in BSS) to (STHM) and (TSHM) which are alternate entry points forcing these routines to do Monitor functions, namely 1) Call the EXIT routine if an EOF is encountered with a READ INPUT TAPE statement, 2) Check for EOT on the Monitor output tape, and 3) Update the Monitor output tape line count.

VI. Unused Locations

In the 32K system, unused locations are (in decimal) 3 through 7, 9 through 18, and 50 through 99. In the 8K system, unused locations are 3 through 7, 38, and 39. Location 8 is used only at object time for floating point traps. Locations 0, 1, and 2 are used as erasable only for load tape and load card sequences and at object time for I/O routine linkage. In other words, an installation may feel free to write routines that destroy locations 0, 1 and 2 as long as these routines do not interrupt an object time I/O sequence.

XIII

GENERAL DIAGNOSTIC

The general diagnostic for the FORTRAN system covers machine and source program errors revealed by Sections 1 Prime through 6. When a machine or source program error is encountered in any one of these executive system records, a TSX DIAG, 4 transfers control to the diagnostic caller. In the 709, the diagnostic caller remains in lower memory with 1 to CS. In the 704, the TSX DIAG , 4 is actually a transfer to 1 to CS which then reads in the diagnostic caller record positioned after that particular system record on the system tape. The caller then dumps a buffer of 25 ØØ words onto tape A3 on the 709, or onto drum 4 on the 704. The diagnostic caller then spaces the system tape to the 4th file and proceeds to read in the main diagnostic record.

The main diagnostic record of the 4th file (record 1 in the 709, record 2 in the 704) contains all of the subroutines needed for converting and printing error comments, and for returning to the proper record in the FORTRAN Monitor. The main record converts the contents of index register 4 back to the location number of the TSX, and uses this constant as the error number. It is also in the main record that the title - FORTRAN record number and location of the TSX - is printed. The 709 diagnostic also prints the section of FORTRAN involved.

The main record performs a table search in order to determine which of the fourth file records contains information pertinent to the stop. The error number (2's complement of IR4) is compared to a list of errors. This list has 2-word entries. The first word is an error number corresponding to the location of a TSX in the executive system. The second word is the record number in the fourth file which contains the pertinent BCD and coding to print out information about the error. If the second word is minus, it will also contain the FORTRAN record number of the TSX. The minus indicates that the error number may be duplicated in the error list and if the FORTRAN record number does not match the one picked up by the diagnostic from 1 to CS, the comparison with the error list continues.

When the match has been found, the diagnostic record number is used to space the system tape to the correct record in the 4th file. If a match is not found in the error list, the main record will then read in DØØ2 which concerns unlisted stops. The pertinent diagnostic record is then read in over the error list and the main record transfers control to it.

Each of these records is set up to handle information about one error, or one specific type of error, only. Usually, this is done by straight forward coding which makes use of the subroutines in the main record. The program instructions executed may obtain further information to be inserted into the error comment from tapes, cores, or the core dump. The error comment, which is contained in that particular diagnostic record in BCD, is then printed.

After all error comments have been printed, control is always returned to one of two points in the main diagnostic record. This will depend upon whether the error encountered was a machine error or a source program error.

The main diagnostic record spaces the System tape to either the machine error or the source program error record in the FORTRAN Monitor, depending upon the aforementioned error return. The diagnostic then prints the end comment and transfers control to 1 to CS to read in the proper Monitor error record.

Operator options, if any, are printed by the Monitor error record on the 709, and by the diagnostic on the 704. The options will vary depending upon whether the system is operating in the Monitor mode or single compile mode.

THE DIAGNOSTIC RECORD FOR SECTION 1 DOUBLE PRIME

A few diagnostic records obtain information from an error list left in upper memory by the system record that has called the diagnostic. The diagnostic record for Section 1 Double Prime (DØØ3 in the 709, DØØ4 in the 704) is such a record. DØØ3 is unique in that it contains all of the error comments for Section 1 Double Prime, rather than just one comment. In the case of DØØ3, the information for a particular error is preceded by a flag. The format of the error list is described in the write-up for Section 1 Double Prime.

DØØ3 performs a table search in order to determine which subroutine within itself is to process the error currently being treated in the error list. This table search is done by comparing the flag in the error list with the first word of a two word entry in an error table. The first word in the error table entry is the location of a TSX to the error routine in Section 1 Double Prime. The second word is the location of the subroutine in DØØ3 for processing that type of error. When a match has been found, the table search routine transfers control to the proper subroutine. The subroutine extracts whatever information it may need from the error list and, like other diagnostics, uses the subroutines in the main diagnostic record for producing an error comment. When the subroutine has finished its task, control is returned to the table search routine. At this point the subroutine will have correctly incremented the index register that references the error list so that the table search routine will examine the next flag in the error list.

DØØ3 is also given a word count of the number of words in the error list by Section 1 Double Prime. The table search routine tests against this word count for an exhausted error list. If the error list has not been fully treated, the process of table searching, transferring to a subroutine, and returning to the table search routine continues. When all accumulated errors have been treated, DØØ3 then returns control to the main diagnostic record.

XIV

THE FORTRAN EDITORS

There is a considerable difference in the method of editing the system proper, the library and the diagnostic records. Therefore, there are three editors in the FORTRAN Editor deck, the system editor, the librarian, and the diagnostic editor. This section deals with the system editor and the diagnostic editor.

THE SYSTEM EDITOR

The system editor is a self loading program that edits the first and second files of the FORTRAN system. These two files comprise the Monitor and the Compiler. Both the Monitor and the Compiler records have the same format and are treated in the same manner by the editor. The system editor reads into, and writes from a buffer with a common origin. For this reason all editing is done according to a computed effective address rather than the locations specified on the absolute binary cards and on the Master tape.

The system editor must have a control card corresponding to each record and end of file mark on the Master tape. The only exception is the first record in the 709 which contains 1 to CS and the diagnostic caller. These two routines are contained in the 709 editor and are written as the first record on the System tape from the editor. The first record on the Master tape is spaced over and the editor proceeds to the card reading routine.

Because of the manner of execution, the remainder of this write up will be concerned with the details of the 709 system editor. However, the logic still applies to the 704 system editor.

Card Reading Routine

This routine reads only the 9 left and 9 right rows of the card and interrogates the 9 left prefix to determine what type of card is being read. A transfer to one of the other routines in the editor will occur depending upon the type of card being read. The routine transferred to will execute a load channel instruction to read the remainder of the card.

The routines that may be transferred to are the following.

Master Record Card Routine-

The 8 left and 8 right rows are read in order to obtain the transfer address, the first address of the record, and the last address of the record. From this information the editor computes the length of the record and tests for deletion of the record. If the record is to be deleted, the editor spaces over that record on the Master tape and proceeds to read the next card. If the record is not to be deleted and a previous record is in cores, that record is

written on the system tape before proceeding. The control words for the next record are then set up from the master record card just read and that record is read from the Master tape. Control is then returned to the card reading routine.

There are two words preceding each record on the System tape and the Master tape in the first and second files. These two words are written by the editor from the master record cards. They are the following:

IORT Load Address,,Word Count of Record

TXI Transfer Address,,Record Number

The IORT command is used by the editor and 1 to CS to read the record from the Master or System tape. The TXI is used by 1 to CS to commence execution of the record. The record numbers read from the Master tape are in multiples of ten in order to allow for the insertion of new records. Therefore, record numbers on the System tape are 10_{10}, 20_{10}, 30_{10}, 40_{10}, ..., 410_{10}, 420_{10}, 430_{10}, and should be interpreted as 1_{10}, 2_{10}, 3_{10}, 4_{10}, ..., 41_{10}, 42_{10}, 43_{10}. A new record inserted after record 2 or after record 41 would appear as 21_{10} and 411_{10}, and should be interpreted as record numbers 2.1_{10} and 41.1_{10}, respectively.

New Record Routine -

In the case of a new record, the procedure is the same as for a Master record with the following exceptions.

1) A record is not read from the Master tape.
2) The last record number read from the Master tape is incremented by 1 and assigned as the new record number.

Program (Absolute) Card Routine-

The effective address in the buffer is computed and used to read the remainder of the card into its proper location in the buffer. Control is then returned to the card reading routine.

End of File Card Routine-

The end of file mark on the Master tape is spaced over and an end of file mark is written on the System tape. Control is then returned to the card reading routine.

End of Editing Card Routine -

Sense light 1 is turned on to signal the librarian that the first two files of the system have been edited. If the librarian finds sense light 1 off, it will copy the first two files from the Master tape onto the System tape. The system editor then simulates a load cards sequence to load the librarian. If an end of file is read from the card reader the editor will come to a final halt.

DIAGNOSTIC EDITOR

The diagnostic editor edits the 4th file which contains records pertaining to error conditions occurring in Sections 1 prime and on through the rest of the system. This editor is loaded by the load-button sequence programmed at the end of the librarian which, in the 709 only, turns on sense light 1 to signal that the first 3 files have been written on the System tape. It is self loading if used alone, and, finding sense light 1 off, will rewind and copy the first 3 files and read the first record of the diagnostic from the Master tape before reading the next card. The 3-file copy device is not a part of the 704 diagnostic editor.

Ideally, the 4th file requires no cards in the Editor deck except the diagnostic editor and the card which signals the end of editing from the card reader. In this case, the editor will copy all of the 4th file from the Master tape onto the System tape. However, records are changed, deleted, added, and any of these operations requires a diagnostic master record card. This card is distinguished from the absolute correction cards by the column 1 punch in the 9 left row. The 9 left address contains the number of the record the editor is to read into memory before proceeding. Any records preceding this specific record on the Master tape, are automatically copied onto the System tape. The editor reads the 8 left row of the master record card and resets the parameters of the record according to the decrement field, address of the first word; and the address field, address of the last word.

Absolute correction cards are read until another master record card is encountered. The contents of absolute cards are read into the actual locations specified by the load address, and a checksum is computed. The current buffer is written on the System tape unless the previous master record card has caused the record to be deleted by setting its length zero or minus.

A new record can be substituted for a deleted record, or added to the file, by using master record cards and absolute cards. No attempt is made to read another record from the Master tape once the end of file has been read from this tape. However, new records may be added to the System tape from cards after the end of file has been read from the Master tape.

On the other hand, if the end-of-editing card has been read before the end of file mark on the Master tape, the editor automatically copies onto the System tape any records remaining on the Master tape.

APPENDIX I

FORTRAN TAPE STATUS AT END OF SECTION

(This configuration holds only at the end of the given Section)

Tape 2 (704) - Tape B2 (709)

File	Contents				Written by Section	Overwritten by Section***
1	SOURCE PROGRAM (BCD) - 1 FORTRAN Statement card/record				PRE-1 (Card to tape)	
2	COMPAIL - 100 words/record				I'-704,* 709/8K I-Pass II-709/32K	VI
3	COMPAIL RECORD COUNT and FORSUB (if it exists)				I'	VI
4	Table Label	Table** Name	Maximum Number of words 704*, 709/8K	709/32K		
		FLOCON	450	1800		
	10	FORMAT	1500	6000		
		SIZ	580	2320		
5		END	5	15	I'	
	11	SUBDEF	180	180		
	12	COMMON	600	2400		
	13	HOLARG	900	3600		
	0	TEIFNO	750	3000		
	2	TIFGO	600	2400		
	3	TRAD	250	1000		
	1	TDO	750	3000		
	6	FORVAL	1000	4000		
	5	FORVAR	1500	6000		
	4	FORTAG	1500	6000		
	7	FRET	750	3000		
	8	EQUIT	1500	6000		
	9	CLOSUB	1500	6000		
6	DOTAG B - variable number of records--variable number of entries/record--9 words/entry				II	
7	DOTAG B RECORD COUNT				II	
8	DOFILE C - CIT's for A) subroutines				II	III
9	DOFILE C RECORD COUNT				II	III
8	ASSIGN CONSTANT				III	
9	FIXCON				III	
10	ASSIGN CONSTANT				V	
2	STORAGE MAP (BCD) FOR PROGRAM				VI	
3	SYMBOLIC LISTING FOR PROGRAM				VI	

*704/4K, 8K and 32K systems.

**In order as on tape.

***Any overwriting of file(s) obsoletes all information previously following it on the tape

APPENDIX 1
FORTRAN TAPE STATUS AT END OF SECTION
(This configuration holds only at the end of the given Section)
Tape 3 (704) - Tape B3 (709)

File	Contents	Written by Section	Overwritten by Section***
1	COMPAIL - 100 words/record	One	One-Prime
1	704* - Max. no of words - 709 8K only FORMAT 750 / NONEXC 750 / TST OPS 300 / TSKIPS 425 NONEXC 300 / TSTOPS 300	One-Prime	Two
1	DOTAG A - Variable number of records; variable number of entries/record; 9 word/entry; maximum of 1350 words DOFILE - INTERMEDIATE CIT's for DO STATEMENTS 704*-709/8K-- 100 words/record 709/32K -- 400 words/record	Two-Block 1 Two-Block 5	Three-Block 1
1	MERGED CIT's OF COMPAIL AND COMPDO-100 words/record	Three-Block 1	Five-Block A
2	CIT's FOR FORTRAN FUNCTIONS - 100 words/record	Three-Block 1	Four-Block 3
2	DOUBLE END OF FILE MARK	Four-Block 3	
3	TAGLIST - 15 words/record	Four-Block 3	
4	BBLIST - 6 words/entry	Four-Block 3	
1	CIT's including: DO FILEC, FORTRAN FUNCTIONS	Five-Block A	Five-Prime
1	CIT's	Five-Prime	Pre-Six
1	EIFNO (709 only)	Pre-Six	Six
1	BINARY OUTPUT (card image form) a. Program Card b. Binary Object Program c. Library Routines (if requested) d. Transfer Card e. EOF	Six	

* 704/4K, 8K, 32K Systems.
***Any overwriting of file(s) obsoletes all information previously following it on the tape

APPENDIX 1
FORTRAN TAPE STATUS AT END OF SECTION
(This configuration holds only at the end of the given Section)
Tape 4 (704) - Tape A4 (709)

File	Contents	Written by Section	Overwritten by Section***
1	Various table buffers written in the order in which filled. Each table is preceded by an identification label. (See tape 2/B2, files 4 and 5). An EOF is written only on the 709/32K system.	One	Two-Block 1
1	TRALEV - maximum number of words: 704*-709/8K 600 words/record 709/32K 2400 words/record	Two-Block 1	Three-Block 3
2	TAGTAG - 1 record/nest of DO's with tags; 4 words/tag entry COMPDO 100 words/record	Two-Block 2 Two-Block 6	Three-Block 3
1	MERGED CIT's OF COMPAIL, COMPDO, TIFGO	Three-Block 3	Six
2	CIT's for CLOSED SUBROUTINES FOR DOFILEC and FORTRAN FUNCTIONS	Three-Block 3	Six
1	CIT's (complete)	Six	
2	CLOSUB (1 record)	Six	

*704/4K, 8K and 32K systems.

***Any overwriting of file(s) obsoletes all information previously following it on the tape.