XEROX

Technical Information

Xerox FORTRAN IV

9300 Computers

Technical Manual

90 08 83A August 1965

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245
?13 679-4511

XEROX

Xerox FORTRAN IV

9300 Computers

Technical Manual

90 08 83A August 1965

Price: $5.75

Printed in U.S.A.

CONTENTS

STRUCTURE

OVERALL FLOW

SDS 9300 FORTRAN IV is a 3-pass compiler; that is, it passes over the source information (in one form or another) three times.

Pass 1 reads the source program, performs syntax analysis, registers symbols and other constructs, prints the source listing and all diagnostics, and outputs a parsed form of the source program to pass 2.

Pass 2 generates the object code and makes one pass of assembly on it.

Pass 3 makes the final pass of assembly, outputs the binary object code, and prints the object listing (with source lines intersperced) in a format similar to META-SYMBOL, a symbol table, and a summary listing.

There is also an allocation phase, called pass 1A, that performs all allocation and equivalence of variables and prints diagnostics for improper equivalences. It is logically considered part of pass 1, although it is physically located in the pass 2 core load. It does not make a pass through the source program.

This manual contains independent discussions of each of the passes and of the compiler debugging system, an elaborate executive routine which may be interfaced with the system during debugging of the compiler itself. Furthermore, the first section of the manual contains a description of some of the concepts, techniques, and routines that are utilized by all sections of the compiler.

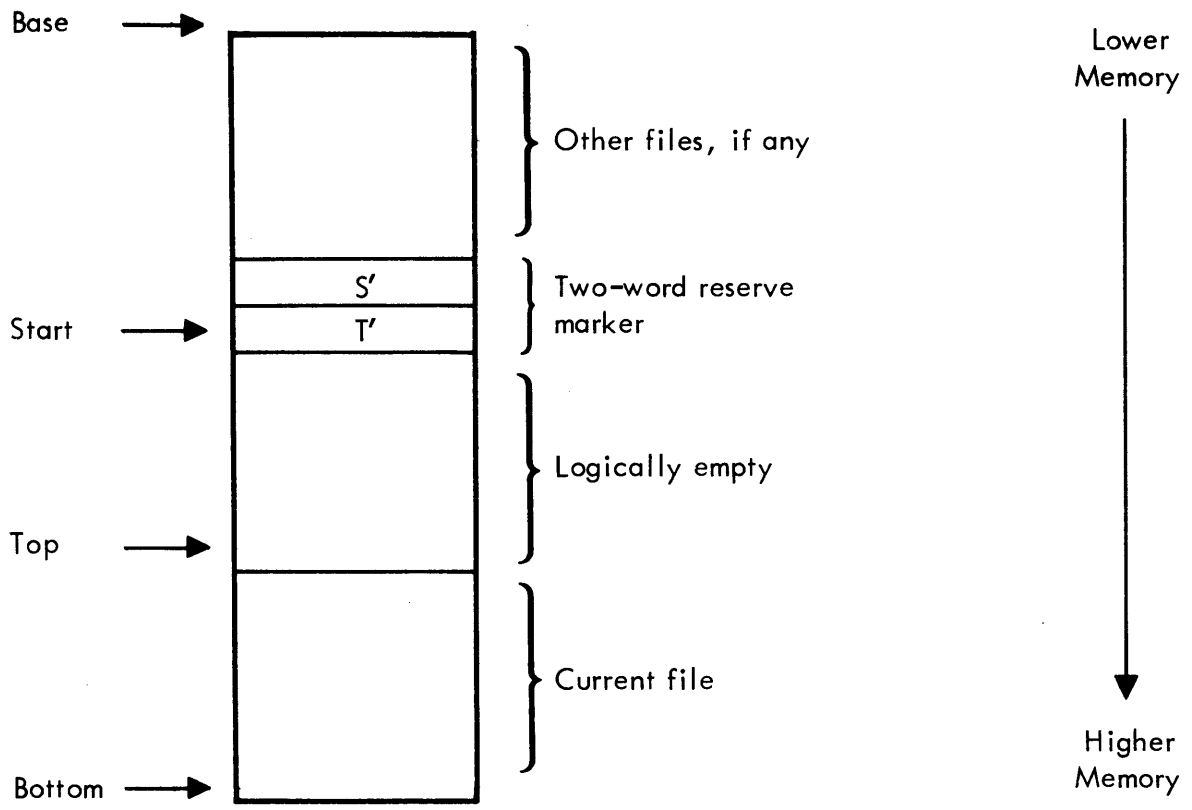
LISTS

Most of the compiler's processing and information saving is done on lists. Lists are used both to accumulate data (e.g., the symbol table) and to manipulate it (e.g., the work list). In the latter sense, they are commonly used instead of registers because this facilitates recursive use of subroutines.

These lists are not threaded lists. They are consecutively stored, for easy addressing by relative location within the list. Overflow is handled by dynamically moving the lists when room is needed.

Associated with each list are four parameters: base, start, top, and bottom. (There is also a fifth, called code (see "Pointers") and a sixth, called list flags.) These parameters are illustrated by the diagram on the following page.

The parameters for each of the lists are kept in individual cells in memory. Each parameter is an absolute address telling where that part of the list is.



The base is the word immediately above the word that is physically first in the list, i.e. the first word in the first file.

The start is the word immediately above the word that is physically first in the bottom (current) file.

The top is the word immediately above the word that is logically first in the bottom file. It is possible to logically remove entries from a file without altering the physical location of the start of that file.

The bottom is the last word on the list.

The "file mark" for these lists is called a reserve marker. This is a 2-word entry containing the start and top (relative to the base) of the previous file. The RSV (Reserve) and RLS (Release) POPs are the primary ones used in creating multiple files.

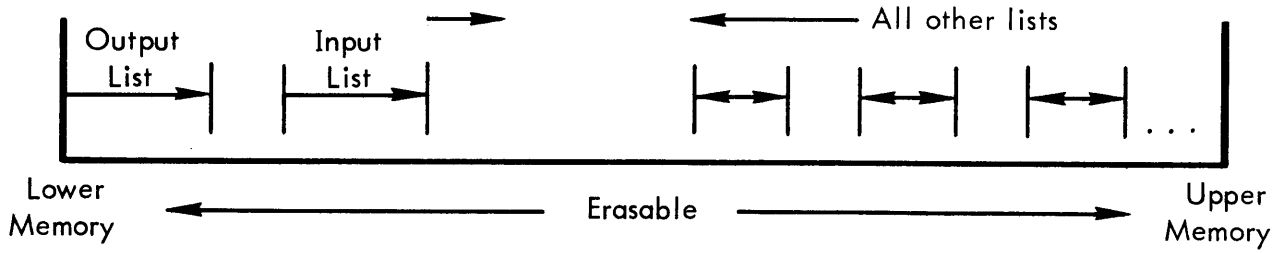
Note that, on unreserved lists, the start equals the base. Furthermore, if no information has been logically removed from the top of the list, the top also equals the base. This is the most common status for lists in general. In this case, the bottom is the only one that physically lies within the list.

The POPs used in manipulating lists are discussed later in this section.

REASSIGN MEMORY

The lists are stored in the erasable area of memory; that is, the storage that is not occupied by the compiler itself or the operating system. All the lists share this one area of memory; therefore, the storage for one list is not exhausted until all the storage is exhausted. Before the arrangement of the lists is discussed, it is necessary to explain a conflict in terminology related to them. As stored in memory, the bottom of a list is in a higher location than its top. This facilitates appending a source line to the bottom of the input list by just reading directly into memory, as well as simplifying some other manipulations. This means that "moving a list down" (in memory) corresponds to moving it towards its own top. Keep this anomaly in mind when reading the following section.

In concept, the lists occupy erasable memory as depicted in the following diagram:



The output list contains the intermediate output between passes. It is an ever growing list that may eventually push everything else up to the top of memory and use up all available storage. Only when this happens, is it dumped onto an intermediate tape. See below for details on this.

The input list contains the source input as it is read in and is generally only two cards (40 words) long. It moves up through memory, adding on at the front and taking off at the back. In doing so, it stays in front of the output list. The input list expands to greater than 40 words only when necessary to accommodate continuation cards, and it will thus accept any number of these until memory is used up.

All the other lists are initialized near the top of memory and push down. The exact mechanics of reassigning the various lists are as follows: (In this discussion, "a list" is assumed not to be the input or output list unless so specified.)

1. A list always requires room at its bottom, never at its top. When it needs room, the space between it and the base of the next list up is used if this is sufficient.

2. If that space is not large enough, the space between the input list and the lowest list is measured. If this plus the space in step 1 (above) is not sufficient, go to step 3. Otherwise, all the lists from the current one, down to and including the lowest list, are moved down the required amount. These lists are moved as a whole, maintaining the spaces between them. This space will be closed up only if necessary, since it is desired to retain some room for each list to grow.

3. If there was not sufficient space, the input list is moved down to just above the output list and step 2 is repeated.

4. If this has not obtained enough space, the lists are pressed upward: all the lists above the current one are packed against the top of memory, removing all spaces in between, and step 1 is repeated.

5. If step 1 is not successful, all the lists below the current list are packed against the current list (removing spaces) without moving the current list. Now step 2 is repeated.

6. If this procedure is unsuccessful, there is not enough room in memory. At this point, it is necessary to dump the output list onto a scratch tape. It is written out from its top to its bottom. Since it is written in constant length records, whatever is left over at the end that will not fill a complete record is left in memory. From this point on, the output list is periodically checked (see "File Code" in pass 1 and "Output Code" in pass 2) and an intermediate output record written whenever the list is large enough. In other words, the output list is no longer allowed to continuously grow as before. Thus, if this step has already been done once, no gain will result here.

7. Now the input list is moved down again (as in step 3) and step 2 is repeated. This must produce enough room or else memory has overflowed, which causes the job to abort.

8. If the list needing room is the input list, the procedure is basically the same except that step 5 is meaningless and step 2 just amounts to step 1.

9. If the list needing room is the output list, the procedure is somewhat different. Of course, if there is sufficient room between it and the input list, there is no problem.

10. If there is not enough space, the space between the input list and the list immediately above it is examined. If this space plus the space in step 9 is not sufficient, go to step 11. Otherwise, the input list is moved up. If possible, when moving it up, some room is left for the output list (about 20 words); otherwise, it is moved up the necessary amount.

11. If there is not sufficient space, all the other lists are pressed up, as in step 4, and step 10 is repeated.

12. If there is still not enough room, the position is the same as at step 6. So the output list is dumped on tape and step 9 is repeated (i.e., is there enough room between

the input and output lists?).

13. Finally, if this is not enough, the input list is moved up (as in step 10), and this must produce the required room or else memory has overflowed.

There are two further facts of interest pertaining to the reassignment of lists. There are two occasions when list parameters are changed without specifically being requested:

1. When information is taken off the top of an unreserved list, the base and the start are moved up to coincide with the top, thus freeing the words removed. Note the fact that this means pointers to this list are no longer valid. That this does not cause trouble is indicative of how seldom information is taken off the top of lists.
2. When information is taken off the top of a list with a reserve marker, a gap of logically empty space is produced, that is not available to other lists. If, however, all the information in the bottom file is removed, the top and bottom will be adjusted down to coincide with the start, thus freeing that space at the opposite end from method 1 above.

INITIALIZE LISTS

This routine is used at the very beginning of each compilation, i.e., at the beginning of pass 1. It sets up the parameters of all the lists so that they are all empty, the input and output lists are at the bottom of erasable memory, and all the other lists are at the top of erasable memory.

It also puts one unusable word on the exit list (unusable as an exit) for use in setting answers at recursive level zero.

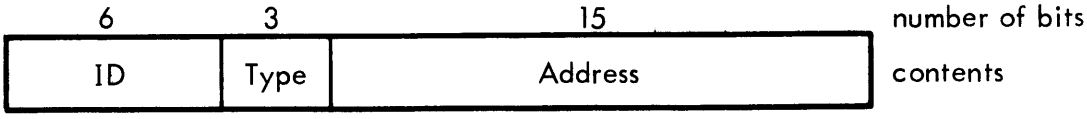
It initializes the input offset (see "Next Input Character").

If the DEBUG option was specified on the FORTRAN control card, this routine loads the F4DEBUG routine from the system tape and adjusts list parameters accordingly.

If S option (S in column 1) was not specified on the control card, the initialize routine adjusts the list parameters so that the lists will destroy the S-in-column-1 processing code in the compiler.

POINTERS

Pointers tell where things are located on lists. They are 1-word items as follows:



The ID field tells what list a thing is on, and in so doing, tells what kind of thing it is. The type field tells real, integer, etc., whenever this information is relevent. The address field gives the position in the list, relative to the base.

There are three kinds of pointers, corresponding to three ranges of values of the ID field:

1. Ordinary list pointers. Here the ID field is simply the list number. The first list is number 00, the next is 01, etc.
2. Plex pointers. The middle range of values (starting after the highest list number) points to the plex list. The plex list has a number, but it will not generally appear in the ID field. Instead, it will indicate what kind of plex this is - sum, product, function, etc. The thing to which it points on the plex list is a variable-sized group of words, the first of which tells how many words are in the group.
3. Symbol table pointers. Pointers to the symbol table will not generally contain the symbol table's list number, but instead will be derived from the third word of the item, which tells what kind of symbol it is (see "Identifiers").

There are two ways in which pointers are formed:

1. The BOP (Bottom Pointer) POP builds a pointer to the word which is about to become the bottom of the list. The upper nine bits of such a pointer are derived from a table called CODE which contains one word corresponding to each list. One normally executes the BOP just before putting a thing on a list.
2. The SER (search) POP builds a pointer to the thing it found. When the symbol table is searched, the pointer is derived from the third word of the item found, so that the pointer indicates what was found as well as where.

THE COMPILER INTERPRETER

The SDS 9300 FORTRAN IV Compiler is written almost entirely in an interpretive language. The interpretive instructions look very much like ordinary machine instructions; each has an operation code, an address, and tag and indirect bits. The only difference between the interpretive instructions and 9300 machine instructions is that bit 1 is interpreted as a seventh operation code bit instead of selecting index register 2 or 3. This allows more than 64 operation codes as well as the use of index registers 2 and 3 in the interpreter itself.

A few of the instructions are identical to machine instructions and retain the same operation codes: BRU, SKR, MPO, MPT, EXU, LDB, STB. Most of the other instructions are quite different. The subroutine jump JRS (Jump Recursively to Subroutine) places the return address on a push-down list, called the exit list, and jumps to the address specified. To return from a subroutine, one branches to a special location in the interpreter called EXIT, which removes the bottom entry from the exit list and goes there.

All the testing instructions except SKR return their answers in the answer flag. Two instructions, JAT and JAF (Jump if Answer True, and Jump if Answer False), test this flag. This way of doing tests makes it unnecessary to test the answer flag immediately after doing the test. One can do several more instructions before JATing or JAFing to test the previous answer. This often removes the necessity of having identical code in two branches. The flag is made more convenient by having a new one for each level of subroutine calling. Every time a JRS instruction is executed, the old answer is saved (on the exit list) and will not become active again until exit. Thus it becomes impractical to carry any answer into a subroutine as an argument, but it is quite practical for a subroutine to return an answer as a result. For example, suppose a subroutine A calls subroutine B just before exiting. It is unnecessary for A to have

```
JRS  B
BRU  EXIT
```

Instead, it is standard practice for A simply to BRU to B. Even better, subroutine A can sometimes be placed immediately before B and simply "fall into" B. This practice saves both time and space.

POPs are not the only things which can return answers: subroutines which have been called with JRS may also return them. Instead of branching to EXIT, one goes to EXIT TRUE or EXIT FALSE.

The exit list is not the only push-down list built into the interpreter; there are about 68 others. The most important of these is the work list; it is the "A register" of the interpreter in that most of the work is done there. The active end of any list (see Lists) is called the BOTTOM, and the bottom of the work list is where most arithmetic is performed, most tests are made, etc. The work list is also used to carry arguments into subroutines and often to return results.

The FET (Fetch) POP picks up the word addressed and appends it to the bottom of the work list. The bottom element of the work list is called W0, the previous word W1, etc. Thus, when FET is executed, the word which was W0 becomes W1, and the word fetched becomes W0. Corresponding to FET is a STO (Store) POP which stores W0 and removes it from the work list - - thus making W1 into W0. There is also an STK (Store and Keep) POP which stores W0 but does not remove it from the work list, and there is a GET POP which fetches a word from memory and replaces W0 with it. There is XCH (Exchange) which swaps W0 with something. It frequently addresses something else on the work list, although it is not required to do so.

It is possible to move things from the work list to other lists and back again. MON (Move On) takes W0 and moves it onto the bottom of some other list. MOF (Move Off) removes the bottom of some list and appends it to the work list, provided that there is anything on the other list to get. Thus MOF returns an answer true if it got anything or an answer false if the list was empty. This feature makes it convenient to control a loop with MOF. If a loop is supposed to do the same thing with all the items of a list, it is not necessary to know just

how many there are. One simply keeps moving off items until there are no more left.

It is possible to copy a whole list to another list. It takes two POPs to do this, one to tell which list to copy from and one to tell which list to copy to. To make a copy of the work list on the term list, for example, one writes:

```
ADR  TERMLIST
CPY  WORKLIST
```

This operation does not destroy the work list, nor does it destroy the previous contents of the term list; it appends a copy of the work list, however big it may be, to the bottom of the term list.

Items may be removed from the top of a list via TOT (Take Off Top). Like MOF, TOT returns an answer false and does nothing if the list is empty, or appends the word to the bottom of the work list and returns an answer true if there was anything to take. If a list contains just one word, TOT and MOF have the same effect.

The JRS (Jump Recursively to Subroutine) POP can be used to write a subroutine which calls itself. Sometimes it is necessary for a subroutine to have a nice clean list to work with every time it is called. This can be accomplished with the RSV (Reserve) POP, which causes a list to become "empty" while still retaining the information it used to have. RLS (Release) causes the list to revert to the state it was in before the RSV. When a list has been reserved, it has a top and bottom which are not confused with previous tops and bottoms; it may be MONed, MOFed, TOTed, etc., without disturbing the information which was put on before the RSV.

Reserve and Release may be used not only to make a list available recursively, but also to define files on a list. RSV essentially makes a file mark on the bottom of a list and RLS takes it off. In removing the file mark, RLS also discards any information which was put onto the list after the file mark. There are two other POPs which are variations on RLS: EMP (Empty) discards the information after the latest file mark but does not remove the file mark; UNR (Un-Reserve) removes the last file mark but does not remove the information which follows it. It pulls the mark out of the middle, thus combining the current file with the previous file.

Release is often combined with other operations. There are CAR (Copy And Release--equivalent to CPY, RLS) and CAE (Copy And Empty--CPY+EMP). MOR (Move Off and Release) and TOR (Top Off and Release) are a bit strange: they behave like MOF and TOT if the list is not empty, but release the list and return an answer false if the list is empty.

SER searches a list. For searching purposes, each list is divided into n-word items, the first m words of each being the key on which to search. Two parameters given at assembly time define the item and key sizes for each list. A typical searchable list in the FORTRAN compiler is the symbol table which has 6-word items of which the first two are the key. Item size can be whatever is desired; key size, in the existing version, can be 1, 2, 3, or 4 words.

When a list is searched, it is compared against a set of words called CENTRAL. If a list has 1-word keys, it is searched against integer central (INTCENT); if it has 2-word keys, they are compared against CENTRAL1 and CENTRAL2; if 3, against CENTRAL1, CENTRAL2 and CENTRAL3; and if 4, against CENTRAL1, 2, 3, 4.

SER searches a list from top to bottom (i.e., ignoring previous files). If it finds what it is looking for, it returns an answer true and appends a pointer onto the bottom of the work list, pointing to the item it found. This pointer indicates on what list and what position on the list the item is. If SER does not find something, it returns an answer false and leaves the work list alone.

Using the pointer, there are two ways to obtain the information in the item pointed to. Assuming that the pointer does not have to be saved and that it is in W0, one can use BNG (Bring). The address of the BNG POP is relative to the pointer; i.e., BNG 0 means bring the word pointed to, BNG 1 means bring the word following that, etc. In any case, the word brought replaces the pointer in W0, leaving the size of the work list unchanged.

If the pointer is to be saved, one can use the instruction FET W0 before the BNG POP. This will append a new copy of the pointer to the work list, leaving the old copy in W1. Another way of fetching a word relative to a pointer without destroying the pointer is with POX (Pointer to Index). POX addresses a pointer and places the address pointed to by the pointer in index register 1. It will remain here only until the next instruction, where it may be used to modify the address. Index 1 is not saved by the POPs, so it is not permissible to POX one place and expect the address to remain in the index register several instructions later. This restriction is not quite as bad as it sounds since storage for the lists is assigned dynamically, and a list is apt to move, thus invalidating the address. The fact that a list may move does not invalidate a pointer, since pointers contain relative addresses, not absolute ones.

One may POX the pointer (which may be in W0) then FET relative to the pointer using an index 1 tag. If the purpose is not to fetch but to store something near where the pointer points, STO, STK, STB, etc., may be used after POX instead of FET.

```
BNG     n
```

is equivalent to

```
POX     W0
GET     n, 1
```

When something cannot be expressed conveniently in interpretive language, one can return to machine language mode via BRL (Branch and Leave Interpretive mode) which branches to the location specified and is in machine language mode thereafter. BRL can be used both as a straight branch and as a subroutine call. On arrival at the branch location, index register 3 contains the location of the BRL in the address portion, and 1 in the increment portion. Thus, when BRL is used to call a subroutine, one exits from the subroutine by writing

```
BRX     INTERP, 3
```

thus returning to interpretive mode at the instruction following the BRL. BRL is used for many "POPs without operands." For example, there is a machine language subroutine to negate W0. It needs no operand address, so it is called with BRL. To enter interpretive mode but not return to the BRL, one writes

BRM ENTER (Enter Interpretive Mode)

thus entering interpretive mode at the location following the BRM. Here, unlike the BRX exit, it is unnecessary to have anything special in any of the registers. BRM ENTER is used infrequently in the compiler, since BRX INTERP,3 is faster and makes subroutines conveniently.

A subroutine called with BRL can also return an answer. Instead of BRXing to INTERP one branches (BRU) to POPEXTRU or POPEXFAL. Here, like BRX INTERP,3 it is a good idea not to have destroyed index 3.

It has been implicit throughout this discussion that W0, W1, W2, etc., are special addresses which may be used to refer to the bottom few items of the work list. These are converted by META-SYMBOL into special POPs which perform the address calculation. There are also some special addresses P0, P1, etc., which refer to the bottom few items on the parameter list (param list). P0 refers to the bottom parameter, just like W0; P0IND (P0 Indirect) refers to the address pointed to by the bottom parameter. To put things onto the param list, one uses the ADR (Address) POP, which appends the effective address of the POP itself to the bottom of the param list. To send parameters to a subroutine, one sometimes does a few ADRs before JRSing. The subroutine then refers to the parameters with P0IND, P1IND, etc., (or sometimes P0, P1, but these forms are not quite as useful). It is usually the subroutine's responsibility to get the parameters off the param list. It may do so by SKRing the PARAMBOT the appropriate number of times, or it may exit through DITCH1EX (Ditch 1 and Exit), DCH2EXTR (Ditch 2 Exit True),or DCH3EXFA (Ditch 3 Exit False). The word "ditch" implicitly means get rid of things from the param list, while the word "clear" refers to getting rid of things on the work list. There are exits which clear or ditch 1, 2, or 3 before exiting true, exiting false, or just exiting.

SORT (Sort) and FLIP (Flip) are also useful. SORT sorts the bottom file of a list, item by item, using the same keys as in SER (Search). Things are sorted in lexicographic (unsigned) order. FLIP turns the bottom file of a list upside down, word by word, so that the top becomes the bottom and the bottom the top.

Most of the other POPs are associated with some particular operation (as opposed to general list manipulating POPs) and are described under the appropriate pass.

PROGRAMMED OPERATORS BY CATEGORY

BUILD

BAF Build Absolute instruction and File

BAF builds an absolute instruction and puts it on code list. AXB is usually built this way. The absolute address is found in W0 and is removed from the work list.

BAM Build by Address Mode

Although this instruction is usually used for code generation, it does not actually build anything. It takes the mode field (bits 6-8) of W0, shifts it to the address field, and adds it to the effective address. It then executes the POP at the location addressed.

This instruction usually addresses a 4-word table of instructions which do the building. BAM does not disturb the work list, but most of the instructions it addresses remove W0.

BBA Build BMA, PZE to call a system library routine

The address of the POP tells what routine to call (addresses a 2-word BCD symbol). W0 tells the operand. BBA may build LDX, AXB, or EAX to handle the subscripting of the operand. It may add tag or indirect bits to the PZE. It discards W0.

BBR Build BRM to system routine

The effective address tells what system routine (addresses a 2-word BCD name). BBR builds only a BRM; it does not disturb the work list.

BEX Build EXternal reference other than BRM or BMA

BEX is used to build unusual references iike LDA 8DBL0. It addresses a 2-word constant which specifies both operation code and address, and it ignores the work list. BEX appends the two words addressed to the bottom of the code list.

BIC Build Instruction with Constant

BIC registers a constant and builds a reference to it. It is used for building things like SKG =0, FLM =-1.0, etc., (addresses a several-word item which tells operation code and constant). It ignores work list.

BIF Build Instruction and File

This is the basic "workhorse" of the instruction-building POPs. The lower nine bits of the effective address specify the tag and operation code of the instruction to be built. The operand is found in W0. If the operand is subscripted, BIF also builds the necessary LDX, EAX, etc., commands to handle that. It may add tag or indirect bits to the specified operation code. It may also build addends to expedite constant

subscripting. This is the only POP which calls itself.

BIF removes operand from W0.

BSI Build Special Instruction

Used mainly for building COPY instructions, BSI addresses a many-word constant containing the instruction in binary form and as a BCD string to appear on the object listing. Ignores the work list.

CONTROL

NOP No OPeration

EXU EXecUte

EXU executes the POP in location α. That POP may be another EXU, if necessary.

EFFECTIVE ADDRESS

ADR ADdRess to parameter list

The effective address (with 0 operation code) is appended to bottom of param list.

EAT Effective Address to Temp (EATEMP)

The effective address replaces $EATEMP_{9-23}$. Zero replaces $EATEMP_{0-8}$.

FEX Fail EXit

Effective address $\longrightarrow$ $FAIL\ EXIT_{9-23}$ $\quad 0 \longrightarrow FAIL\ EXIT_{0-8}$.

The FEX tells where to go when a fail occurs, provided that the fail was not under control of TRY. See Try-Fail in Pass 1.

POX POinter to indeX (to modify next POP)

This POP addresses a pointer. The effective address pointed to by the pointer is placed in index 1 where it can serve to modify only the immediately following POP.

JUMPS

BRL BRanch and Leave interpretive mode

Leave interpretive mode and branch to the location designated by the effective address.

This instruction is frequently used to call a machine-language subroutine, since the return address remains in index 3.

BRU BRanch Unconditional

Branches unconditionally to addressed location.

JAF Jump if Answer False

Jump to location α if answer flag is false. The answer flag is recursive and is found in bit 0 of the bottom word on the exit list (0 = false, 1 = true).

JAT Jump if Answer True

Jump to location α if answer flag is true. The answer flag is recursive and is found in bit 0 of the bottom word on the exit list (0 = false; 1 = true).

JRS Jump Recursively to Subroutine

Append return address to exit list and jump to the addressed location.

TRY TRY

This has to do with syntactic analysis which may fail. Basically, it behaves like JRS, but converts EXIT to EXIT TRUE and FAIL to EXIT FALSE. See Try-Fail in Pass 1.

GENERAL LIST OPERATIONS

CNT CouNT

CNT appends to the bottom of the work list the number of items in the bottom file of the list addressed.

Bottom (α) - Top (α) $\longrightarrow$ Work List

EMP EMPty

EMP empties the bottom file from the addressed list. The reserve marker is not removed.

ZAP ZAP

ZAP completely empties a list, removing all files (i.e., sets BOTTOM = TOP = START = BASE).

FLIP FLIP list file upside down

FLIP inverts the bottom file of the addressed list. Inversion is done word by word, regardless of the item size of the list.

MCO Move Central 1 and 2 Onto list

MCO appends to the bottom of the addressed list the contents of CENTRAL 1 and CENTRAL 2.

MOF Move OFf bottom and onto work list

MOF moves the contents of one cell from the bottom of the addressed list (α) to the bottom of the work list. List α is reduced by 1 cell, and the work list is expanded by 1 cell.

In the event the bottom file of list α is empty, neither list is changed and the answer flag is set false.

MON Move ONto from work list

MON removes the bottom item from the work list and appends it to the bottom of the addressed list.

If the work list is empty, this POP does not function correctly and may destroy lists.

MOR Move Off and Release if empty

If the addressed list (α) is not empty, MOR behaves just like MOF and returns an answer true. If it is empty, it releases the list (removes a reserve marker) and answers false. In the latter case, the work list is unchanged.

RLS ReLeaSe list

Removes the bottom file (including reserve marker) from the addressed list. If there is no reserve marker to remove (i.e., if START = BASE), it merely empties the list.

RSV ReSerVe list

Appends a reserve marker to the bottom of the addressed list. This creates a new file on the bottom of the list, and the file is empty (BOTTOM = TOP).

SAL Save A List

This POP is used to remember the state of a list by saving its start, top, and bottom (relative to its base) on the save list. See Try-Fail.

SORT SORT a list

SORT sequences the bottom file of the addressed list into increasing order. The item and key size information come from the code table.

TOR Take Off top and Release if empty

If the bottom file of the addressed list (α) is not empty, TOR removes top word, appends it to the bottom of the work list, and returns an answer true. If the file is empty, TOR leaves the work list alone, removes a reserve marker from list α (if one exists to remove), and returns an answer false.

TOT Take Off Top

TOT removes the top item (from bottom file) of the addressed list (α) and appends it to bottom of work list. In the event bottom file of α is empty, lists are left unchanged and the answer flag is set false.

UNR UN-Reserve a list

This POP removes the last file mark from the addressed list, but does not empty the bottom file. Thus, it combines the bottom file with the previous file.

If there is no file mark (has been no previous RSV), the POP does nothing.

BOP BOttom Pointer

BOP creates a pointer to what would become the bottom of the addressed list if one more item were to be added to it. The pointer is appended to the bottom of the work list.

TOP TOp Pointer

TOP is similar to BOP, but creates a pointer to the top word in a list. It returns an answer true if there is anything on the list; otherwise, it returns an answer false and does not create the pointer. The pointer is appended to the bottom of the work list.

PNI Process Next Item

PNI is used in conjunction with TOP. It increments the pointer in W0 by the entry size of the list addressed to point to the next entry. If there is another item, it returns an answer true; otherwise, it returns an answer false.

LIST COPYING

CAE Copy And Empty

CAE copies the bottom file of the addressed list to the list specified by the previous ADR POP and empties the bottom file of the addressed list but does not remove the reserve mark.

CAE α is equivalent to CPY α
EMP α

It removes the bottom cell from the param list.

CAR Copy And Release

CAR copies the bottom file of the addressed list to the list specified by the previous ADR POP. Then it empties the addressed list and removes a reserve marker (if there is one). Thus, the list may not be empty when this POP is executed. It also removes the bottom of param list.

CAR α is equivalent to CPY α
RLS α

CPY CoPY

CPY copies the bottom file of the addressed list to the list specified by the previous ADR POP. It also removes the bottom cell from the param list.

SPECIAL ADDRESSING

PAD Parameter Address Direct

This is used in connection with special addressing through the param list. Bits 9 through 14 of the effective address specify what number is to be subtracted from param bottom to determine the new effective address. Bits 15 through 23 (of which 15 and 16 are always 0) specify which POP to execute with the new effective address. This POP is not normally written as PAD, but is generated automatically by the assembler when P0, P1, etc., are used as operands.

PAI Parameter Address Indirect

PAI is like PAD except that an extra level of indirect addressing is added at the end. It is generated by the assembler when P0IND, P1IND, etc., are used as operands.

WAD Work list Address Direct

This POP is generated by the assembler when another POP addresses W1, W2, etc. (For W0, the assembler makes an indirect reference to WORKBOT.) Bits 9 through 14 of the effective address are subtracted from (WORKBOT) to determine a new effective address. Bits 15 through 23 (of which 15 and 16 are zero) tell which POP to execute with the new effective address.

This POP causes the interpreter to call DEBUG twice with the same location in X3 - once with WAD as the operation code and once with the POP specified in bits 15 through 23.

CODE LIST

FIL FILe (fetch to code list)

FIL appends to the bottom of the code list the word in the addressed location.

FAD File ADdress

FAD is like FIL but files its own effective address rather than the contents of that address.

LOAD/STORE

LDB LoaD B

Load B register from the addressed cell. This is useful for SST, SSK, STB, and SME. Many other POPs destroy B, so it is unwise to try to keep something there for long.

LOD LOaD a list (for debugging)

LOD puts a string of words onto the bottom of the addressed list. The string is addressed by EATEMP (see EAT POP); the first word of the string tells how many words follow, and it is the following words which are put onto the list.

LX1 Load indeX 1

$(\alpha) \longrightarrow X_1$

The contents of the addressed location replace the contents of index register 1. The number loaded may be used to modify only the immediately following POP. After that, the interpreter has destroyed X1. Unlike LDX, this instruction has full addressing capabilities, and a tag means modification, not selection.

STB STore B

$(B) \longrightarrow \alpha$

The contents of the B register replace the contents of the addressed location.

MEMORY

MEF MEmory False (store zero)

Stores zero in the addressed location.

MET MEmory True (store ones)

Stores all 1's in the addressed location.

MPO Memory Plus One

$(\alpha) + 1 \longrightarrow (\alpha)$

The contents of the addressed location are incremented by 1.

MPT Memory Plus Two

$(\alpha) + 2 \longrightarrow (\alpha)$

The contents of the addressed location are incremented by 2.

SKR SKip Reduce

$(\alpha) - 1 \longrightarrow (\alpha)$

The contents of the addressed location are reduced by 1. If the result is negative, the next instruction is skipped.

TMT Test Memory True (set if negative)

Set answer flag true if the contents of the addressed location are negative; otherwise, set the answer flag false.

PLEX

FIC Flesh, Inherit, and Count (build plex)

Builds a plex from the bottom n items on the work list, where n = 1, 2, . . . 6. N is supplied in the 2-word plex description addressed by FIC. Traits are inherited only from the first of the n items (the one farthest from the bottom). The n items are removed from the work list, and a pointer to the plex is appended.

FIP Flesh and Inherit Plex (build plex)

FIP is like FIC except that the traits are inherited from the merge of all the words in the plex.

PLO Plex Open

PLO is used in connection with CIC and CIF to build a plex of variable size. PLO addresses a 2-word plex description constant which is placed in PLOWORDS and PLOWORDS+1.

PUL PULL plex to a list

W0 contains a pointer to a plex. PUL removes W0, then copies the items of the plex to the bottom of the addressed list.

CIC Copy, Inherit, and Count

CIC is used in connection with PLO to build a variable-sized plex. The bottom file of the addressed list (α) is built into a plex. Traits are inherited from all the words. The list α is left empty, but the file mark is not removed. Plex pointer is appended to the bottom of the work list.

CIF Copy and Inherit First term traits

CIF is like CIC except inheritance is done only from the first (topmost) word instead of all the words.

GROUP

COG COpy Group

COG builds a group from the bottom file of the addressed list (α) and appends the group on the bottom of the group list. A pointer to the group is appended to the work list. The bottom file of list α is emptied (the reserve marker is not disturbed).

Equivalent to:

```
BOP   GROUP LIST
CNT   α
MON   GROUP LIST
ADR   GROUP LIST
CAE   α
```

(Groups are much like plexes.)

PUG PUll Group

Assuming W0 contains a pointer to a group on the group list, PUG will copy the items (not including the count) from the group onto the bottom of the addressed list. W0 is removed. No change is made to group list.

PRINT

PRC PRint Character

PRC addresses an item in the character translate table of which the left six bits are the character and prints it (i.e., causes it to be set up in a line image).

PRQ PRint Quote

PRQ addresses a variable-length quote constant and prints it.

INPUT SCAN

CSA Character Scan with Answer

If the "current character" is empty, the next character from the input string is retrieved. Then "current character" is compared with the character in the addressed location of the character translation table. If they are equal, "current character" is emptied and the answer flag is set true. If they are unequal, the answer flag is set false.

CSF Character Scan or Fail

If the "current character" is empty, the next character from the input string is retrieved. Then "current character" is compared with the character in the address location of the character translation table. If they are equal, control is returned. If they are not equal, control goes to Illegal Syntax Fail.

CSK Character Scan and Keep

CSK is like CSA except that it does not "empty" the current character.

SOC Set On Character

If the current character is empty, SOC advances to the next active character and performs an SKA-type comparison with the word addressed and current character, thus testing various character flags such as DIGIT FLAG, LETTER FLAG, etc. If any 1-bits agree, the answer is true; otherwise, it is false. SOC does not advance beyond the character in either case.

QSA Quote Scan with Answer

QSA is like CSA, except it addresses a variable length quote constant, scans the character string from the current character onward, and asks for complete agreement

with the quote. If agreement is found, the scan pointer is moved beyond the quote and the answer true is returned; otherwise, the scan is returned to where it was and the answer false is given.

ERROR MESSAGES

ERC ERror on Current character

ERC appends an error message item to the bottom of the error list. The character on which the error occurred is the current character, i.e., the character number comes from current character count.

ERL ERror on Last character

ERL is the same as ERC except the character number comes from last active character count; i.e., the "bad" character is the last active character before current character.

ERW ERror on Work list

ERW is like ERC except that the character count comes from W0 and is removed from there.

SEARCH

SER SEaRch a list

SER can do single and double-precision searching on lists with different item sizes. From the code table it finds out how big the items are (how far from item to item) and how many words of each item to compare. If doing double-precision searching (usually for an identifier), it compares CENTRAL 1 and CENTRAL 2 with the first two words of each item; if doing single-precision searching, it compares CENTRAL 2 with the first word of each item.

SER searches a list from top to bottom; if the item for which it is searching appears in more than one place in a list, it will find only the first one.

If it finds the item sought, SER appends a pointer to the item on the bottom of the work list and gives the answer true. If it does not find the item (or if the list is empty), it makes no pointer and returns the answer false.

SER1 SEaRch single precision

SER1 is almost the same as SER. SER gets item size and key size from the code table, which contains information about each list. SER1 gets key size from the code table but sets item size to 1. Thus, if addressing a list with 2-word keys, it performs double-precision comparisons but advances by only 1 between comparisons - thus performing an overlapped search.

SER2 SEaRch double precision

SER2 is like SER1 except that it sets item size to 2.

SET

SNE Set if list Not Empty

Set answer flag true if bottom file of addressed list is not empty; otherwise, set answer flag false. Does not disturb any lists.

SOF Set On Flag

SOF expects a pointer in W0. This pointer may be to a list or may be a plex pointer. In either case, there is a flag word associated with the pointer. In the case of a list, the flag word is in a table called list flags and applies to everything on the list. For a plex pointer, the flag word is part of the plex on the plex list. SOF fetches the appropriate flag word and performs an SKA-type comparison with the contents of the addressed location. Answer is true if any 1-bits agree; otherwise, it is false. SOF does not disturb work list.

SOM Set On Mode equality with W0

SOM compares mode of pointer in W0 with mode field in the addressed location. The answer flag is set true if the comparison is equal; otherwise, it is set false.

The mode field is bits 6 through 8.

SOR Set On Range

This POP is used for checking the size of numbers written in such statements as SENSE LIGHT n, WRITE TAPE n, etc. These statements allow full expressions for n, but in practice are usually written with integer constants.

SOR expects a pointer in W0 which has been returned by Expression Scan. If the pointer is not to the integer constant list, the answer is true. It is assumed that a non-constant expression will be in range at run-time. If the pointer is to the integer constant list, the constant pointed to is compared with the contents of the addressed location (α) and ($\alpha + 1$).

If (α) $\leq$ Constant $\leq$ ($\alpha + 1$), the answer is true; otherwise, it is false. The numbers are compared as signed integers.

SOT Set On Test (compare W0 ID field)

The ID field of the pointer in W0 (bits 0 through 5) is compared with bits 0 through 5 of the addressed location. This is not an immediate value anymore. The answer flag is set true if they are equal; otherwise, it is set false.

SRC Set on Range of Central

SRC sets the answer flag true if (α) ≤ (Integer Central) ≤ (α + 1); otherwise, it sets the answer flag false.

SYMBOL TABLE

REG REGister

REG is used for setting up various traits of an item in the symbol table. The third word in a symbol table item contains a 6-bit ID (telling what sort of thing the identifier is), a 3-bit mode field, and 15 bits of miscellaneous flags. REG can affect any or all of these.

REG expects a symbol table pointer in W0. It addresses a 1-word constant which is essentially merged with the third word or the item pointed to. Bits 6 through 23 are really merged; if bits 0 through 5 of the constant are non-zero, they replace bits 0 through 5 in the symbol table. Otherwise, bits 0 through 5 are left unchanged.

Afterwards, bits 0 through 8 of the symbol table word replace the corresponding bits of W0.

REG changes the symbol table. Since the statement may later fail, it may be necessary to restore the symbol table. Therefore, REG records the change on the symbol table change list. An item on that list contains two words: a pointer to the word changed (not the beginning of the item, but the actual word) and the old contents of the word.

SOP Set On Permissibility

SOP expects a pointer to the symbol table in W0. It performs an SKA comparison with the permission word in the symbol table item (fifth word) and (α + 1). Answer is true if any 1-bits agree.

SOL Set On Label

SOL is like SOP, but expects a pointer to a label (statement number). It performs an SKA on the label item (first word) and α. Answer is true if any 1-bits agree.

SUP Set Up Permissibility

SUP expects a pointer to the symbol table in W0. It alters the permissibility word (fifth word in the item) by ANDing it with (α + 2) and saves the old permission word on symbol table change list for possible restoration (see REG).

OTHER TABLES

RGI ReGister Integer constant

RGI addresses an integer constant. It searches the integer constant list for the item addressed. If the item is found, RGI appends a pointer to the work list. If the item is not found, it creates a new item on the integer constant list and returns a pointer to that.

SUL Set Up Label

Like SUP, SUL expects a pointer to a label. Also, it performs a merge rather than an extract with the label item and (α + 1). It saves the old label item on symbol table change list for possible restoration. (See REG.)

WORK LIST

ADW ADd to W0

(α) + (W0) ⟶ W0

The contents of the addressed location are added to the contents of W0, and the sum is placed in W0. ADW may set overflow trigger, a word which contains 0 if no overflow, all 1's if overflow. Overflow is accumulative. ADW never turns the overflow trigger off, but may turn it on. SUW and MPW also use the overflow trigger.

ETW Extract To W0

(α) AND (W0) ⟶ (W0)

The contents of the addressed location are ANDed with the contents of W0 and the results are placed in W0.

BNG BriNG relative to pointer in W0

This POP computes the effective address of the pointer in W0, then adds to this address the effective address of BNG. This address is used to retrieve a word from memory and store it in W0, thus replacing the pointer there.

EOS Exclusive Or from W0 to Storage

(α) EOR (W0) ⟶ (α)

The results of an exclusive OR operation with the contents of the addressed location and the contents of W0 are placed in the addressed location. EOS removes W0.

FET FETch to work list

FET appends to the bottom of the work list the word in the addressed location.

GET replace W0

$(\alpha) \longrightarrow$ W0

The contents of the addressed location replace the contents of W0. If the bottom file of the work list is empty, this POP may destroy a nearby list.

MPW integer MultiPly W0

(W0) * $(\alpha) \longrightarrow$ W0

The product, formed by multiplying the contents of W0 by the contents of the addressed location, replaces the contents of W0. MPW may set the overflow trigger (see ADW) if overflow occurs in the signed integer sense.

DVW integer DiVide W0

(W0) / $(\alpha) \longrightarrow$ W0

The contents of W0 are divided by the contents of the addressed location, and the quotient is placed in W0.

SSK Selective Store and Keep W0

SSK uses the mask in B register to selectively store contents of W0 in the addressed location (α). Bit positions in α corresponding to 0 bits in B are not changed.

The work list is unchanged.

SST Selective STore and discard W0

SST uses the mask in B register to selectively store contents of W0 in the addressed location (α). Bit positions in corresponding to 0 bits in B are not changed. W0 is discarded.

STK STore W0 and Keep

(W0) $\longrightarrow \alpha$

The contents of W0 replace the contents of the addressed location. The work list is not changed. If the bottom file of the work list is empty, this POP does not operate correctly.

STO STOre W0 and discard

(W0) $\longrightarrow \alpha$

The contents of W0 replace the contents of the addressed location. W0 is then removed from the work list. If the bottom file of the work list is empty, this POP does not operate correctly.

SUW SUbtract from W0

(W0) - (α) ⟶ W0

The contents of the addressed location are subtracted from the contents of W0, and the difference is placed in W0. SUW may set the overflow trigger (see ADW).

SWA SKA test on W0

Set answer flag true if (W0) .AND. (α) not equal zero; otherwise, set it false. (α represents the addressed location.)

SWE Set if W0 Equal to memory, discard if equal

SWE compares the contents of W0 with the contents of the addressed location. If they are equal, W0 is removed and the answer flag is set true. Otherwise, the work list is not disturbed and the answer flag is set false.

SEK Set if Equal and Keep

SEK is the same as SWE except W0 is never removed.

SED Set if Equal and Discard

SED is the same as SWE but W0 is always discarded.

SME Set on Masked Equality

SME is the same as SWE but compares only those bits which are present in the B register (i.e., like SKM as opposed to SKE).

SWG Set if W0 Greater

The answer flag is set true if the contents of W0 are greater than the contents of the addressed location; otherwise, the flag is set false. Both are considered signed integers. SWG does not alter work list.

XCH eXCHange with W0

XCH exchanges the contents of the addressed location with the contents of W0. It may be used to exchange W0 with W_n; e.g., XCH W_n. Here the WAD POP calculates the address of W_n, and XCH does the exchanging.

If the bottom file of the work list is empty, this POP may destroy part of another list.

COMPILER OVERLAY STRUCTURE

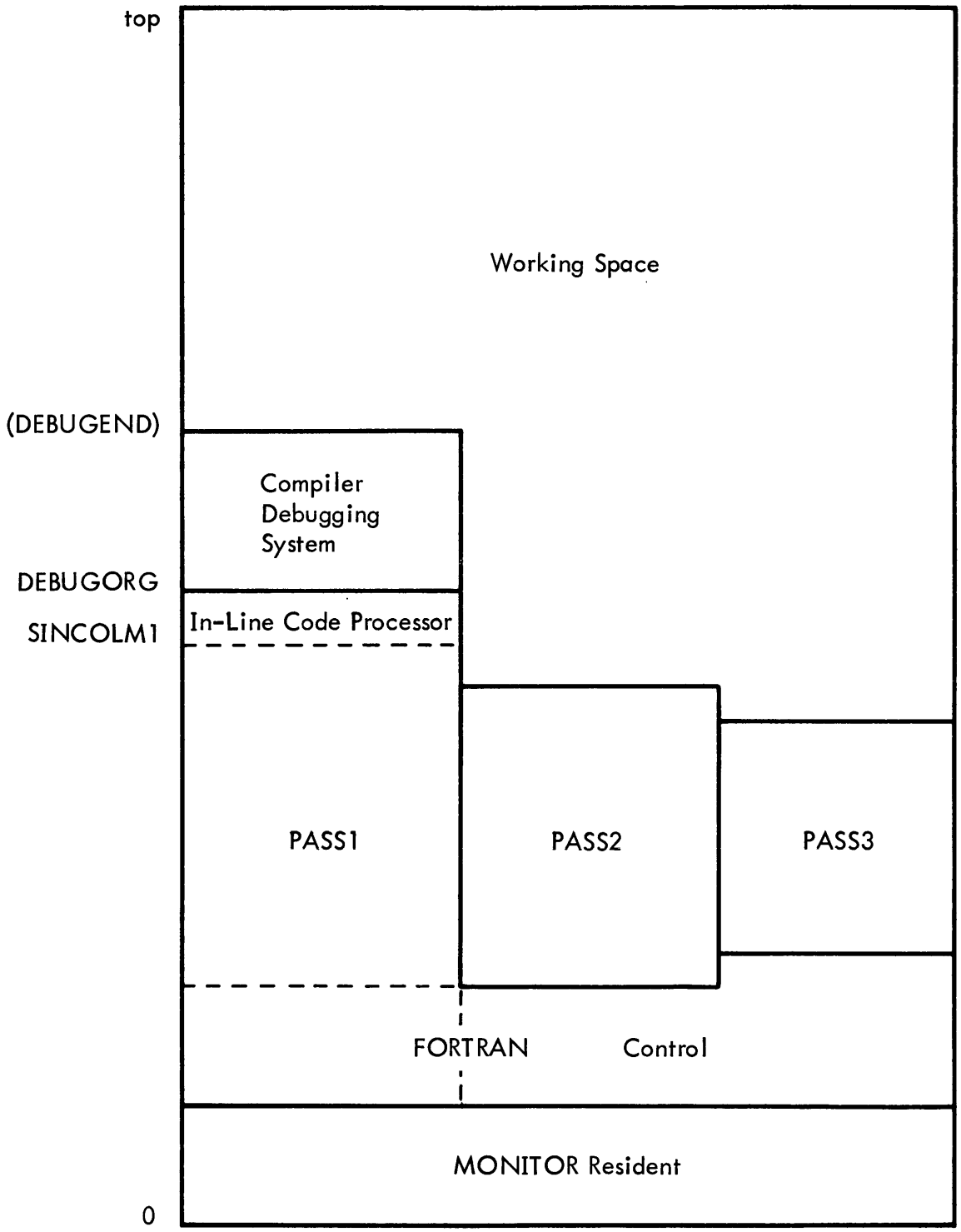
The compiler is on the system tape in the form of four records (not including label records) in the following sequence:

F/DB	(FORTRAN Debug System)
FORT	(FORTRAN Control, PASS1, In-Line Code Processor)
F/P2	(PASS2)
F/P3	(PASS3)

Upon encountering a △FORTRAN control card, MONITOR will load the FORT record and transfer control with a BRM to the first cell of FORT. If the debug system is needed, F/DB will be loaded from the system tape. FORTRAN will then establish the beginning of working space at one of three possible points:

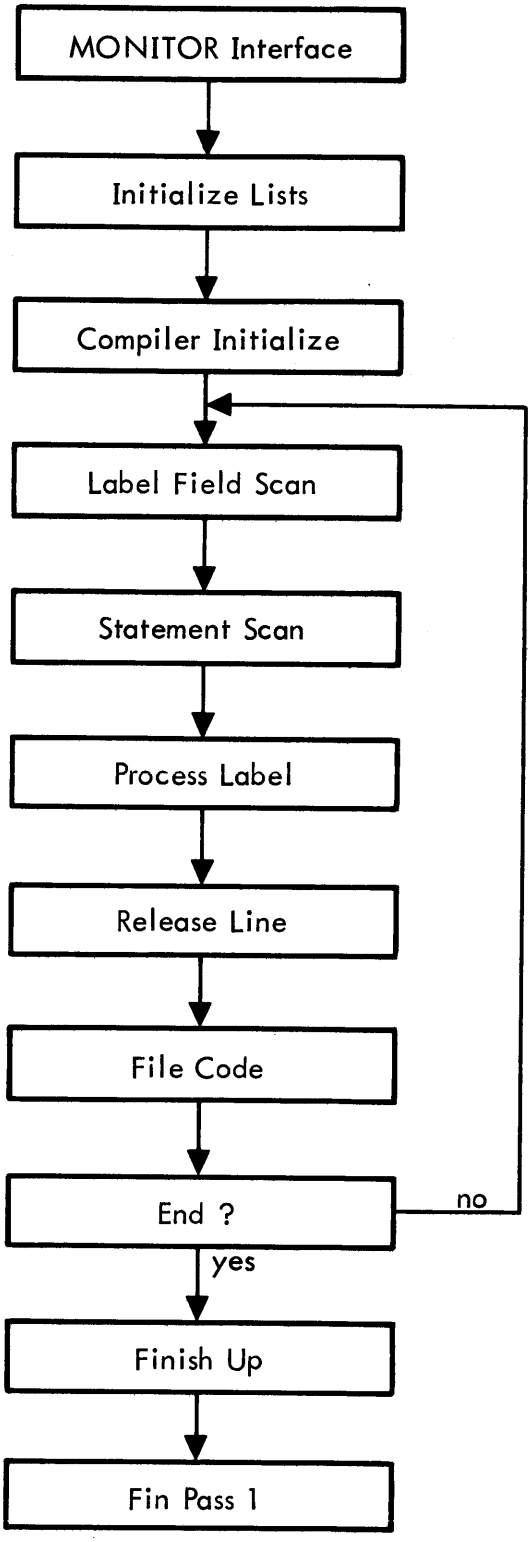
1. if DEBUG option - at DEBUGEND
2. if in-line code option and no DEBUG - at DEBUGORG
3. if no in-line code or DEBUG option - at SINCOLM1.

Pass 1 will then process the source program, and the FORTRAN control will load F/P2 and transfer control to pass 2. At the completion of pass 2, pass 3 will be loaded. At the completion of pass 3, control will be returned to MONITOR.


top
Working Space
(DEBUGEND)
Compiler
Debugging
System
DEBUGORG
SINCOLM1
In-Line Code Processor
PASS1
PASS2
PASS3
FORTRAN
Control
MONITOR Resident
0

PASS 1

OVERALL FLOW



MONITOR INTERFACE

The MONITOR interface section receives, in X2, the word indicating which control card options have been selected on the △FORTRAN card. It sets up compiler triggers (true or false) which can be tested during compilation, either in interpretive mode or out.

The options which may be selected are the following:

LS	List Source
LO	List Object
SO	Source Out
BO	Binary Out
GO	Go tape Out
ASA	ASA allocation mode
X	Compile X cards
S	Accept S in column 1 (in-line Symbolic code)
EI	Encoded In (ignored but warning message printed)
EO	Encoded Out (ignored but warning message printed)
C	Compatibility mode (interpreted as ASA compatibility)
DEBUG	Activate compiler debug trapping mode

The following diagram indicates the bit configuration for the above options. (Although other bits in the word may be set (e.g., bit 4), only those specified are tested by the compiler.)

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Value	0/1	0	0	0	0/1	0/1		0/1	0/1	0	0/1	0/1	0	0/1	0/1	0/1	0/1	0/1	0	0/1	0/1	0	0/1	1
Option	ASA					C		DEBUG	SO		LS	EO		GO	BO			LO		X	EI		S	SI

SI is set automatically

0 = Option not requested
1 = Option requested

Registers upon entry to compiler:

(X1) = Control card buffer address
(X2) = Processor options
(X3) = Processor entry point address

COMPILER INITIALIZE

Set up element size table according to either SDS or ASA storage mode allocation.

Initialize the following:

Input character count = -1
Last active character count = -1
Input stop count = -1
Not end trigger = TRUE
Main program trigger = TRUE
Subprogram trigger = FALSE
Current character = 0
Save active characters flag = FALSE
Use blanks trigger = FALSE
Number of X cards = 0
Any statements = FALSE
Executable statement trigger = FALSE
Block data trigger = FALSE
After IF trigger = FALSE
Multiple delta trigger = FALSE
Number of statements with errors = 0
Number of statements deleted = 0
Bob count = 0
Private temp counter = 0
Highest error severity (error level) = 0
Name list trigger = FALSE
Line count = 1

Initialize the LS print buffer to blanks and set pointers to start putting characters in at the beginning of it.

Rewind T1 and T2 (intermediate output tapes), set current output tape to T1, and set the output to tape trigger to FALSE.

If Sense Switch 4 is set, accept patches through DEBUG if present, otherwise through MONITOR PATCH routine.

LABEL FIELD SCAN

The primary purpose of label field scan is to analyze columns 1 through 6 of each statement, and to accumulate the label (if any) in columns 1 through 5. The label is put on the local label list or the non-local label list, depending on whether there is a dollar sign after it. The label is also filed on the code list for Pass 2. Unless column 1 contains a special character, column 6 must be a blank or zero to indicate that it is not a continuation card. (Continuation cards are not scanned by label field scan.)

The following characters are processed specially when they appear in column 1:

blank - Ordinary label field.

digit - Ordinary label field.

C - Comment card. No further scanning is done; the card is released immediately (see Release Card) and the next card is examined.

* - Comment card. (Same as C)

$ - Comment card. (Same as C)

✓ - Comment card. (Same as C)

X - X card. If the X option has been specified on the FORTRAN control card, the X is ignored and the card is processed as if column 1 were a blank. Otherwise, it is treated the same as C.

D - Double precision (7090 FORTRAN II). An error message is printed, and the D is ignored.

I - Complex (Imaginary). Same as D.

B - Boolean (7090 FORTRAN II). Same as D.

F - F card (7090 FORTRAN II). Uses external scan as if the F had been EXTERNAL instead. However, column 2 must be blank. This is to avoid statements such as FUNCTION A, if mispunched in column 1 instead of column 7, being treated as an EXTERNAL statement. After scanning the external names, label field scan returns with exit false to indicate not to scan for a statement.

△ - Control card or EOF read. Compilation is terminated, and a "Missing END card" message is printed.

S - Symbolic in-line code. The S option must have been specified. The compiler first tries OPDSCAN. If not an OPD, then sets S card trigger, scans label normally, and waits for statement scan to call symbolic code scan.

Label field also initializes the fail procedure for each statement and saves the lists which it may effect, namely:

Work list
Exit list
Local label list
Non-local label list
Symbol table
Code list

See "Try-Fail" for further information.

STATEMENT SCAN

The purpose of this routine, whose parts encompass most of pass 1, is to scan for one statement and generate the proper code and error messages (if any) for it. It is recursively reentrant since in the middle of it, if it is scanning a logical IF statement, it has to call itself. If label field scan found an S in column 1, a special routine is called at this point (described under In-Line Symbolic Code), and the rest of statement scan is not used. The S card scanner returns to statement exit.

Since FORTRAN statements are initially so ambiguous, the routine must be able to scan part of a statement, setting up lists and generating code, and then upon deciding that it really is not this kind of statement at all, restore all the lists, throw out the code, reset the scan, and start all over again. Thus, this routine saves all the lists which it can affect as well as certain other parameters, such as the character counts. (See Try-Fail.)

Most of the statements can be separated from one another because they begin with different special words. There are two exceptions: the assignment statement and the statement function definition (hereafter called an ASFD), which can begin with any name including those that identify the other statements. For example, a statement that begins

```
REAL M(J)
```

may be a REAL statement, or it may be an assignment statement or ASFD if it continues

```
REAL M(J) = J * SIN(J)
```

Which of these latter two it is depends on whether REALM has been dimensioned or not.

Thus, there are three basic ambiguities which must be resolved. (In some cases, there are more; see below.) Since only one of them can succeed, they could be tried in any order. In practice, they are tried in the following order:

1. Quote statements; i.e., statements that begin with a special quote (anything other than 2 and 3).
2. Assignment statement.
3. ASFD.

This order is chosen for the following reasons:

1. Assignment statements and ASFDs do not generally begin with special quotes, and it can be quickly determined that the initial characters do not match any quote, whereas almost all of the quote statements look like assignment statements initially (to wit, the above example).
2. Assignment statements are much more common than ASFDs.

3. There are a small number of actual ambiguities in the language. It is desirable that these be resolved in the direction of statements with special quotes. (See SDS FORTRAN IV Reference Manual, 90 08 49, on "Syntax Ambiguities.")

Statement scan uses a special POP called FEX (Fail Exit) to aid in this multiple statement trying. FEX simply saves its effective address in a special location that is used, whenever a statement fails, to determine what kind of statement to try next. Thus the procedure is as follows:

1. FEX Assignment statement.
2. Examine the first character in the statement and branch through a sieve to scan for the quote statements which begin with that letter. For example, if S is the first letter, the statements SUBROUTINE, STOP, and SENSE LIGHT are tried in that order (the order is usually chosen on the basis of likelyhood).
3. Each statement does a QSA (Quote Scan with Answer) for its particular quote, and if it is not found, goes right on to the next statement.
4. If none of the quotes succeed, the routine proceeds directly to assignment without failing, since no lists have been affected, and QSA keeps setting back the scan when it fails.
5. If any quote succeeds, the routine proceeds to analyze that statement, assuming that it is indeed one. Then, if something goes wrong, control goes to FAIL, which restores all the lists and the scan to the way they were at step 1 and proceeds to the latest FEX, namely assignment.
6. Assignment FEXes ASFD and, if it fails, will go there.
7. ASFD FEXes Illegal Statement and goes there if it fails.
8. Naturally, any statement which succeeds does not fail and will therefore return to statement scan at statement exit which releases all the information that has been saved (this enabled the routine to keep returning to the state of step 1) and exits to whatever called it, with the generated code and error messages (if any), piled on the code and error lists respectively.

A further discussion of this fail procedure is found under Try-Fail.

The above procedure assumes that no two quote statements can begin with the same quote. There are five exceptions to this rule:

1. ACCEPT - ACCEPT TAPE
2. PUNCH - PUNCH TAPE
3. READ - READ TAPE - READ INPUT TAPE - READ DISC/DRUM
4. REAL - REAL FUNCTION (or any other type)
5. Arithmetic IF - Logical IF - Device IFs (e.g., IF SENSE SWITCH)

These statements interrupt the normal sequence of FEXes. As an example, if the READ quote is found, and no left parenthesis (which unambiguously indentifies a FORTRAN IV READ, as opposed to a FORTRAN II READ), the compiler FEXes the FORTRAN II READ (cards) statement and scans for TAPE, INPUT TAPE, DISC, and DRUM. There is clearly no ambiguity between these. If none of these quotes is found, it goes to READ (cards) directly. If one of them is found, but the statement fails (e.g., READ DISC, List), the scan fails, which, because of the special FEX which was done, gives control to READ (cards) instead of Assignment. READ (cards) then FEXes Assignment statement again and proceeds. Thus, READ only inserts one extra FEX, as do all of the others except IF, which has the only true triple ambiguity. Logical IF is tried before the device IFs not only because it is expected to be more common, but also because of an ambiguity that exists between it and them (see Syntax Ambiguities, op. cit.).

The following flow chart illustrates the structure of statement scan. Note that if any statement succeeds, it does not proceed to the next statement, but directly to statement exit.

PROCESS LABEL

After all the necessary code has been generated for a statement, a check is made to see if there are any DO or REPEAT loops ending on it. Only if the statement has a label is this check necessary.

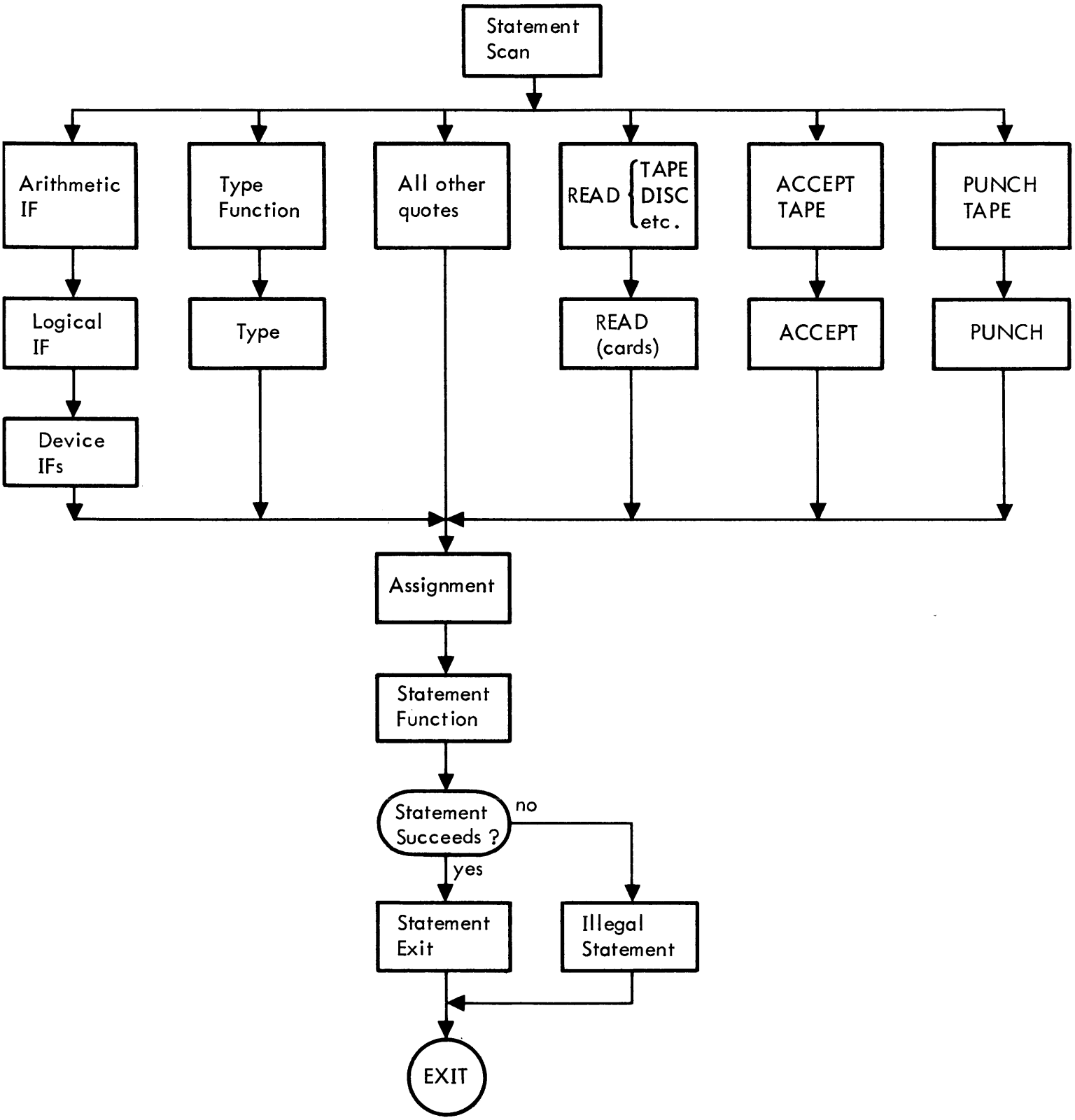
The loops must be closed in the inverse order from that in which they have been opened. Since the openings have been consecutively filed on the DO list, the bottom entry is closed first, and then the next and so on. (See DO list.) When, after n DO loops (where n may be zero) have been closed, the bottom entry on the DO list is not the current label, there can be no more legal DO terminations. However, the DO list is searched to determine if there are any illegally nested loops further up the list. If so, an error message is printed for each (with the delta under the label of the statement), and they are closed despite being improperly nested.

If any DO loops are terminated on this statement, and this is a type of statement that does not permit this (such as GO TO), a warning is printed to the effect that the loop has been terminated here anyway.

RELEASE LINE

After each statement has been completely processed, the following are done:

1. Print the line and any error messages. If LS is not specified, the statement will be printed on the LS device only if it has errors or is a SUBROUTINE, FUNCTION, or END statement. If LO is specified, these printed lines will also be compressed and passed on for printing during pass 3.
2. Remove the input line (including any continuation cards) from the input list. This line is no longer accessible, and the storage it occupied in the input list is now free (see Reassign Memory).


Statement Scan
Arithmetic IF
Type Function
All other quotes
READ
TAPE
DISC
etc.
ACCEPT TAPE
PUNCH TAPE
Logical IF
Type
READ (cards)
ACCEPT
PUNCH
Device IFs
Assignment
Statement Function
Statement Succeeds ?
no
yes
Statement Exit
Illegal Statement
EXIT

The test for continuation is done by noting the input character count, then requesting an active character and observing whether next active characeter has proceeded to the next card (see Input Scanning).

FILE CODE

After each statement has been processed, the plex list and the code list are put onto the output list for pass 2. During the processing of each statement, the intermediate output is built up on the code list rather than the output list so that it can be thrown away if the statement eventually fails. Furthermore, in some situations, such as I/O DO implied lists, it is necessary to retroactively insert some code in front of a mass of code that has already gone out.

At this point, also, a call on Dump Out ML is made. This routine determines whether the output list is being written on T1 (see Reassign Memory) and, if so, dumps the current output list if it is big enough to fill the resident buffer.

FINISH UP

Release the OPD list.

If this is the end of a FUNCTION or SUBROUTINE, its name, which has been a scalar during the program, must be reinstated as a subprogram name with the proper type, if any. (See Kludge list under Special Lists Set up by Pass 1.)

Check for any DO or REPEAT loops which have not been closed. Close these and print error messages for each. They are closed in the inverse order that they were opened.

If the last statement was not a transfer of some sort, put out a RETURN or STOP.

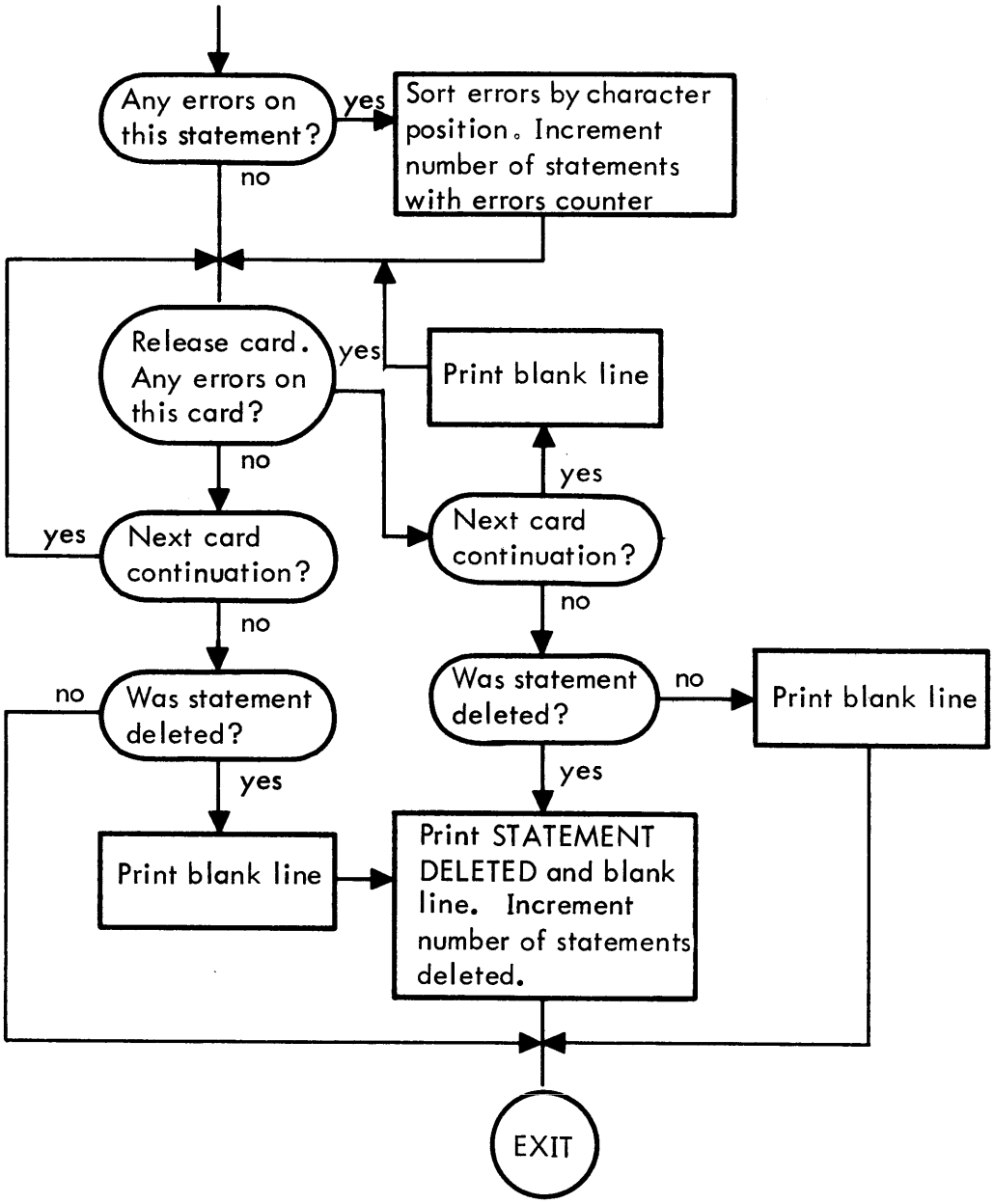
Print out any undefined labels.

File an END statement number on the code list.

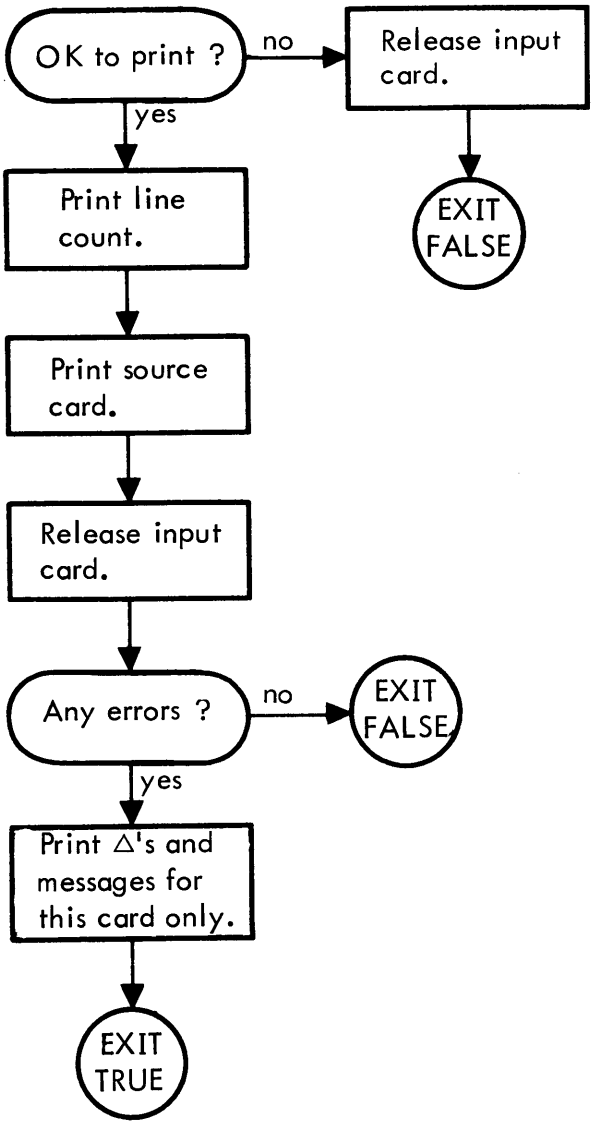
Make final pass through symbol table.

FINAL PASS THROUGH SYMBOL TABLE

At the end of each program, it is necessary to do some implicit classification of identifiers. Anything that is not explicitly classified yet is made a scalar if there is any reason to. That is, a name which has appeared in COMMON, for example, and nowhere else must be made a scalar in order to occupy the right amount of space in COMMON. However, a name that has appeared only in a REAL statement does not have to be classified at all since it is never used. Identifiers in the latter category are not allocated and appear as UNUSED in the symbol table printout in pass 3.



RELEASE LINE



RELEASE CARD

Furthermore, names which do not yet have a type are typed implicitly (by the I, J, K, L, M, N rule) unless they are subprogram names. This is because SUBROUTINE names must have no type. No distinction is actually made between SUBROUTINEs and FUNCTIONs, but any name which has been used as a function will already have a type.

Both of the above tests are accomplished by testing the permissibility of subprogram. Any name that has been defined or referenced as a subprogram or has not been used at all will be permissible to be a subprogram. Such names have neither their class or type classified here. All other names are arrays, multiple dummies, scalars, or unclassified (e.g. appeared in COMMON but undetermined whether array or scalar). The latter are classified implicitly as scalars. Then they are all given implicit type if they have no type already.

Next, if a NAMELIST statement has appeared with no list (indicating everything should be NAMELISTed), every non-dummy array and scalar is registered on the name list.

Finally, the fifth and sixth words in each symbol table entry are set up for use during allocation. GLOBAL variables have special information in these words. All other symbols have these words zeroed.

FIN PASS 1

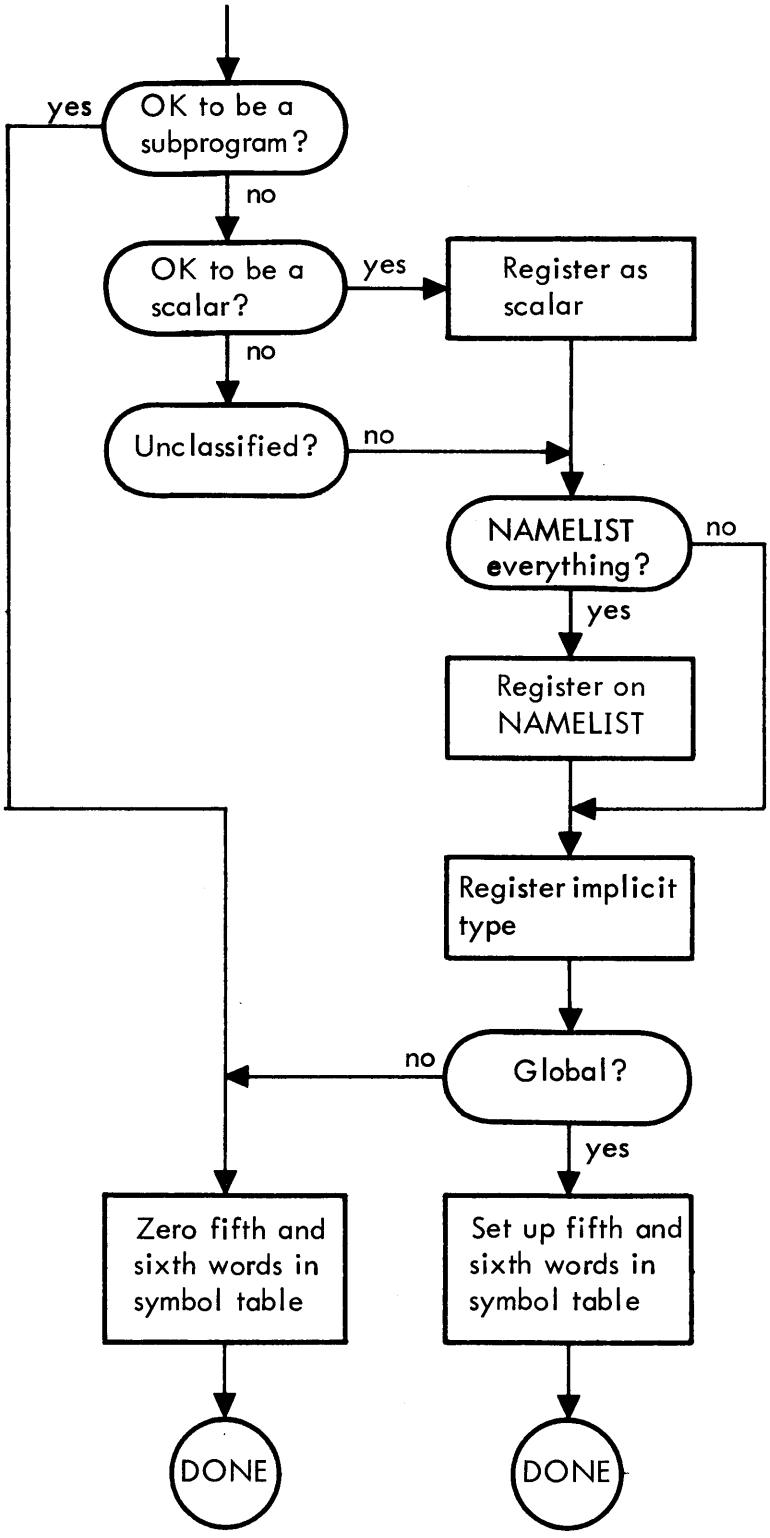
If T1 has been used, due to list overflow, write out on it anything that is left on the output list and rewind it.

Load pass 2.

IDENTIFIERS

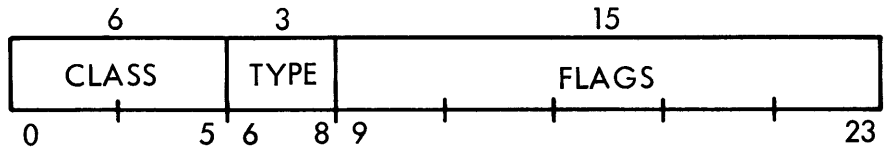
All identifiers, other than COMMON block names, are registered in the symbol table. (See COMMON statement for information on block names.) The symbol table is an ordinary, dynamically allocated list consisting of 6-word entries. The six words are as follows:

1. First four characters of name (left justified with trailing blanks)
2. Second four characters of name
3. Class, type, flags
4. Special information
5. Permissibility word (see Permissibility)
6. Special information



FINAL PASS THROUGH SYMBOL TABLE

The third word is arranged:



The class may be:

70 Unclassified
71 Scalar
72 Array
74 Multiple dummy
75 Subprogram

Type is:

0 Unknown
1 Integer
2 Real
3 Double precision
4 Complex
5 Logical

The last fifteen bits are various flags about the symbol, most of which are used only during code generation in pass 2:

Bit 9 I/O list unsubscripted array - or - Complex mixture
10 Addressable
11 A register
12 Constant
13 Double precision
14 Dummy
15 External
16 Global
17 Intrinsic
18 Multiple dummy
19 Signed addressable
20 Subprogram definition
21 Subscripted
22 Not used
23 Not used

All of the information in this third word is set up at one time or another by the REG (register) POP (see "Symbol Table POPs").

The fourth word is used only by two kinds of identifiers:

1. For arrays it contains a pointer to the array list, where the pertinent information (such as offset, multipliers, etc.) is stored (See Array List).
2. For intrinsic functions it contains a number identifying to pass 2 which intrinsic function it is. (The intrinsic functions are numbered from 1 up.)

The fifth word is described under "Permissibility."

The sixth word is used during allocation (see Allocation) and is also used temporarily in pass 1 for intrinsic functions. It contains in bits 6 through 8 the type of the function and in bits 9 through 23 the branch location for processing its arguments.

PERMISSIBILITY

The fifth word of each entry in the symbol table contains the permissibility bits. These are used to determine the legality of the appearance of a symbol in a given context. For example, a name clearly cannot appear in a COMMON statement if it is a subprogram name. It must be either a scalar or an array. Even then, however, it may not be legal; such as if it has already appeared in COMMON or is a dummy. Often there are many such characteristics that would have to be tested before it can be certain that a usage is permissable. So instead, each name has a group of bits associated with it, each of which indicates whether the name may be used in a particular way. The bits are set to all 1's when the symbol table entry is first created, and each succeeding appearance masks off those bits pertaining to usages which are no longer permitted. Thus, for example, when a name has been used as a dummy, the permissibility bit for COMMON is zeroed, as are those for GLOBAL, intrinsic function, etc.

Contained in the compiler is a table of traits, such as COMMON, dummy, scalar, etc. This table consists of 3-word entries:

1. REG word. This word is used for setting up the third word of the symbol table entry. It may contain class and/or type and/or flag fields.
2. SOP word (Set On Permissibility). This word is almost always a single bit, used for testing the permissibility of one particular thing.
3. SUP word (Set Up Permissibility). This word is an extract mask used on the fifth word in the symbol table (permissibility word) to remove the legality of certain characteristics.

SYMBOL TABLE POPS

REG REGister

This POP modifies the third word of the symbol table entry pointed to in W0, using the word addressed by the POP (i.e., the first word of the 3-word entry.) Basically,

this POP merges the trait addressed into the ID class word to give that symbol a trait (or combination of traits). However, the class field is treated somewhat specially. If the trait addressed contains any bits in the class field (bits 0 through 5), the entire field is inserted into the class word, rather than merging it in. This is due to the fact that a symbol with no class contains 70 in its class field, rather than 00.

SUP Set Up Permissibility

SUP does an extract on the permissibility word of the symbol pointed to in W0, using as an extract mask the third word in the trait group for the trait addressed. That is, the POP actually addresses the REG word, but adds 2 to the address and uses the SUP word.

SOP Set On Permissibility

This POP performs an AND between the permissibility word of the symbol pointed to in W0 and the second word in the trait group addressed. Instead of storing this result back into the symbol table, however, it tests the result. If the result contains any 1's, it returns with an answer true; otherwise, it returns the answer false. In other words, the SOP word in the trait group usually contains a single bit; this POP tests whether that bit is also present in the permissibility word of the symbol in question.

There is one peculiarity with respect to REG and SUP. They are normally used in the course of analyzing a statement, and they cause changes in the symbol table. If that statement fails, it is desired that all lists are restored to their condition before this statement. This is done with most lists using SAL (Save A List, see "Try-Fail Procedure") on the assumption that the only change made to lists is to append additional information on the bottom of them and, therefore, it is only necessary to remove that new information. Here, however, an established symbol table entry is being changed, right in the middle of a list. In order to restore these changes, a record is kept of them on the Sytch list (Symbol Table Change List). Whenever a symbol table entry is changed (i.e., with REG or SUP), two words are placed on the Sytch list. The first is a pointer to the symbol table word being changed; the second is the old value of that word. This information is used by the FAIL routine.

There are three additional POPs used to test the characteristics of symbols. They are used specifically to test the class, type, and flag fields of the class (third) word in the symbol table entry pointed to in W0.

SOT Set On Test (Really means set on class)

Returns answer true if the symbol pointed to (in W0) has the same class field (in its class, i.e. third, word) as the word addressed by the POP (i.e., the REG word); otherwise, the answer false is returned. For example,

```
SOT SCALAR
```

to see if a symbol is a scalar.

SOM Set On Mode (or Type)

Returns answer true if the symbol's type field is the same as the word addressed. For example,

SOM LOGICAL.

SOF Set On Flag

Returns answer true if the bits present in the word addressed are also present in the symbol's class word, flag field. Whereas SOT and SOM are like SKE, SOF is like SKA. For example,

SOF DUMMY.

Note that the characteristics which these POPs test are all set up by the REG POP. The characteristics SOP tests are set up by SUP.

INPUT SCANNING

The source input string is analyzed using five POPs and several subroutines. Unless otherwise specified, blanks are not significant and are skipped. The POPs are:

CSA Character Scan with Answer

If the current character is empty, the next character from the input string is obtained; otherwise, the POP uses current character. The routine returns the answer true if that character is the same as the one addressed by the POP or the answer false, if it is not. If the answer is true, current character is emptied.

CSK Character Scan and Keep

CSK is the same as CSA but does not empty current character in any case. This allows the character to be scanned again.

CSF Character Scan or Fail

CSF is the same as CSA but if the characters are not equal, instead of returning answer true, it goes to illegal syntax fail.

SOC Set On Character

SOC is like CSK except it does not look for one particular character, but a class of them; e.g. digit, letter, IJKLM or N. All 64 character codes are stored in a table called the character translate table. The upper six bits of each entry contain the BCD character code. The other bits are used for flags indicating whether the character is in each of several classes.

QSA Quote Scan with Answer

QSA is like CSA, but scans for equality on a variable length string of characters rather than a single character. It returns the answer true if all of the characters compare equal; otherwise, it resets the scan to the first character and returns the answer false. Thus, a QSA which is false has no effect on the scan position. The quotes against which the input string are compared are packed together, separated by dollar signs. The QSA POP scans the addressed field to the first dollar sign and compares all the characters which follow it up to the next dollar sign.

The following routines are used to scan for names and numbers:

Symbol Scan

This routine produces an 8-character, 2-word, BCD symbol and stores it in ID central (there are several "central" locations used for temporary storage). The first character must be a letter, the rest letters or digits. The scan stops when a character is found which is neither a letter nor a digit. Characters beyond the eighth are scanned and ignored. If the symbol is shorter than eight characters, it is left justified with trailing blanks.

ID Scan

This routine uses symbol scan to scan for a symbol and put it in ID central. Then it registers the symbol in the symbol table. This means, if the name is already in the symbol table, it returns a pointer to it (in W0). If the name is not already there, ID scan puts it there and returns a pointer to it.

Constant Scan

This routine scans for any type of constant (integer, real, double, logical, octal, Hollerith) other than complex, registers it on the appropriate list (integer constant list, real-double constant list, Hollerith constant list), and returns a pointer to it. (See "Expression Scan" for further description.)

Integer Scan

This routine scans for an integer and puts it in integer central. It does not register the integer or return a pointer. Furthermore, unlike constant scan, if the integer is too big, integer scan flags it and uses the 24 low-order bits left over, rather than floating it. This routine is used, for example, to get dimensions and statement numbers.

The following are low-order machine language subroutines which are used by all the above routines and POPs. These routines handle continuation cards automatically, so that the upper routines do not have to; they also read more input from the SI device when needed.

Ready Scan Character

This routine returns (in the A register) the current character, unless empty, in which case it gets the next active character (see below) and returns that; i.e., if the last character has not been used yet, it is still current.

Next Active Character

The next active character routine usually returns (in A) the next non-blank character in the input string. Sometimes, as in Hollerith fields, blanks are considered active. This is indicated by the use blanks trigger. If the next character (as returned by next input character) is an end-of-line character, it is returned as active unless the next card is a continuation, in which case the next active character is assumed to start in column 7 of that card. The next card is a continuation card if column 6 contains something other than a zero or blank and columns 1 through 5 contain only blanks or digits. There are two exceptions to this:

1. If just 'END' has been found so far, the end-of-line character is made active so that no attempt will be made to read the next card. This is revealed by the not end trigger.
2. An 'S' in column 1 (in-line symbolic code) has the same effect; continuation is not permitted. This is signaled by the S card trigger.

Next Input Character

This routine returns the next character in the input list and stores it in current character. If the list has run empty, it reads another card into it. Note that it does so only when that card is really needed. This fact is what keeps it from reading past the END card.

When it reaches column 73, it does not return that character. Instead it returns an end-of-line character and jumps the input character count to point to the first character in the next card. It determines this by testing the input character count modulo 80.

There are several counts used in locating input characters:

Input Character Count

This is simply a count of all the characters in the input string. The first column of the first source card is character zero, the first column of the second card is character 80, and so on. Thus every character in the input string has a unique number. This number is also used to attach deltas to the proper character when printing error messages. At any moment, the current value of the input character count determines which character is being scanned.

Input Offset

This count is used in combination with the input character count to determine the exact position in memory of any character. The input offset is essentially the word address of input character number zero. If this is added to the input character count divided by 4, the result is the location of the current character (the remainder of the division determines the position within the word). Initially, of course, the input offset is the address of the first word in the input list. As cards are removed from the back of the input list and it is reassigned downward, the input offset decreases. It may even become negative. Suppose, for example, that the current character is the 12000th, and it is presently stored in location 3000. The offset would then be -1000. The input offset is automatically adjusted by reassign memory whenever the input list is moved.

Input Stop Count

This is the character count of the last character in the input list at any given time. Thus, when the input character count becomes larger than the input stop count, the desired character is not in the list, and another card must be read.

Last Active Character Count

This is the character count of the most recent active character other than the current one. It is primarily useful in attaching error messages to a place when the desired item cannot be found after it.

Current Character

This usually contains the current character (full word, as taken from the character translate table). When it is zero, the next character to be processed is really the one associated with the input character count plus 1.

Note that the numbers defining the size of an input card (20 words) and the number of useful columns (72) are assembly parameters and could be changed.

ERROR MESSAGES

Most error messages are produced during the scanning of a statement and are associated with a particular character in the statement. There are exceptions, such as the "missing END card" warning, which are discussed later. This discussion does not concern them.

Error messages cannot be printed immediately when the error is detected. For one thing, if there are multiple errors on a statement, they have to be printed in the right order, which is not necessarily the order in which they are discovered. But more important, the error message may not be the least bit pertinent, or even valid. Consider the following example:

```
      EXTERNAL K
3     LOGICAL DO3K
      DO 3 K = .TRUE.
         ^ ^       ^
```

This last statement will be scanned first as a DO statement since it begins with the "DO" quote (see Statement Scan). As such, errors will be found at the three points indicated above; statement 3 has already appeared, K is not a scalar, and a DO parameter may not be logical. At this point, however, the statement fails and later turns out to be a legal assignment statement.

All error messages are saved on the error list at the time they are discovered. Each error produces a 2-word entry on the error list:

1. Character count of the character under which the delta is to be printed.
2. Location of the appropriate error message.

There are three POPs used in setting up these entries:

ERC ERror on Current character

ERC puts input character count on error list, except if the current character is an END character. In this case, it takes the last active character count, adds 1 and uses this. This is so that a statement such as,

$$X = A + \Delta$$

will produce the error message as shown instead of over at the end of the line.

ERL ERror on Last active character

ERL puts last active character count on the error list. This places the delta under the last (usually) non-blank character which was scanned before the present one.

ERW ERror on character count specified in W0

ERW is used to put an error message on a character previously scanned. The input character count can be saved in a location at any time, and later picked up to the work list, and an error message put under it using this POP. ERW removes the count from W0.

These POPs all do one further thing of importance. As discussed under Try-Fail, there are various ways in which a scan (either for a statement or under a TRY) can fail. If something drastic and unrecoverable happens, control can go directly to FAIL. Often, however, an error is detected which is fatal, but it is possible to continue the scan in the hopes of detecting any further errors which may be present, as in the example above of the three errors on the DO statement. So, there are two kinds of errors, fail errors and warnings. The type of error is not determined by the routine that is doing the scanning, but is inherent in the error message itself. Each error message is stored with preceding information indicating whether it is a fail error or not. (With respect to the severity of errors printed on the object listing, 1 is a minor error, and not a fail error; 2 is a major error, but also not a fail error; and, 3 is a major error and a fail error.)

When one of the above POPs is executed, it saves the information as to whether the error message it is addressing is a fail error or not. This is saved on the TRY list (See Try-Fail). Later, at either statement exit or when TRY thinks it has succeeeded, this condition is tested to determine whether to fail at that time. This is called a delayed fail.

As mentioned, there are error messages which do not pertain to a particular point in a scan and are put out only when it is certain that they should be. These are printed using the same lower level routine that is used to print the errors accumulated on the error list at the end of a statement (see Release Line), which is called Print Error Message:

Print Error Message

> This routine expects the address of the error message in the effective address temp (EATEMP, see EAT POP). If the error message is a fail error, this routine prints ERROR in front of it. Otherwise, it prints WARNING. Also, it updates the ERRLEVEL counter, which indicates the highest error severity, if this error has a higher severity than any previous error. Note that this is done here, when the error message is actually printed, rather than in the error POPs when it is unsure whether the error is valid.

TRY-FAIL

FAIL is a routine which enables the compiler to discontinue a particular scan and to rescan a string of source input. When this is done, all lists and counts must be restored to the way they were, so that it is just as if the first scan had never occurred. The information indicating how to do this is saved on two lists, the Save List and the Symbol Table Change List (Sytch list). The usual theory (to which the Sytch list handles the exceptions) in saving a list is that it can only be changed by having information added to it. Thus to save it, it is merely necessary to remember where the start, top, and bottom were. This information is recorded on the save list by the SAL (Save A List) POP:

SAL Save A List

SAL puts onto the save list the number of the list being saved, preceded by any of the following which are non-zero:

BOTTOM - TOP	(00700000)
TOP - START	(07000000)
START - BASE	(70000000)

The word containing the list number also contains, in the upper nine bits, the flags (indicated in parentheses above) to signal which of the three words are present. In addition, it always contains the POP bit (20000000) to signal that this is a saved list, not an individual item. For example, the save list entry for a list which has not been reserved but has had information taken off the top might look like the following:

00000005	(BOTTOM - TOP)
00000002	(TOP - START)
27700027	(List number 27_8)

Individual values that are not lists are saved using a 2-word entry on the save list. The first word is the old value and the second word is its location. Note that bit 1 is not set, indicating that this is not a saved list.

This method of saving lists does not work when they are being changed not by having information added onto the bottom, but by having values changed which are already in the list. The only lists to which this happens are the symbol table and the label lists. These lists, in addition to being saved in the normal way, are protected by the symbol table change list. This is a list of 2-word entries, set up any time a word in one of these lists is changed. The first word is the old value and the second word is a pointer to the location in the list. (This is discussed in greater detail under Identifiers.)

The FAIL routine, then, performs the following actions:

1. Replaces the symbol table and/or label table entries that have been changed (by SUP, REG, SUL) to their original states. Empties (but does not release)

the Sytch list, since it will never be necessary to recover this way again.

2. Restores everything that has been saved on the save list, but does not release it yet because this information may have to be recovered again (in Statement Scan, not TRY). Entries on the save list are either single values or lists.

3. When FAIL is finished, it exits to the location specified in FAILEXIT. This address is always one of the single-value items which is restored. It is set up either by TRY or Statement Scan. (See below.)

The two ways in which information is saved, for later failing, are in the TRY POP and in statement scanning. These will be discussed separately.

TRY TRY

TRY gives the ability to do a non-fatal JRS. Normally, in scanning a statement, if a fatal error appears and the scan fails, it is assumed that the input string being scanned is not the type of statement being scanned for presently, and the next kind of statement is tried. (This is explained in Statement Scan and below.) Sometimes, however, ambiguities arise within one particular statement, where it is necessary to try more than one interpretation at a given point. The TRY POP gives the compiler the ability to say, "JRS to this routine and if does not fail, return just as with JRS, but with answer true. If, however, this routine fails, do not proceed to the next kind of statement; instead restore things to the way they are right now and return here, but with answer false." Thus if a TRY fails, it is possible to proceed as if the TRY had never been done and scan for some other construct.

Before going to the routine being tried, TRY calls FEX-TRY Save, which is always used in preparation for failing.

FEX-TRY Save

This routine does the following:

1. Reserves the save list, so that when fail occurs, only the information from this try will be saved and not that which has been previously saved.

2. Saves (on the save list) the work list, the FAILEXIT, input character count, current character, and last active character count.

3. Puts two zeros on the try list. The first of these is the delayed fail flag and the second is the automatic succeed flag (see below).

4. Reserves the error list and the Sytch list. If the scan fails, the compiler must be able to throw out the errors associated with that fail if it is to try something else. Also, once the Sytch list items are restored, they are not needed any more and must be disposed of also, but without alteration of information which was on the list before this point was reached.

Sometimes it may happen that a scan will fail, which would normally cause TRY to return answer false, but there has been an indication that what is being scanned really is the construct being tried, though incorrectly formed. For example, one of the things that is tried is logical expression. If logical expression fails, then arithmetic expression is assumed. Suppose that the following string appears:

SL .OR. $M

where the $ was presumably meant to be an S. This is not a legal logical expression, but it is clearly not an arithmetic expression. The presence of the logical operator .OR. strongly indicates that the expression was meant to be logical. In cases like this, the automatic succeed flag is used. When a logical operator is discovered, this flag (which is on the TRY list) is set to true. A similar thing occurs in other TRYs. How this situation is handled when the scan is finished is discussed below.

As mentioned under Error Messages, it is possible to have a delayed fail. This is where the scan proceeds successfully, without going to FAIL, but a fail error (level 3) message has been generated. Thus, when the routine that is being TRYed returns to the TRY routine, tests for this situation are made. If no delayed fail exists, the following is done:

1. Release the save list. Note that FAIL did not do this, because it does not know whether it may have to recover this information again.
2. Unreserve the Sytch and error lists. This is done because the TRY is now just like a JRS, and any errors generated or symbol table changes made are now the responsibility of the routine above the TRY. TRY has done its work; the situation is now just as if it had been a JRS instead of a TRY.
3. Remove the delayed fail and automatic succeed flags from the TRY list.
4. Exit true.

If, on the other hand, a delayed fail does exist, the automatic succeed flag is tested. (As mentioned under FEX-TRY Save, the automatic succeed flag and the delayed fail flag are recursively contained on the try list.)

If the automatic succeed flag is not set, control goes to FAIL, which will eventually end at Try-Fail (below). If, however, the automatic succeed flag is set (meaning, "Yes, it was one of these despite any fail."), the procedure is then the same as if there had been no delayed fail, with one exception. There has been a delayed fail in this scan and, even though it is desired to accept that this was indeed the proper scan (i.e., and not try anything else), it is necessary to remember that there was a delayed fail. So, the delayed fail flag at the current level is set. That is, if this TRY is within another TRY and this one gets a delayed

fail, but with the automatic succeed flag set, this produces a delayed fail for the upper level TRY.

If, for any reason, a routine that is being TRYed goes to FAIL (which may result from a delayed fail, as discussed above), control will pass from FAIL to Try-Fail (as opposed to FEX-Fail). This does the following:

1. Releases the save list and the Sytch list, which FAIL would do except that it cannot when under FEX control.
2. Removes the two flags from the TRY list.
3. If the automatic succeed flag was not set, it releases the error list (thus discarding any errors accumulated; they are not needed since some other scan is going to be tried instead) and goes to Exit False.
4. If, however, the automatic succeed flag is set, the situation is similar to above, where the delayed fail is passed on to the upper level. In this case it is not a delayed fail but an immediate one, so the analagous thing is done, i.e. an immediate fail at this level. This routine has been reached via FAIL and returns immediately to FAIL. Before doing this, the error list is unreserved, rather than released as in a normal fail. The idea here is that, when the automatic succeed flag is set, TRY behaves very much like JRS; the errors it has collected are valid for the upper level and it is appropriate to fail now at that level since an unrecoverable error has been reached (otherwise FAIL would not have been called).

As discussed under Statement Scan, when a fail occurs that is not under a TRY, the usual procedure is to reset everything to the beginning of the statement and try another kind of statement. Whereas the preparation for FAIL is done within the TRY POP itself, for statement scanning it is done in Statement Scan. Both places use FEX-TRY Save. Using SAL, statement scan also saves all the lists which may have information added to them by any statement (essentially all of them) and reserves the save error list.

When a statement is successfully scanned, control passes to Statement Exit. The first thing done here is to check the delayed fail flag. If there is a delayed fail, control goes to FAIL immediately. Otherwise, the following actions are taken:

1. Release the Sytch list. This information does not have to be recovered. It is correct.
2. Throw away any errors that have been saved on the save error list. Since this statement succeeded, its errors (if any) are the valid ones.
3. Release the save list. Note that FAIL may have been repeatedly putting back the information saved here. Finally, now it is no longer needed.
4. Unreserve the error list to combine the present errors with the upper level, if any (e.g., in logical IF).

5. Remove the delayed fail and automatic succeed flags from the Try list. (Remember, these were put there by FEX-TRY Save.)

6. Exit.

When a statement fails, either directly or because of a delayed fail, control passes from FAIL to FEX Fail, rather than TRY-Fail. This routine compares the input character count (indicating how far this statement got) with that of the previous statement which got the furthest to determine whether to treat this input as this type of statement or the other.

If this one is no greater, the error list is emptied and the delayed fail flag is reset to false. (It may have caused a fail on this type of statement, but now that a new kind of statement is being tryed, it no longer has any meaning.) Then control passes to the location specified by the last FEX.

If this statement did proceed further than any previous one, the same actions are taken. In addition, however, the new input character count now becomes the old one, and any errors which may have been saved on the save error list are thown away and replaced with the current errors (if any) from the error list.

If all the FEXed statements fail to successfully scan a statement, control falls to illegal statement. Of course, FAIL has once again restored everything to the way it was at the beginning of the statement, so a new type of statement may be tried, but there are no more. In this case, the same six steps are followed as above under Statement Exit, except that the errors preserved are those on the save error list, i.e., those for the statement which got the furthest. Furthermore, instead of just exiting, a trigger is set which tells release line to print "Statement Deleted" and go right back to FAIL. This situation requires an upper level saving, in order for FAIL to function. It is usually set up in label field scan, but in the case of the statement scanned as part of a logical IF, the upper level will be the logical IF scan itself.

SPECIAL LISTS SET UP IN PASS 1

SYMTABLE — Symbol Table

Entry size: 6 words

Discussed under "Identifiers."

LOCLBLST — Local Label List

Entry size: 2 words

First word contains the numeric representation of the label in the low 17 bits. Upper bits are used for flags similar to those in the third word of the symbol table:

Bit 0 - Defined
1 - Defined on an executable statement
2 - Defined Multiply

Bit 3 - Referenced
4 - Referenced as a FORMAT

These bits are set up and tested with the SUL and SOL POPs respectively.

The seond word is set up by pass 2 as the relocatable address of the label.

NLCLBLST Non-Local Label List

Entry size: 2 words

This list is the same as local label list but for labels which appeared with a $ after them.

ICONSLST Integer Constant List

Entry size: 2 words

First word contains the value of the constant. Second word is set up by pass 2 to indicate whether the constant is ever used. For example, the statement,

J = J + 1

registers a 1 on this list, but the statement is actually generated using an MPO instruction.

RDCONLST Real-Double Constant List

Entry size: 4 words

First three words contain the value of the constant, expressed in double-precision form. All floating-point constants are scanned in double precision, regardless of whether a D exponent follows them, in case they need to be double. For example, in the statement,

PRINT 5, 2.3 + DBL

where DBL is a double-precision variable, the constant looks REAL, but the whole expression is double and must be computed in double precision. Therefore, not only is less code involved not to generate a REAL constant and convert it, but, since 2.3 does not come out even in binary, more accuracy is obtained. Note that, although both real and double-precision constants are stored on the same list, any pointer to such a constant will indicate whether it looked real or double-precision. Thus, a constant specifically written with a Dexponent will force the expression into double-precision. Pass 2 eventually decides which kind of constant to make out of it (if any). Note that, as a result, both a real and a double-precision constant could be produced from the same entry in this list.

The fourth word contains the indicator whether the constant has been used, as with the integer constant list.

CCONSLST Complex Constant List

Entry size: 5 words

First four words contain the REAL values of the real and imaginary parts.
Fifth word contains the "used" indicator.

HCONSLST Hollerith Constant List

Entry size: 2 words

Same as integer constant list except that since these are on a separate list, pass 3 knows they are Hollerith and prints them that way instead of as decimal integers.

BLCOMLST Blank Common List

Entry size: 1 word

Each word is a pointer to a variable in the symbol table. The order corresponds to the order of blank common; that is, the TOP of the list is the first word in blank common, and the BOTTOM of the list is the last.

LBCOMLST Labeled Common List

Entry size: variable

Each entry consists of the following:

1. An integer 1, indicating start of a new entry.
2. A pointer to the block name (on the block name list, see below).
3-N. Pointers to the symbol table. These are the variables to be put into this labeled common block.

Note that one entry does not completely define a block. It may be reopened later, possibly even in the next entry, as in the statement,

COMMON /B/A/B/Y

BLNAMLST Block Name List

Entry size: 3 words

First two words contain the BCD name of the block.
Third word is set up during allocation and equivalence to indicate the size of the block.

Note that block names are the only ones which are not registered in the symbol table. Because they are allowed to conflict with most other names, they are treated independently.

EQUIVLST Equivalence List

Entry size: variable

Each entry corresponds to one equivalence set and contains:

1. The number of the line on which the set began, with a sign bit merged in. This is used to indicate beginning of a new set and also to print meaningful diagnostics in allocation and equivalence when errors are discovered that this equivalence set caused.

2 - N. Symbol table pointers to the variables in the set. Each of these may be followed by any number of integers indicating the "subscripts" which appeared after name in the EQUIVALENCE set. These are not pointers to the integer constant list (which is unusual) but just integers. They are not even registered on the integer constant list.

NAMELST Name List

Entry size: 1 word

Each entry is the relative position in the symbol table of a variable which is to be name listed for use by the INPUT statement. Note that they are not ordinary pointers to the symbol table. They have had their upper nine bits (class and type) stripped off so that, when registering the names on the list, the search will find the name even if it has subsequently changed class or type.

INTRILST Intrinsic List

Entry size: 3 words

First two words contain the BCD name of an intrinsic function recognized by the compiler.
Third word contains the type (integer, real, etc.) of the function and the location, in pass 1, to which to branch to process this function's arguments (each intrinsic function must have the right number and type of arguments). This is an unusual list, in that it is never used, in the ordinary sense. Nothing is ever put on or taken off it. It is just there in the middle of pass 1 and is searched every time a new function name appears.

ALPHALST Alphanumeric List

Entry size: 1 word

This list is used to save alphanumeric characters that must be included in the code. Thus, it is used by the FORMAT and OUTPUT statements, both of

which want to save the active characters which they scan and build a FORMAT out of them. The characters are not packed into this list; they are stored one per word in the character translate table form. There is a subroutine called PACKALPH (Pack Alpha list) which takes these characters off the top of the alpha list and packs them onto the code list preceded by a word count.

IFLIST — If List

Entry size: 1 word

In order to generate more efficient code on arithmetic and device IFs, it is necessary to know what the statement number (if any) of the following statement is. Thus, upon encountering the statement directly after each such IF statement, a pointer to its label is added to the if list. If it has no label, a zero is put on the list. There will be one entry for each IF statement in the program, and pass 2 pulls them off and compares them with the transfer labels on the IF statements as it receives them.

DSUBLIST — DATA Subscript List

Entry size: 3 words

Used only in the DATA statement, to save the names that occur as subscripts or DO-control parameters.
First word contains pointer to name in symbol table.
Second word contains character count at which the name first appeared.
Third word is a trigger indicating whether the variable is under control of a DO.
This allows detection of two errors. Each DO-control index can be checked and each subscript or DO-parameter variable can be checked to assure that it is under control of a DO.

This list is also used later by Expand Data Pair to contain the DO index, the increment, and the count, during DO loop expansion.

INDOLIST — Inner DO List

Entry size: 1 word

Also used only in DATA statements. Each time a DO-control variable appears, this list is searched to make sure that the variable is not already controlling an inner DO. Then a pointer to the name is stored on the list so that outer DOs can search for it.

There are several other lists used only in DATA statements, but they are just ordinary manipulation lists of no particular interest.

DOLIST DO List

Entry size: 2 words

Used during the range of a DO or REPEAT loop.

First word contains the label of the statement on which the loop ends.

Second word contains the number of the line on which the DO or REPEAT statement appeared.

This list is searched after each statement which has a label to determine if there are any loops ending on it. If so, the appropriate End-of-Do indicators are sent to pass 2. The line count is there in case the terminal statement never appears. The error message at the end states on which line the unclosed loop was opened and on which label it should have been closed.

KLUDGLST Kludge List

Entry size: 2 words

First word of each entry contains a pointer to the symbol table location which was destroyed.

Second word contains the old contents of that location.

During the scanning of the expression which defines a statement function, its dummies must be treated as scalars even though their names may have already appeared in the symbol table and are, say, subprograms or array names. Also the name of a FUNCTION subprogram must be used as a scalar during the program but restored as a function at the end. The procedure in both of these cases is to create a new entry in the symbol table, with the same identifier as the old one, and temporarily destroy the old one. This is done by storing all 1's in the first half of the name in the symbol table. When the function is finished, the old name is restored. The new name remains also. This list contains the information necessary to restore the old names.

The following describes the information set up for arrays. There are three lists involved in expressing all of the pertinent statistics about an array: the symbol table, the array list, and the group list.

The symbol table, of course, contains the name of the array and other information (see Identifiers). The fourth word of a symbol table entry for an array contains a pointer to the array list.

There are three important things that must be known about an array that are not in the symbol table. These are its offset, size, and subscript multipliers. For arrays with constant dimensions, all these values are also constants. However, an adjustably dimensioned dummy array may have expressions representing any or all of these quantities. In the discussion below,

anything called an expression may actually be something quite simple, such as a constant or a variable, or it may be more complicated. A "complicated expression" is one which is not directly addressable in INTEGER and must therefore be computed at the beginning of the subprogram and stored in a private temp for use.

ARRAYLST Array list

Entry size: 3, 5, or 6 words

These entries are

1. Offset (in elements) - Pointer to expression
2. Size (in elements) - Pointer to expression
3. Pointer to group list entry containing the multipliers (see below).
4. On dummy arrays only, this is a pointer to a group list entry containing pointers to the complicated subscript multipliers and the temps into which they are to be stored (see below).
5. On dummy arrays only, this is a pointer to a group list entry which is a copy of the plex list used to represent the expressions.
6. On dummy arrays only, and only if the offset is not an integer constant, this is a pointer to a private temp into which the offset (in words) is to be stored.

GROUPLST Group List

Entry size: variable

The group list can be thought of as the resident plex list. It is used by many routines to hold information that must not be sent along to pass 2 after each statement. One of its primary uses is to contain array information, as indicated above under the ARRAYLST.

Each "group" on the group list is formed using the COG (Copy Group) POP, and can be pulled back using the PUG (Pull Group) POP. The first word of such a group is always the word count of the entire group. There are three important groups that may appear relevant to arrays:

1. All arrays have a group containing their multipliers. Although, for an N-dimensional array there are theoretically only N-1 multipliers, N multipliers actually exist, the extra one being the number of words per element, which can really be treated as another dimension (since different type and SDS/ASA storage allocation affect this number). Thus, the first word in this group is the number of words per element and the next N-1 are the subscript multipliers. If these are complicated, this will contain pointers to the temps into which they will be stored. The actual expressions representing the multipliers will be found in the next group.

2. On dummy arrays, this group contains 2-word entries, one for each complicated multiplier. The first word is a pointer to the expression used to compute the multiplier, and the second word is a pointer to the private temp into which this multiplier should be stored. These temps are then pointed to by the group discussed above. If there are no complicated multipliers, but the array is nonetheless a dummy array, this group still exists but is empty.

3. Again on dummy arrays only, this is a group containing a complete copy of the plex list as it stood after computing the various parameters of the array. All the pointers to expressions mentioned above actually point to the plex list. However, since these are not sent to pass 2, as such, and the compiler must be able to get at them whenever necessary (e.g., every time subscripting on the array is computed), the whole plex list is saved on the group list.

IN-LINE SYMBOLIC CODE

When pass 1 encounters a source line with an S in column 1, the S trigger is tested to see if the symbolic code processor was requested. If the symbolic code option was not requested, an illegal syntax message is printed and the statement is deleted.

The symbolic code processor consists of two parts:

1. OPDSCAN - tests card for OPD pseudo-operation
2. SINCOLM1 - processes OP code and Operand fields

The OPDSCAN does a "look ahead" on the card to see if the operation code is an OPD pseudo-operation. If it is, the first nonblank character string in columns 2 through 5 becomes the mnemonic for the new numeric definition. If the new mnemonic is identical to another mnemonic which has been entered as an OPD, the later one is discarded. If the new mnemonic coincides with one of the basic mnemonics, the new definition overrides the old. Any characters (except blank, which is the field separator for an operationcode) are allowed in the label field for an OPD, and subsequently as legitimate operation code entries. If an OPD (operation code) is found by OPDSCAN or a nonblank in column 6, OPDSCAN returns the answer true to pass 1; otherwise, it returns the answer false.

Pass 1 first tries OPDSCAN. If the answer true is returned, the statement processing is complete and pass 1 does the final cleanup after the normal end of statement is reached. If the answer false is returned, pass 1 processes the label field and during statement scan branches to SINCOLM1, the operation code-operand processor.

SINCOLM1 looks in columns 7 through 14 for a nonblank character; if none is found, a NOP is assumed and the processing is assumed completed. When a nonblank is found, the QUOTE is tested to see if it is SHIFT; if so, the Shift Operation Code is entered and processing goes to the operand field processor after the first blank is found. If the SHIFT quote is not found, the operation code is built from the next (at most 4) nonblank characters, with trailing blanks inserted if fewer than four consecutive nonblank characters are found. The OPDLIST is searched and then the OPCODLST. If the mnemonic is found, the operation code is entered from the list and the operand field processor is entered after the first blank is encountered. (The OPDLIST contains the mnemonics and new operation codes for the OPD's used by the programmer to this point. The OPCODLST contains the list of basic mnemonics and corresponding octal operation codes).

The operand field processor allows only specific formats in the address and tag fields. The tag field is a number from 0 through 3 only, and a diagnostic is produced if the value is outside that range.

The address field may be one of the following:

1. An octal or integer constant. A constant is octal if it begins with a leading zero; otherwise, it is decimal. Constants exceeding five octal digits are truncated mod 32,768 and cited as errors.
2. A literal. A literal may be any of the following preceded by an equal sign:
   a. Any INTEGER, REAL, or DOUBLE PRECISION FORTRAN constant except an octal or Hollerith constant.
   b. An octal constant, as defined in paragraph 1 above. (Note that an octal literal has a leading zero but no trailing B.)
   c. A character string of not more than four characters enclosed in quotation marks.
3. A relocatable address plus or minus an octal or integer constant. A relocatable address may be any of the following:
   a. $ indicating current location counter.
   b. DDDDS indicating local label DDDD.
   c. DDDD$ indicating non-local label DDDD$.
   d. Any FORTRAN identifier other than those being used to identify an intrinsic function.

After the operand field has been processed, the operation code, addend, and address field are passed on to pass 2 after a special identifier on the code list.

Warning messages are printed if a relocatable address is used where it is not normally accepted.

SINCOLM1 returns to pass 1 at SCRDEXIT.

PASS 1A

ALLOCATION AND EQUIVALENCE

Between the syntax analysis phase (pass 1) and the code generation phase (pass 2) of the FORTRAN IV compiler is the allocation and equivalence phase (pass 1A).

Pass 1A must make an initializing sweep through the symbol table and pass 1 must make a finalizing sweep through the symbol table. These are combined into one sweep in pass 1. The initializing done for pass 1A is identifying global symbols as global and the remaining symbols as allocatable.

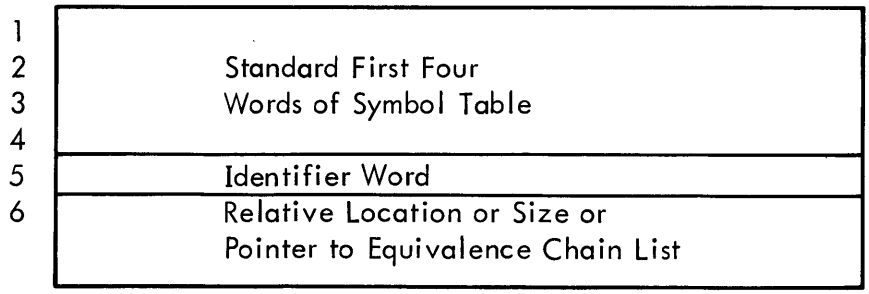
After the initalizing is completed, pass 1A processes the blank Common entries, then the labeled Common entries, identifying each in the symbol table and setting the specified relative location to the Common base.

Once all the direct external allocations have been made, the equivalence list is processed and all indirect external allocations are made and conflicts in allocation or type are listed. Also a tree structure is built up for the allocation of equivalenced local variables.

When the equivalence list is completely processed, a final sweep is made through the symbol table allocating all local variables relative to the base of local storage. If an element of an equivalence tree is encountered, the whole tree is allocated at that time.

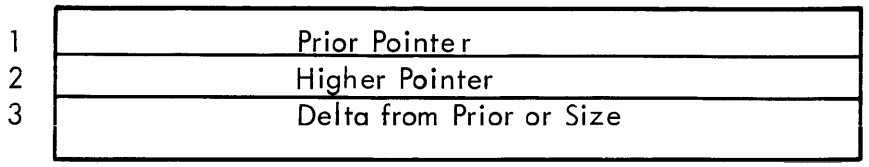
Upon completion of this final sweep through the symbol table, the complete variable storage requirements for the program are known.

Symbol Table - Pass 1A



Block Name List

Equivalence Chain List - Pass 1A



SYMBOL TABLE USE IN PASS 1A

During the initialization of the symbol table:

1. For global variables the identifier word is set to the value of GSYMFLAG(3) and the size (number of words required) is entered in word 6.
2. For all other variables the identifier word and relative location word are set to zero.

During blank Common allocation, the blank Common flag (1) is put into the identifier word and the relative location of the symbol in blank Common is entered in the relative location word.

During labeled Common allocation, the pointer to the block name list is put into the identifier word and the relative location of the symbol in the labeled Common block is entered in the relative location word.

During equivalence processing, a tree structure is used to keep track of equivalences among local variables. If any equivalences link a free element (an element not external or in a tree) to an external variable, that element is not entered into a tree, but is allocated (in the relative sense) immediately. (NOTE: allocation due to equivalence to an external is done only if there is no conflict due to extension and hereafter means that.) If an element of a tree is in equivalence to an external, (See Structure of Equivalence Trees) the whole tree is allocated immediately and the tree structure is forgotten (the pointers are dropped) but remains on the equivalence chain list.

When an element is in equivalence to a global directly, a pointer to the global is entered as the identifier and the relative location to the base of the global is entered in the relative location word.

When an element is in equivalence to a global indirectly (i.e., in equivalence to a symbol which is in equivalence to a global), the pointer to the global is entered as the identifier and the relative location to the base of the global is entered in the relative location word.

An element in equivalence to an element of Common (blank or labeled) is identified as if it had appeared in the original common list.

In making the final sweep through the symbol table:

1. The blank Common and global identifiers are modified for pass 3.
2. All equivalence trees are allocated.
3. Labeled Common and elements in equivalence to global are unchanged.
4. All unallocated variables (arrays and scalars) are allocated except for dummies.
5. External subprograms are flagged in the relative location word with 070000000.

The above are done in order of appearance in the symbol table.

USAGE OF WORDS 5 AND 6 IN SYMBOL TABLE

Word 5 is a pointer or an integer.

Word 6 is a pointer or a word of form 9, 15.

Usage During Equivalence

Type	Word 5	Word 6
Global Symbol	00000003	0, size
Blank Common	00000001	0, relative location
Labeled Common	Pointer to block name list	0, relative location
Equivalenced to Global	Pointer to symbol table	0, relative location
Equivalenced to another local symbol	00000002	Pointer to equivalence chain list
Not equivalenced or allocated yet	0	0

Output to pass 2, 3

Type	Word 5	Word 6
Global Symbol	00000000	3, size
Local Scalar	00000000	5, relative location
Local Array	0	4, relative location
Blank Common	0	1, relative location
Labeled Common	Pointer to block name list	0, relative location
Equivalenced to a Global Symbol	Pointer to symbol table	0, relative location
External Subprogram	0	7, 0
Not allocated	0	0

STRUCTURE OF EQUIVALENCE TREES

An equivalence tree is composed of three types of elements:

1. The lowest (first) member
2. The intermediate member
3. The highest (last) member

There is always one type 1 and one type 3 element. There can be varying numbers of type 2 elements, depending on the size of a particular tree.

The intermediate member is distinguished by a symbol table pointer (in Prior Pointer) to an immediately prior (allocation wise) member of the tree and a symbol table pointer (in Higher Pointer) to an immediately following member of the tree, and a delta (which may be zero, but never negative) that specifies the number of memory locations from the prior member to this member.

The highest member of the chain is distinguishable from an intermediate member because the higher pointer is zero.

The lowest member of the chain is distinguishable from any other member due to the fact that it has no symbol table pointer, but zero, in its prior pointer. Also the total size of the chain appears in the delta word for this element, since a delta is meaningless.

USE OF EQUIVALENCE TREES

When a free element is in equivalence to an element of a tree, it is 'fitted' into the tree. If the free element belongs higher in the tree, the next higher element of the tree is compared to see if it goes between the two. If so, it is entered into the tree with a delta and pointers, and the corresponding pointers and delta are corrected to place this new element in the tree. Then the size of the tree is increased if necessary. If the free element belongs lower in the tree, the next lower element of the tree is inspected and, if the new element is higher, it is entered between the two; otherwise, the process is repeated, going down the tree. If the free element belongs below the lowest, the free element becomes the new lowest, and the old lowest is modified to reflect the change. In either case, the total size is modified to reflect any necessary changes to tree size.

PASS 2

OVERALL FLOW

At the highest level, pass 2 is quite simple. It begins by initializing a few things--zeroing the location counter, temp counters, etc. It generates a BRM 9INITIAL if this is a main program. Then it generates code for statements, one by one as they come, until it encounters an END statement, at which time it generates array pointer constants, label constants, and NAMELIST (if necessary), and calls pass 3.

Generation of each statement is handled by a subroutine called Next Statement Gen. It reads one word from the input string by calling Next Input Item. The word it reads is a simple integer indicating what kind of statement this is, and this integer is used to index a jump through Statement Gen BRU Table to take it to the appropriate statement routine. Next input item tries to take off the top of the input list. If it can, it returns the word it got on the bottom of the work list. If the input list is empty, next input item reads the next record from the scratch tape, appends it to the input list and then takes off the top and returns. Most of the statement generating routines also use next input item to get whatever words are needed to specify the statement.

The statement generating routines leave the code they have generated on the code list and exit through statement exit, thus effectively exiting from next statement gen. The code on the code list must be assembled and put onto the output list. Assembling the code amounts to keeping a location counter and recording the location at which each label is defined. Thus the Assemble Code routine is rather like the first pass of a conventional symbolic assembler. Definitions of labels are recorded in the local label list, the nonlocal label list, and the created label list. Definitions of subprogram names and dummies are recorded in the symbol table. For created labels assemble code also creates a numeric value for the label (to be used on the object listing--e.g., 23G). Thus, the numbers reflect the order of definition of the labels, and 23G will never represent a higher location than 24G.

It is also the responsibility of assemble code to keep track of which constants on the various constant list are actually used. There are a fair number of reasons why a constant may be registered on one of the constant lists but never be used by the object program, and it is unnecessary to create literals for the unused ones. So each item on the constant lists has a special word in which assemble code indicates whether they were used or not. On the real-double constant list, two bits are used--one to indicate if the constant were used as a real constant, the other to indicate if it were used as a double-precision constant. The literal table is generated by pass 3 and is built so that constants may overlap. For example, if the floating-point constant -1.0 (40000000 00000000) and the integer constants -8388608 (40000000) and 0 (00000000) are needed, all three constants will come from the same two words.

As each word of code from the code list is assembled, it is put onto the output list, from which it may automatically go to a scratch tape if memory overflows. At the end of each statement, the output-to-tape trigger is tested to see if memory has already overflowed; and,

if it has, the output is written on tape immediately, without waiting for memory to overflow again. This test is made at the exit from assemble code, which is called output code.

The reason generated code is assembled a statement at a time instead of a word at a time is that many of the statements generate code out of order and re-order it. The most obvious example of this is the DO statement which generates the top and bottom of the loop at the same time, then saves the bottom part to be put out later when the DO ends. In calling sequences, the code to calculate argument addresses is generated at the same time as the PZEs which transmit the addresses, and PZEs are delayed to come out where they belong. The standard way of delaying code is to reserve the code list before generating the code, then copy the bottom file over to the delayed code list. When the delayed code is to be put out, the bottom file of the delayed code list is copied back to the code list.

Some of the things which pass 1 sends as "statements" are not really statements. For example, there is the "Load Plex List" statement which means, "here are some things for the plex list which will be used by the next statement." The next word is a count indicating how many words are involved, followed by the words themselves. Then there is the "Compressed Source Line Statement" indicating that the next n words are to be sent to pass 3 for listing as a source line on the object listing. These lines are not sent unless the LO control card option has been specified. Corresponding to DO and REPEAT FOR statements, there is the "End of Loop" statement which means "end the most recently started loop." It simply causes the last file on the delayed code list to come out. Clearly, End of Loop is not adequate for improperly nested loops, so there is an "End of Illegally Nested Loop" statement to indicate which loop should be ended. This takes a certain amount of manipulating to pull the appropriate file out of the middle of the delayed code list. Individual items in an input/output list are sent as if they were complete statements in themselves.

The assignment statement is generated by Assignment Statement Gen. Aside from the word indicating that this was an assignment statement, there is only one other word which appears in the input string: a pointer to a Replacement Plex. Like many pointers involved with statements, however, the assignment has subscripts involved with it, so it may in fact be a subscripted expression plex connecting a script group plex and the replacement plex. Therefore, the assignment statement does not use next input item to get the next word but uses next item script gen which gets the next input item and checks to see if it is a subscripted expression plex. If it is, code is generated to evaluate all the subscripts and store their values in temps, and the pointers to the temps are left on the script list.

Two statements--arithmetic IF and DATA--receive information other than that in the input stream. For the arithmetic IF statement there is the IF list, which contains the labels of the statements following arithmetic IFs. These labels enable pass 2 to sense when certain branches of an IF can "fall into" the next statement. Thus, the arithmetic IF generator is concerned with four labels--the three written in the statement itself and the one (if any) attached to the following statement. It tests the four for equality in various combinations to eliminate redundant testing and branching instructions.

For DATA there is the data pair size list, indicating how many words there are in a particular data pair. (A data pair is the combination of a variable list and a constant list; a DATA statement may contain several data pairs.) The reason for this list is to allow pass 1 to discard a data pair retroactively. In general, pass 1 cannot hold all the information generated for one data pair in core at one time--there may be thousands of words of it. At the same time pass 1 is deriving these thousands of words, it is checking subscripts and constant types for errors, and if it finds an error it must discard the whole data pair. Unfortunately, by the time it has found the error, thousands of words may have gone onto tape. Therefore, it keeps track of how many words it has been putting out; and, if the data pair errs, pass 1 enters the word count onto the data pair size list, thus telling pass 2 how many words to ignore. If the data pair is all right, pass 1 enters zero on the data pair size list, telling pass 2 not to ignore any of this data pair.

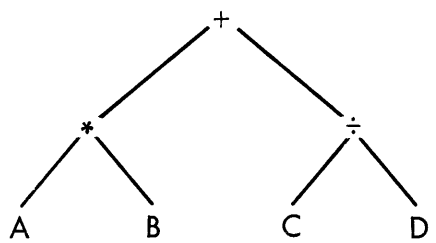
The logical IF statement is rather unusual, too, in that it controls another statement or group of statements. (The group of statements can come when a logical IF controls a compound statement.) The compiler needs to know if the statement the logical IF controls is a GO TO, since knowing this fact it can generate better code. So the compiler reads ahead into the next statement to see what it can be. If the next statement is not a GO TO, code is generated which jumps if the logical expression is .FALSE. - jumps to a created label which will be defined later. Then the compiler generates code for the statement or statements controlled by the logical IF, sensing the end of them by an end-logical-IF-statement indicator. The compiler cannot simply go to next statement gen to generate the next statement, since it has already called next input item and read the first word of the next statement. Therefore, it calls statement gen from W0, which assumes that the first word has already been read. After logical IF has caused all the statements controlled by it to be generated, it defines the label to which the jump was made.

If the statement controlled by the logical IF was a GO TO, code is generated which jumps if the logical expression is .TRUE. - jumps to the label the GO TO was going to.

INTERNAL REPRESENTATION OF EXPRESSIONS

Pass 1 translates statements and expressions into a form that is more convenient than source form for manipulation and code generation. The internal representation of expressions is a tree-structure form indicating what operands are connected by what operators. The representation is rather like Polish notation, but it is somewhat less order-dependent than Polish.

The basic unit of representation is the plex. A plex is a collection consisting of an operator and all of its operands. For example, the expression A*B + C/D is represented by the following tree structure:



The expression consists of a sum plex whose two operands are a product plex and a quotient plex. The operands of the product plex are the two scalars A and B, while those of the quotient plex are the scalars C and D.

When expression scan scans the above expression, it leaves a pointer to the sum plex on the bottom of the work list. The rest of the information about the expression is contained on the plex list. The pointer on the work list 40200011. This pointer indicates that it points to a sum plex (40), that it is real (2), and that the plex is in relative location 00011 on the plex list. The information on the plex list is as follows:

relative location	contents	
00001	00000004	4 words in this plex (2 operands)
00002	41200000	Product (41), Real (2)
00003	71200001	Scalar (71), Real (2), location 1 in SYMTABLE
00004	71200007	Scalar (this one is B)
00005	00000004	4 words in this plex (2 operands)
00006	42200000	Quotient (42), Real (2)
00007	71200015	Scalar (this one is C)
00010	71200023	Scalar (this one is D)
→ 00011	00000004	4 words in this plex (2 operands)
00012	40200000	Sum (40), Real (2)
00013	41200001	Product plex, location 1 in plex list
00014	42200005	Quotient plex, location 5 in plex list

└— Pointer on work list points here.

The expression A + B + C + D is represented as one sum plex containing four operands. The code generator can use the four operands in any order it chooses. However, if the expression were written (A + B) + (C + D), it would be represented as one sum plex combining two other sum plexes. This informs the code generator that it may not combine A with C or B with D, as it would have been able to do without the parentheses; nor may it add A to B, then add C to the result, then add D to the result; it must combine C and D with each other before it combines them with anything else. In this way, the integrity of the user's parenthesis groupings is preserved.

The expression A - B involves two plexes. It is treated as if it were written A + (-B) and consists of a sum plex combining the scalar A with a minus plex that has a single operand – the scalar B.

In summary, there are five kinds of plexes involved in a basic arithmetic expression: the sum plex and product plex, containing a variable number of operands (but always at least two); the minus plex involving one operand; and, the quotient plex (/) and expon plex (**) involving two operands.

In the previous examples, the only operands have been scalar variables, but there are other kinds of operands than scalar variables. Constants are represented as pointers to the appropriate constant list (there are four: integer constant list, Hollerith constant list, real/double constant list, and complex constant list). Two other operands are array elements and function calls.

A function call is represented with a function call plex which contains two operands: a symbol table pointer to the function name and a pointer to a subprogram argument (SPROG ARG) group plex. The latter is a variable-sized plex and may contain as few as 0 arguments or as many as desired. Each argument is an expression pointer of some kind, a subscripted array plex of some kind, an array name pointer (unsubscripted), a statement label pointer, or a subprogram name pointer.

There are two kinds of plexes involved in array elements: the constant subscripted array plex and the fully subscripted array plex. A constant subscripted array plex is formed whenever the effective subscript (produced from combining all the subscript expressions) is a constant. This means that all the subscript expressions must be constant (though for subscripting purposes, 3 + 5 - 2 is considered a constant), and the dimensions must be constant; i.e., constant subscripts are not enough if the dimensions are adjustable. The constant subscripted array plex contains two operands: a symbol table pointer to the array name, and a pointer to the integer constant list.

A fully subscripted array plex contains three operands: a symbol table pointer to the array name, a pointer to the integer constant list, and a pointer to the non-constant part of the subscript. The constant is an addend which has been "factored" out of the subscript expression . If the non-constant part of the subscript is non-addressable, the pointer will be to the script list, where the pointer to the actual expression will be. The script list contains all the non-addressable non-constant parts of all subscripts used in a particular construct, which is usually a statement, but sometimes is smaller than a statement.

The plexes involved in logical expressions are quite analogous to the others. The .AND. plex, .OR. plex, and .EOR. plex are variable-sized like sum and product. The .NOT. plex is unary like minus, and the .NE., .LE., .GT., .EQ., .GE., and .LT. plexes are binary. The extended relational plex (for constructs like A .LT. B .LT. C) is variable-sized, but always contains an odd number of operands. The first, third, fifth, etc. operands are true operands, while the second, fourth, etc. are operators. The operators are the addresses of the plex constants that would have been used if the relational were not extended.

There are two plexes involved in the way pass 1 tells pass 2 about subscripts. The script group plex is used to tie all the subscript expressions on the script list together into a bundle, and the subscripted expression plex is used to tie the bundle to the expression it goes with. These plexes are created only if there are some subscripts. For the most part, there is only one subscript bundle per statement, but in some statements it is important that the subscripts be evaluated at the right place. Individual I/O list items have their own subscript bundles (it is legal to input J, then A(J), such that the subscript for A(J) could not properly be evaluated at the beginning of the statement). The individual items in a REPEAT FOR list have their own subscript bundles, as do the assigned labels involved in computed GO TO

statements and arithmetic IF statements.

At the end of each statement, pass 1 outputs the plex list for pass 2. It does not let the amount of this information grow over the whole program.

The DATA statement is an exception to most rules, and plexes are no exception. In scanning the DATA statement, pass 1 may build several plexes, the way it would for most statements. Then, instead of sending these to pass 2, it unravels the plexes itself. The reason for doing this is that there are a large number of errors the user can make in writing a DATA statement, and some of them are very inconvenient to detect while scanning. Problems like constants of the wrong type or subscripts out of range (especially when the subscripts are under DO control) cannot be detected without actually "doing" the DATA statement. This is what pass 1 does, and when it gets done the output it sends pass 2 is quite simple, though possibly quite verbose. It consists of little substatements of the form: "put this constant into that variable" – one of these for each replacement implied by the DATA statement.

PLEX-BUILDING POPS

There are four plex-building POPs. Two of them – CIC and CIF – build variable-sized plexes, and the other two – FIP and FIC – build fixed-sized ones. Variable-sized plexes, such as sum plex, product plex, .AND. plex, are built from lists which have typically been reserved, and the plexes are built from the bottom file of such lists. One writes CIC SUM LIST to build a sum plex from the bottom file of the sum list. With variable-sized plexes it is necessary to specify what kind of plex to build as well as where to build it from. To specify the former, one writes PLO (plex open) as in

```
PLO     SUM PLEX
CIC     SUM LIST
```

Having built the kind of plex they were told to, CIC and CIF append a pointer to it onto the bottom of the work list. The only difference between CIC and CIF is that CIC inherits traits from all the terms in the plex, whereas CIF inherits only from the first (top-most) term. CIC and CIF empty the bottom file but do not release the list.

FIC and FIP build their plexes from the bottom few items on the work list. It is inherent in each fixed-sized plex how many terms it involves (from minus plex with one all the way up to DATA DO control plex with six), and that many terms are taken from the bottom of the work list and built into a plex. The pointer to the plex is appended to the bottom of the work list after the other terms have been removed. It takes only one POP to build a fixed-sized plex, e.g.,

```
FIP     QUOTIENT PLEX
```

The only difference between FIP and FIC is that FIP inherits traits from all the terms in the plex, while FIC inherits only from the first (top-most) term. It can be seen that for building a 1-word plex like minus plex, FIP and FIC will have the same effect.

It should be noted that the inheritance of traits is not important in some plexes. Certain plexes, such as the ones used in the DATA statement, are used only to indicate structure. Similarly the script group plex, the SPROG ARG group plex, the dummy group plex, the subscripted expression plex, the repeat triple plex only indicate structure. The main use of traits is in plexes used in expressions.

TRAITS

Pointers have traits, and the SOF (Set On Flag) POP tests them. SOF finds a pointer in W0 and tells whether or not is has a particular trait.

Traits are characteristics which generally overlay several different classes of other characteristics. For example, scalar is not a trait, since only a scalar can be a scalar (testable with the SOT POP), but dummy is a trait, since there can be dummy scalars, arrays, and subprograms. Constant is a trait, overlaying integer constant, real constant, Hollerith constant, etc., and subscripted is a trait that is common to fully subscripted array plex and constant subscripted array plex, which in turn may involve arrays or multiple dummies. Plexes can have traits, too, and keeping track of traits constitutes most of the work of the plex-building POPs.

The SOF POP uses a subroutine called Fetch Flag Word, which is also shared by the plex-building POPs. Fetch flag word may find the traits in any of three places, depending on the kind of pointer involved. For a symbol table pointer, the traits are found in the third word (ID word) of the symbol table item pointed to. For a normal list pointer, the traits are found in a special table called list flags which is indexed by the list number. Most lists do not have any traits; it is mainly the constant lists which do.

The data subscript list (used in the DATA statement) has addressable and signed addressable traits. TEMP and PTMP pointers do not actually point to a list, but they look just like list pointers, and there are entries in the list flags table for them, corresponding to the lists they would point to if they were list pointers. Then there are plex pointers. For a plex pointer, the traits are found in the second word of the plex pointed to.

Traits get into plexes in two ways: they are built in or they are inherited. For example, a fully subscripted array plex has the subscripted trait built right in, but can inherit the dummy trait; i.e., this plex is always subscripted, but is dummy only if the array is dummy.

Some plexes inherit traits from all the constituents of the plex, while others inherit only from the first constituent. For example, the dummy trait in the fully subscripted array plex, mentioned above, is inherited from the array name, not from the subscript. Similarly, the traits of a replacement plex are inherited from the left-most variable being replaced, whereas the traits in a sum plex are inherited from all the terms of the sum. Where the inherited traits come from is determined by what POP is used to build the plex. FIC and CIF inherit from the first term only, while FIP and CIC inherit from all the terms.

Plexes are defined with 2-word constants. Such a constant is addressed with a PLO POP if the plex is being built with CIC or CIF and by a FIC or FIP POP if the plex is being built with FIC or FIP. The right 15 bits of each word in the plex constant represent the traits. The first word contains 1's for all the traits that are built into the plex, and the second word contains 1's for all the traits which can be inherited.

Bits 6 through 8 in both words are the mode fields of the plex constant. The first word contains the mode of the plex (1=integer, 2=real, 3=double, 4=complex, 5=logical) if it has a built-in mode; the second word contains all 0's or all 1's in the mode field, depending on whether mode is constant or inherited. When mode is inherited from the constituents of a plex, it is taken as the highest mode found. This establishes the mode hierarchy, such that when a real element is combined with an integer element the result is real.

Bits 0 through 5 of the first word are the ID field of the plex constant. They go into the plex itself, as well as into the plex pointer, to tell what kind of plex this is. Bits 0 through 5 of the second word are unused if the plex is built with CIC or CIF, and contain a word count if the plex is built with FIC or FIP. For a minus plex, the word count is 1, denoting a 1-word plex; for a quotient plex it is 2; and, for a DATA DO control plex the word count is 6.

When traits are inherited from the constituents of a plex, they are usually ORed together; e.g., if either operand of a logical .AND. is to be evaluated in the A register (as indicated by the A REG flag), the result of the whole .AND. will be in the A register. Two traits are not inherited by ORing: the complex mixture flag and double flag (both concerned with complex mixture arithmetic).

The complex mixture flag is derived from the mode fields of the constituents of a plex, not from the traits. It is set if at least one of the constituents is complex and at least one is non-complex. It has nothing to do with whether any subplexes contain complex mixtures, but only with whether this very plex is a mixture.

The double flag indicates whether there are any double-precision elements anywhere in the expression (aside from subscripts and function arguments). It is derived by ORing the double traits of the constituents and ORing the fact whether any of the constituents specify double in the mode field. It would seem reasonable that anything which says double in the mode field would also have the double flag set. In fact, this is true in all cases except one – pointers to the real/double constant list. The mode field of such pointers indicates whether the constant is thought to be real or double by the scan. However, there is only one list flags word for the real/double constant list, and it cannot both have the double trait and not have it; so, it does not have it. Hence, here is a double pointer without the double flag set.

When a plex-building POP has a plex ready to go onto the plex list, it searches the plex list to see if there are any other plexes just like this one (all words must agree). If so, it does not put the new plex onto the plex list but creates a pointer to the old one instead. This occasionally saves space on the plex list, but its main purpose is to facilitate checking for common sub-expressions. It insures that, if two expressions are identical, the pointers to them will also be identical.

Understand that this sub-expression registration is done on a statement by statement basis, not over the whole program. The plex list is emptied and sent to pass 2 at the end of every statement.

ARITHMETIC EXPRESSION GENERATOR

The two most important traits to pass 2 are the addressable flag and the signed addressable flag. The addressable flag indicates that a thing is in memory and can be addressed by an ordinary machine instruction. Thus, scalar variables are addressable, as are constants and array elements. X + Y is not addressable, since it must be computed before it can be used. The signed addressable flag indicates that something is either addressable or minus something which is addressable. Thus, A is both addressable and signed addressable, whereas -A is only signed addressable.

Pass 2 usually starts a calculation with something which is not addressable. For example,

W + X + Y*Z

is the sum of three terms of which only Y*Z is not addressable. Therefore, pass 2 would start the calculation with Y*Z, no matter what order the three terms had been written in. To begin the calculation by adding W to X would require a store into a temp which is not needed. If there are several non-addressable terms in the sum, they are usually done in right-to-left order, so that the calculation of

W*X - Y*Z

can be done without a negation.

In most cases, pass 2 is more concerned with signed addressability than pure addressability. It is willing to compute expressions with the wrong sign and fix the signs later. Quite often the signs will not have to be fixed later, since wrong signs can cancel. For example,

(U - V*W) * (X - Y*Z)

comes out

```
LDP     V
FLM     W
FLS     U
STD     1TEMP
LDP     Y
FLM     Z
FLS     X
FLM     1TEMP
```

in which the two factors are both computed with the wrong sign, and the wrong signs cancel. An expression involving +, -, *, and / can always be evaluated with at most one negation.

The strategy for generating code for division is quite simple: compute the denominator first, if it needs computing. The operation of computing something if it needs computing and

storing it in a temp comes up quite often, so that there are several subroutines to do it in pass 2. All such subroutines have GRNTEE ADDRESSABLE in their names. To guarantee something addressable means: if it is already addressable, do nothing; otherwise, evaluate it and store it in a temp. There are many variations on this. The one used for the denominator is called GRNTEE Signed Addressable by Mode. This means: allow the sign to come out wrong if necessary, and if the denominator is complicated, evaluate each of the parts of it in the mode of the outer expression. This means that the modes of the individual elements of the expression are promoted independently; i.e., the expression

X / (-J-K)

is done as follows:

```
LDA     K
LDB     =23
FLA     =0.0
STD     1TEMP
LDA     J
LDB     =23
FLA     =0.0
FLA     1TEMP
STD     3TEMP
LDP     X
FLD     3TEMP
FLM     =-1.0
```

Here J and K were independently floated, then added and stored in a temp. The sign of the temp was allowed to come out wrong, and was fixed at the end of the expression with a negate.

Other variations on guarantee addressable include evaluation of the expression in its own mode or some specified mode instead of the mode of the outer expression, and not allowing the sign of the result to come out wrong. The right-hand operand of the ** operator, for example, must have the correct sign.

There are about fifty subroutines in pass 2 which are concerned with generating code for expressions. The fifty can be thought of as all entrances to the same routine, and are minor variations on each other.

First of all, there are different entrances for getting the result into a register, memory, or both. The routines which get the result into memory all have GRNTEE addressable in their names. If the thing is already in memory (e.g., X), these routines do nothing. Otherwise, they cause it to be evaluated and stored in a temp. One place where the GRNTEE addressable entrances are used is in generating code to evaluate the denominator for division. They are also used for subprogram arguments.

In a few cases it is desirable to get a result in memory and the accumulator. For example, if a thing is to be squared (appeared on the left of **2), it is needed both places. The subroutines which do this have GRNTEE ADDRESSABLE AND IN AC or GRNTEE BOTH PLACES in their names.

The routines which get the result in the accumulator (the usual case) have GRNTEE IN AC in their names.

Throughout expression generation, the compiler is willing to generate things with the wrong sign, keeping track of whether the current sign is right or wrong. Each of the above three categories has an entrance which is willing to get the sign wrong and an entrance which insists on the right sign. The latter group will generate a negate if necessary to correct the sign. The entrances which allow the wrong sign all have SIGNED or BEST SIGN in their names. These routines all leave the sign of the result on the sign list - 1's if sign wrong, 0 if it is right. Signs get onto the sign list through a subroutine called Is Term Signed Addressable. (Signed Addressable means that a term is addressable except for a possible wrong sign.) In addition to answering the question, this subroutine appends the sign of the term to the bottom of the sign list. The sign is derived from how many minus plexes there were attached to the term. Is term signed addressable removes the minus plexes, if any. Other parts of the generator update the latest sign - usually by exclusive ORing the bottom two entries on the sign list.

In addition to the six entries discussed so far - signed and unsigned multiplied by accumulator, memory, or both, there is the consideration of mode, which puts in another factor of nine. The general rule for mixed mode expressions is that all arithmetic is done in the highest mode of any element in the expression. The usual way of handling this is to set up the highest mode of the expression on the mode list and see that all of the other elements are converted to that mode. The mode list is used as a push-down list. The mode of function arguments is set up independently, so as not to disturb the mode of the expression containing the function call.

The entrances with BY MODE in their names assume that the desired mode has already been set up on the mode list and that all elements of the expression should be converted to that mode. These entrances are used mainly from within the expression generator. For example, for the denominator in division, the full name of the entrance used is GRNTEE SIGNED ADDRESSABLE BY MODE, meaning: get the result in memory, allow the wrong sign, and convert the modes of the elements individually to the mode on the mode list.

The entrances with OWN MODE in their names mean: set up the highest mode of the expression onto the mode list, generate the expression in that mode, and then remove the mode from the mode list. These entrances are used for subprogram arguments and similar items.

The entrances labeled REMEMBER MODE are just like the own mode entrances except that they do not remove the mode from the mode list at the end. The entrances labeled KNOWN MODE cause an expression to be evaluated in its own mode, then converted to the mode on the mode list. This is quite similar to the by mode entrances in that the mode becomes whatever the mode list indicates. The difference is that in by mode the conversion is performed individually for each of the elements, whereas in known mode the conversion is performed only once, at the end. The known mode entrances are quite popular for evaluating arguments to intrinsic functions. Entrances marked FORGET MODE are just like known mode except that they remove the mode from the mode list at the end.

There are several entrances which specify a particular mode, such as GRNTEE REAL SIGNED ADDRESSABLE, GRNTEE DOUBLE IN AC, etc. These are used mainly for arguments to intrinsic functions. They work like known mode in that they evaluate the expression in its own mode, then convert to the mode specified in the name.

If an expression is not addressable, all the generating subroutines will eventually work their way down to a routine called Gen by Plex Type. This routine uses the number in the left-most six bits of the plex pointer in W0 to index a jump table leading to the appropriate generating routines. There is a sum generating routine, a product generating routine, a quotient generating routine, a function call generating routine, etc. Each of the specific generating routines works in the by mode mode – i.e., it causes each element of the expression or subexpression to be converted to the mode on the mode list. They also work in the signed mode, meaning that each one assumes that there is a sign on the bottom of the sign list indicating whether the sign of this particular term is correct. If a generating routine wants to indicate that it has generated the wrong sign, it merely changes the sign on the sign list. This means that if the sign was wrong to begin with, and the routine generated it with the wrong sign, the sign comes out right.

Short Forms of Names for Entrances to the Arithmetic Expression Generator

GrnTee { ACcumulator / ADdressable / BOth the above } { Unsigned / Signed } { BYM By Mode; OWN Own Mode; KNW Known Mode; REM Remember Mode; FGT Forget Mode; INT Integer; REA Real; DBL Double; CPX Complex }

The Build Instruction and File (BIF) POP is one of the few written in interpretive code, and the only POP which calls itself recursively. BIF receives a pointer in W0 that will become the "address field" of the generated instruction. The effective address of the POP becomes the operation code of the generated instruction. Thus,

BIF FLD MOP

causes a FLD instruction to be generated, the address being supplied from W0. The MOP part of the address means "machine operation" and is a carry-over in terminology from the 920 FORTRAN II Compiler. The symbol FLD MOP has been equated to 066.

Building the instruction is only one of the functions BIF performs. It also takes care of indexing and indirect addressing, and of loading index registers. If the pointer in W0 is to a

dummy scalar, the job is quite easy: simply attach an indirect bit. If the pointer in W0 is to a constant subscripted array plex and the array is non-dummy, the value of the constant simply becomes an addend to apply to the instruction. If the array is dummy, however, the constant value is put into an AXB instruction (using index 1), and the generated instruction addresses indirectly through the dummy. The dummy itself contains an index 1 tag bit, so the constant value in index 1 will be added to the effective address.

The BIF POP maintains a table of what is in the various index registers, and if it finds that something it needs is already in the appropriate index register, it does not load it.

If W0 contains a fully subscripted array plex and the array is non-dummy, the non-constant part of the subscript is loaded into an index register and the constant part becomes an addend. In this case, the BIF POP determines whether any one of the three index registers already contains the needed value and, if so, uses that register. If not, it picks an index register at random and generates a LDX instruction. Since index register 1 is the only one that can be used for dummy arrays, for local arrays the BIF POP uses index 2, then 3, then 1. This way, it will not destroy something which was needed for a dummy array unless there were at least three independent non-dummy subscripts.

If W0 contains a fully subscripted array plex and the array is dummy, the non-constant part of the subscript is loaded into index 1 and the non-constant part, if non-zero, is built into an EAX instruction, tagged with index 1. This adds the constant part to the value of the index register.

If W0 contains a pointer to a multiple dummy, the code is almost the same as if it were a dummy array. The only difference is that the multiple dummy itself has both a tag and an indirect bit in it. The instruction referencing the multiple dummy has only an indirect bit.

LOGICAL EXPRESSION GENERATOR

Arithmetic expressions are handled with a single recursive pass through the expression, but logical expressions are done with two passes. The first logical expression pass is called Simplify Logical Expression. It takes the plex structure of the expression apart and puts it back together again in a form more convenient for generating code. Then the code is generated. There are two generating routines for logical expressions. The first – Evaluate Logical Expression – is used when the logical expression appears in a logical assignment statement or as a subprogram argument. It generates code designed to produce a logical value in the sign bit of the A or B register. The other logical expression generator generates code for jumping purposes. It has two entrances: Logical Expression Gen Jump If True and Logical Expression Gen Jump If False. The jumping generator is used when the logical expression appears in a logical IF statement or in a REPEAT WHILE statement. The generated code usually does not produce a logical value in a register but branches on the truth or falsity of the expression.

Since there are two separate generators, the same logical expression can produce quite different code, depending on the context in which it is used. Consider the expression

P .AND. Q; in the context

```
LOGICAL P, Q, R
R = P .AND. Q
```

it comes out

```
      LDA  P
      ETR  Q
      STA  R
```

whereas in the context

```
LOGICAL P, Q
REAL X, Y
IF (P .AND. Q)  X = Y
```

it comes out

```
      SKN  P
      BRU  1G
      SKN  Q
      BRU  1G
      LDP  Y
      STD  X
1G
```

SIMPLIFY LOGICAL EXPRESSION

Simplify Logical Expression is concerned mainly with simplifying relational operations. It notes that .GT. is the same as .LT. with the operands reversed, that .NOT. .EQ. is the same as .NE., etc. It transforms non-integer relationals into comparisons with zero, e.g.,

```
A .GT. B
```

is transformed into

```
B-A .LT. 0
```

It does not transform things that are already comparisons with zero. For example, it does not change

```
A .GE. 0
```

into

```
A-0 .GE. 0
```

Simplify logical expression does not convert integer relationals into comparisons with zero, since subtraction is an inaccurate way to compute integer relationals. Because of overflow problems in integer arithmetic, the relation

```
-5000000 .LT. +5000000      is .TRUE.
```

while

```
-5000000-5000000 .LT. 0     is .FALSE.
```

This is because -5000000 and +5000000 can both be represented as integers, and the negative one is certainly less; but the difference, -10000000, cannot be represented as an integer, and calculating it causes overflow and produces a positive result. This problem does not arise in floating-point calculations, since the overflow trapping routine always returns a result whose sign is correct. The problem does arise to some degree in very small floating-point numbers which are nearly equal. Suppose two numbers are near 10^{-77} and different, but the differ-

ence is, say, 10^{-79}. This difference is too small to be represented in floating point and is returned as zero, thus claiming that the two numbers are equal. The compiler does nothing to avoid this problem, since for most floating-point purposes the numbers are equal.

It is an insignificant loss to be unable to do integer relationals with subtraction, since most of them can be done more efficiently with SKG, SKL, SKE, and SKU instructions, which is how they are done.

Simplify logical expression takes cognizance of certain addressable comparisons with zero. The truth of a logical variable is stored in its sign bit; therefore the truth of

```
X .LT. 0
```

is stored in the sign bit of X, even though X is a floating-point variable. Therefore, the expression

```
X .LT. 0
```

is equivalent to X itself, considered as a logical variable. The ability to sense this fact enables evaluate logical expression to produce

```
LDA   X
MRG   Y
STA   P
```

for the statements

```
LOGICAL P
REAL X, Y
P = X .LT. 0  .OR.  -Y .GT. 0
```

Simplify logical expression also simplifies extended relationals, transforming

```
X .LT. Y .LT. C
```

into the more traditional

```
X - Y .LT. 0  .AND.  Y - C  .LT. 0
```

When simplify logical expression is finished, the A register flag on each of the plexes in the expression correctly indicates whether that subexpression will be evaluated in the A or B register, a fact that is of some concern to evaluate logical expression and of little concern to logical expression gen jump if true/false. This flag indicates where the result will come out if a logical value is being produced, and has little importance if a jump is being produced.

EVALUATE LOGICAL EXPRESSION

Evaluate logical expression produces code which is optimum for single relationals and for logical variables combined with .AND., .OR., and .EOR. . The code produced when relationals are combined with .AND., .OR., and .EOR. is sometimes less than optimum, as is the code when .NOT. is applied to things other than relationals. This generating routine

is strongly concerned with whether the various subexpressions produce results in the A or B register.

Most floating-point relationals produce results in the A register. For example,

```
X .LT. Y
```

produces

```
LDP   X
FLS   Y
```

giving an answer in the sign bit of the A register. Integer relationals, on the other hand, generally produce results in the B register since A is occupied making comparisons. Thus

```
J .LT. K
```

produces

```
LDA    K
COPY   (-1, B)
SKG    J
COPY   (0, B)
```

giving the answer in the sign bit of B.

When .NOT. is applied to a subexpression evaluated in B, a COPY (IB, A) instruction is produced, giving an answer in A. Similarly, when .NOT. is applied to a subexpression evaluated in A, a COPY (IA, B) instruction inverts the result and places it in B. The reason for this strange convention with .NOT. is that COPY instructions on the 9300 are very fast and there is no COPY instruction that will place the inverse of A into A or the inverse of B into B. In order to get the speed of COPY, one has to change registers.

To do a comparison of an arithmetic expression with zero, the compiler generates code to evaluate the arithmetic expression, allowing the result to come out with the wrong sign. (For example, X - Y*Z is more easily evaluated with the wrong sign than with the right sign.) If the sign comes out wrong, the compiler generates different code to compare with zero. For example, if it were supposed to produce a .LT. 0 comparison and the sign comes out wrong, it produces a .GT. 0 comparison instead, rather than negate the result of the expression. Naturally, for .EQ. 0 and .NE. 0, it makes no difference whether the sign comes out right or wrong.

If the sign of the expression comes out right, the following sequences of code are generated for comparisons with zero:

```
.GE. 0
     EOR   =-1        result in A
.GT. 0
     SKA   =-1
     EOR   =-1        result in A
```

```
.LE. 0
     SKU   0
     COPY  (-1, A)     result in A
.LT. 0
     no code           result in A
.NE. 0
     SKA   =-1
     COPY  (-1, A)     result in A
.EQ. 0
     SKA   =-1
     COPY  (-1, A)
     COPY  (IA, B)     result in B
```

The idea in evaluating .AND. and .OR. is to combine addressable quantities with ETR and MRG instructions and complicated quantities with skips and branches. In general, an .AND. or .OR. operator can have more than two operands, and the operands are divided into three classes:

1. Complicated operands evaluated in B.
2. Complicated operands evaluated in A.
3. Simple (addressable) operands.

Some, but not all, of the above classes may be empty in any particular case. Assume a long expression in which there are representatives of all three classes. In that case, the B register operands are evaluated first. If the operator is .OR., each B register operand other than the last is followed with

```
SKP =040000000
BRU label
```

where the label is defined after the last B register operand. If the operator is .AND., the code is the same except the SKP is replaced by SKB. The SKP instruction will skip if the B register is .FALSE., while the SKB will skip if the B register is .TRUE.

If there are no A register operands, the result is left in B. Otherwise, a COPY (B, A) instruction is generated at the place to which all the branches branch. Then come the complicated operands evaluated in A. All of these except the last are followed by skips and branches, the branches this time leading to a label at the end of the whole subexpression. If the operator is .OR., the skip is

```
SKA =040000000
```

if the operator is .AND., the skip is

```
SKL =-1
```

Finally, the addressable operands are combined with ETR and MRG.

The technique of evaluating the complicated operands first does not necessarily produce the fastest object code, but it does produce shorter code than would come from doing things the other way around. Since complicated subexpressions are combined with skips and jumps, there are never any stores in temps generated in connection with .AND. and .OR.

The evaluation of .EOR. is different. Unlike .AND. and .OR., it is necessary to evaluate all the operands of .EOR. in all cases to compute the result. For .EOR. all the operands are combined with EOR instructions, and intermediate results are stored in temps - the only instance in which logical temps are ever needed.

The jumping generators accept a logical expression pointer in W0 and a label on the bottom of the spec label list. They generate code which conditionally jumps to the label, depending on the truth or falsity of the expression. They remove the expression pointer from W0, but do not remove the label pointer.

The code generated for relational operators is quite straightforward except for the double skip in the following case:

```
IF (J+K .GE. M) GO TO 23

        LDA   J
        ADD   K
        SKG   M
        SKU   M
        BRU   23S
```

.OR. gen jump if true simply generates code for the individual operands, having each one jump if true. Similarly, .AND. gen jump if false generates code that jumps if false for each operand. Both these routines examine the terms being ANDed or ORed and try to do the easy ones first. If a simple logical variable is ANDed with a relational, for example, the logical variable will be tested first, regardless of the order in which they were written.

.OR. gen jump if false and .AND. gen jump if true are more complicated than the above routines. They involve the creation of a label, and not all the jumps go the same way. .AND. gen jump if true generates a label following the last instruction generated and causes the first n-1 operands to jump if false to that label. Then it causes the last operand to jump if true to the label it is supposed to be jumping to. Similarly, .OR. gen jump if false generates a label following the last instruction generated and causes the first n-1 operands to jump if true to that label. Then it causes the last operand to jump if false to the label it is supposed to be jumping to.

These routines also re-order the operands to do the simple ones first, but they also try to select an appropriate one to put last, taking into account that the last operand is generated to jump the opposite direction from the others. Most tests involve the same amount of code regardless of whether they are jumping if true or jumping if false. This is because most skip instructions can be reversed. For SKE there is SKU; for SKG there is SKL; but for SKN there is nothing. (Similarly, the double skip in the above example cannot be reversed in

the same number of instructions.) This non-reversibility of SKN is unfortunate since it is used to test logical variables as well as a few relationals. It takes an extra BRU to make SKN jump the other way. Therefore, .AND. gen jump if true and .OR. gen jump if false try to pick a term to put last that has this characteristic, i.e. of being more convenient to jump the wrong way. .OR. gen jump if false will put a simple logical variable last if it can find one, and .AND. gen jump if true will put .NOT. a logical variable last if it can find one.

The .NOT. operator is easily handled by the jumping generators. .NOT. gen jump if true simply pulls the .NOT. plex (see PUL under "POPs by Category") and goes to logical expression gen jump if false. Similarly, .NOT. gen jump if false pulls the .NOT. plex and goes to logical expression gen jump if true.

There are no special jumping generators for logical function calls, logical replacements, and .EOR. . These operations are generated by the appropriate parts of evaluate logical expression, and the result is tested in the A or B register.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION
PASS 3

IDENTIFICATION: Pass 3 (PASS3)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: JRS PASS3

PURPOSE: To perform the high level control of the operations which comprise pass 3. The following sequence of events takes place in pass 3:

1. Position system tape for next processor.
2. Copy output list (from pass 2) onto input list (for pass 3). This is the T2 file which may have overflowed to the X1 tape.
3. Generate literals (GENLIT subroutine).
4. Output the definition records (ODEFR subroutine)
5. Output data records (ODATAR subroutine)
6. Output reference records (OREFR subroutine)
7. Output end record (OENDR subroutine)
8. Output storage map (OUTMAP subroutine)
9. Return to MONITOR

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Get Next Input Word (GNIW)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
BRM GNIW
Result in A register and CINPW

PURPOSE: To get next input word from T2 (phase 3 principal input file) and store it in CINPW (current input word) and the A register.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Generate Literal List (GENLIT)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: JRS GENLIT

PURPOSE: To generate a list of literals (or constants) that are required in the object program.
Inputs are

CCONSLST	Complex Constant List
RDCONLST	Real/Double Constant List
ICONSLST	Integer Constant List
HCONSLST	Hollerith Constant List

Outputs are

LITERLST	Literal List
CADRLIST	Complex Constant Address List
RADRLIST	Real Constant Address List
DADRLIST	Double Constant Address List
IADRLIST	Integer Constant Address List
HADRLIST	Hollerith Constant Address List

Each constant on the constant lists is checked to determine if it is used. If it is not used, a dummy address is added to the appropriate address list and the constant is ignored; otherwise, the literal list is searched for a previous occurrence of the constant. If such an occurrence is found, its program address is added to the appropriate address list. If one is not found, the constant is added to the literal list, and its program address is added to the appropriate address list.

The basic program size (BPSIZE) is increased by the size of the literal list.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output External Definition Records (ODEFR)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   ODEFR
```

PURPOSE: To search the symbol table for all symbols that are entry points (external definitions) or global blocks. These items plus all the items in the block name list (BLNAMLST) are output in the form of type 1, subtype 0 or 2, binary records. The size of blank common is also output at this point unless it is zero.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Data Records (ODATAR)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: BRM ODATAR

PURPOSE: To process the pass 3 principal input file (T2) and produce the binary data records if requested and the object listing if requested.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output External Reference Records (OREFR)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   OREFR
```

PURPOSE: To output the external reference binary records (type 1, subtype 1 or 3) for the external references contained in the external reference list (XREFLIST).

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: File External Reference (FILEXREF)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

(X2)	=	address of 8-character name
(A)	=	addend
BRM		FILEXREF
(A)	=	chain address or addend

PURPOSE: If (XFILXREF) = NOP, then return with addend in A register. Otherwise, search XREFLIST for a previous reference to the symbol/addend pair. If a reference is found, update the chain address in the XREFLIST with (PROCTR), set (CHAINEND) positive, and return with the previous chain address as a result. If a reference is not found, add the name, addend, and (PROCTR) to the XREFLIST. Set (CHAINEND) negative and return with a zero chain address.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output End Record (OENDR)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   OENDR
```

PURPOSE: If no binary out is requested, then return; otherwise, output any accumulated full or partial records and output an end record (type 3) on a separate physical record.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Storage Map (OUTMAP)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: BRM OUTMAP

PURPOSE: To output the storage map on the LO device.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Convert Pointer to a Program Address (CPAD)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
(A)  =  pointer
BRM     CPAD
(A)  =  address
(B)  =  relocation code
```

PURPOSE: To compute the program address and the relocation code for the item referenced by the pointer using the current contents of ADDEND for an offset. If (XFILXREF) = SKIP and the referenced item is external, then the reference is chained and the appropriate chain address is used for the result. (CHAINEND) is set negative if this is the end of chain; otherwise, it is set positive. If (XFILXREF) = NOP and the referenced item is external, the address is relative to the symbol.

The relocation codes are

0 = Absolute
1 = Program Relative
2 = Blank Common
3 = Labeled Common Reference
4 = Global Reference
5 = External Subroutine Reference
6 = Undefined

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Convert Pointer to a Label (CPLAB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
(A) = pointer
BRM CPLAB
(A) = number of characters in CLABEL

PURPOSE: To generate in CLABEL through CLABEL+9 a label for the symbolic object listing corresponding to the item referenced by the pointer as follows:

Item	Label Form
current location	$
constant	= constant
local label	label S
non-local label	label $
generated label	label G
temporary	number TEMP
private temporary	number PTMP
other	8-character name

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Binary Buffer (BINBUF) Initialize (BINBUFIN)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
BRM   BINBUFIN
```

PURPOSE: To initialize BINBUF to zero.

REGISTERS: Registers X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: File Binary Data Word (FBDATA)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
(A) = data word
(B) = relocation code
BRM FBDATA

PURPOSE: If no binary output is required, return; otherwise, add data word to data record (type 0) in BINBUF, if there is room, and set the appropriate relocation bits according to the relocation code from the B register. In the event there is no room in BINBUF, call ANYBO to prepare and output the data record.

Relocation codes are

0 = Absolute
1 = Program Relative
2 = Blank Common
3 = Labeled Common Reference
4 = Global Reference
5 = External Subroutine Reference
6 = Undefined

Relocation codes 3, 4, 5 and 6 may be program relative or absolute depending upon whether or not the address is the end of an external reference chain. (CHAINEND) if negative indicates absolute, otherwise relocatable.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Any Data Record Binary Output (ANYBO)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: BRM ANYBO

PURPOSE: To test for the accumulation of a full or partial binary data record (type 0). If there is none, return; otherwise, construct the data record according to the SDS standard binary format, call CKSOUT for the checksumming, and output the record.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Checksum and Output (CKSOUT)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   CKSOUT
```

PURPOSE: Return if no binary out is required; otherwise, compute the checksum for the logical record (SDS standard binary language) which is in buffer BINBUF. If there is room in buffer ALBINBUF for the logical record from BINBUF, it is transferred to ALBINBUF (buffer for 1 or more logical records in packed form). If ALBINBUF does not have room for the BINBUF logical record, the packed records in ALBINBUF are output and then the BINBUF logical record is transferred to ALBINBUF.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Unconditional Binary Output (UNBINOUT)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   UNBINOUT
```

PURPOSE: To output the contents of the buffer ALBINBUF unless it is empty or no binary output is required ((XBINOUT) ≡ NOP).

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Check BO Output (CHECKBO)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   CHECKBO
```

PURPOSE: To check for any BO output in process; if output is in process, wait for completion and then return.

REGISTERS: Registers X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Check GO Output (CHECKGO)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM  CHECKGO
```

PURPOSE: To check for any GO output in process; if such output is in process, wait for completion and then return.

REGISTERS: Registers X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Initialize Line (ILINE)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
BRM  ILINE
```

PURPOSE: To initialize the output line used by the output formatting routines to blanks and to set the character position pointer to character position 1 of the line.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: PAGE

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: BRM PAGE

PURPOSE: To eject to top of page on:

1. LO device using MONITOR I/O if (XMONITOR)≡SKIP.
2. LP1A device using own I/O code if (XMONITOR)≡NOP.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: SPACE

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: BRM SPACE

PURPOSE: To upspace one line on either the printer or typewriter as follows:

	(XMONITOR) ≡ SKIP	(XMONITOR) ≡ NOP
(PRTY) ≡ SKIP	TY device using MONITOR	TY1A using own I/O code
(PRTY) ≡ NOP	LO device using MONITOR	CR1A using own I/O code

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Print Line (PLINE)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE: BRM PLINE

PURPOSE: To transmit the output line to the device specified in the following table:

	(XMONITOR)≡SKIP	(XMONITOR)≡NOP
(PRTY)≡SKIP	TY device using MONITOR	TY1A using own I/O code
(PRTY)≡NOP	LO device using MONITOR	CR1A using own I/O code

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Define Special Conversion Line (DSCLSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
BMA  DSCLSUB
PZE  α
```

PURPOSE: To save the current character position pointer, to define a special output line beginning at α and to set the character position pointer to zero.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Restore Normal Output Line (RLINE)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM  RLINE
```

PURPOSE: To restore the output line and character position pointer to what it was prior to the last call on DSCLSUB.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Increment Character Position by 1 (ICP1)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
BRM   ICP1
```

PURPOSE: To increment the output line character position pointer by 1.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Decrement Character Position by 1 (DCP1)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
BRM   DCP1
```

PURPOSE: To decrement the output line character position by 1.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Increase Character Position Subroutine (ICPSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
(A) = N
BRM  ICPSUB
```

PURPOSE: To increase character position pointer by N.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Set Character Position Subroutine (SCPSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
(A) = N
BRM   SCPSUB
```

PURPOSE: To set the character position pointer to the N^{th} character position of the output line.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Store Character (STC)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
$(A_{18-23}) = C$
BRM STC

PURPOSE: To store the character C in the output line at the position designated by the character position pointer.

REGISTER(s): Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Alpha Subroutine (OASUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
(A)  =  N
BRM     OASUB
PZE     α
```

PURPOSE: To transfer the character string beginning at α (and N characters long) to the next N character positions of the output line. Increase the character position pointer by N.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Alpha Minimum Number of Characters (OAMNCSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
(A)  = N
BRM    OAMNCSUB
PZE    α
```

PURPOSE: To transfer the character string beginning at α to the next M character positions of the output line. Increase the character position pointer by M. M is the lesser of two items:

1. N
2. the number of characters in the character string prior to the first blank (060)

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Octal Subroutine (OOSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
(A)  = N
(B)  = V
BRM    OOSUB
```

PURPOSE: To convert V to an N-character octal character string with preceding zeros and to store it in the next N character positions of the output line. Increase the character position pointer by N.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Decimal Subroutine (ODSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
(A)  = N
(B)  = V
BRM    ODSUB
```

PURPOSE: To convert V to signed (if negative, otherwise unsigned) decimal character string with preceding blanks and to store it in the next N character positions of the output line. Increase the character position pointer by N.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Decimal Minimum Number of Characters (ODMNCSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
(A)  = V
BRM    ODMNCSUB
```

PURPOSE: To convert the value V to a signed (if negative, otherwise unsigned) character string and to store it in the next N character positions of the output line. Increase the character position pointer by N. N is the number of character positions required to contain the number and its sign if present.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Real Subroutine (OREALSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:

```
(A, B)  =  argument
BRM        OREALSUB
(A)     =  number of characters output (N)
```

PURPOSE: To convert the 2-word real argument to its BCD representation and to store it in the next N character positions of the output line. Increase the character position pointer by N. N is the fewest number of characters required to represent the number properly.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Output Double Precision Subroutine (ODOUBSUB)

OVERLAY SECTION: Pass 3

CALLING SEQUENCE:
```
BMA  ODOUBSUB
PZE  argument
```
(A) = number of characters output (N)

PURPOSE: To convert the 3-word argument (double-precision value) to its BCD representation and store it in the next N character positions of the output line. Increase the character position pointer by N. N is the fewest number of characters required to represent the number properly.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Read Debug (READBUG)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE: BRM READBUG

PURPOSE: To load the compiler debug system into memory from the system tape.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Read Phase 2 (READP2)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE: BRM READP2

PURPOSE: To load phase 2 of the compiler into memory from the system tape.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Read Phase 3 (READP3)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE:
```
BRM   READP3
```

PURPOSE: To load phase 3 of the compiler into memory from the system tape.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: FORTRAN Rewind Temporary (FREWIND)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE:

```
(X1)  =  symbolic tape name
BRM      FREWIND
```

PURPOSE: To rewind the symbolic tape designated by the name in X1.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: FORTRAN Write Temporary Tape (FWRITE)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE:

(X1) = symbolic tape name
(A) = address of block to write
(B) = number of words in block
BRM = FWRITE

PURPOSE: To write on the designated symbolic tape data block specified, preceded by two control words as follows:

Word 1 : number of data words (N)
Word 2 : checksum
Word 3 } data words
.
.
.
Word N + 2 }

The checksum is the sum of data words.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: FORTRAN Read Temporary Tape (FREAD)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE:

```
(X1) = symbolic tape name
(A)  = address of block
BRM    FREAD
```

PURPOSE: To read from the designated symbolic tape the data block into the memory block area specified.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Error Subroutine (ERRORSUB)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE:

```
BRM    ERRORSUB
TEXT   4,mmmm
```

PURPOSE: To type the following on the TY device:

C/R *ERROR*$_{\wedge\wedge}$IIIII$_{\wedge}$mmmm

where IIIII is the octal location of the call on ERRORSUB.

REGISTERS: Registers X1, X2, X3 are maintained.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: MONITOR Typeout (MONTYPEO)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE:

(A_{0-8}) = 0777

(A_{9-23}) = address of first word

(B) = number of words

BRM MONTYPEO

PURPOSE: To type on the TY device the words specified.

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: Type Character (TYPECHAR)

OVERLAY SECTION: FORTRAN Control

CALLING SEQUENCE: (A_{0-5}) = character
BRM TYPECHAR

PURPOSE: To type the character designated on the TY device.

DESCRIPTION OF PRINCIPAL SYMBOLS IN PASS 3

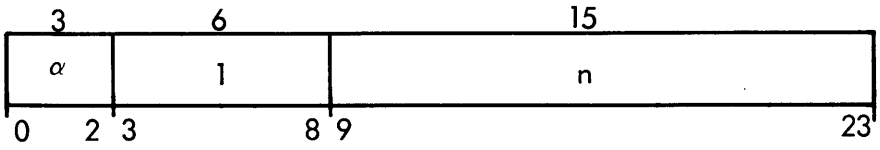
ADDEND	Contains the current addend that is to modify a symbolic address; e.g., if an object instruction references A + 5, the reference would be indicated by a pointer to A in the symbol table and an addend of 5.
BINBUF	Binary buffer in which a logical binary record is developed.
BOTRIG	Negative if BO output required.
BPSIZEML	Contains the basic program size up to but not including the literals (constants).
CHAINEND	Negative if the address of the instruction being processed (of the object program) is an external reference and absolute 0 designating end of the chain as opposed to being positive indicating relocatable (possibly 0).
CINPW	Current input word contains a copy of the last word input from the T2 file (pass 2 output; pass 3 input).
CLABEL	A buffer that contains the label generated by the Convert Pointer to a Label (CPLAB) subroutine.
DEBUGEND	Contains the address of the location immediately following the FORTRAN compiler's debug routines. When in the debug mode, this location is the first cell of working space.
DEBUGORG	This location is the first cell of the FORTRAN compiler's debug system. When not in the debug mode and in the in-line code mode, this location is the first cell of working space.
FILESIZE	Contains a count of the number of records in the input file (T1 or T2).
GOTRIG	Negative if GO output required.
LDLAB	Load label contains the 8-character name corresponding to the global or labeled common block in which initial values are being loaded because of a data statement.
LINE	Output line buffer.
OPTAB	Table of operation mnemonics.
POSIFBOX	Positive if any BO output has been transmitted whose status must be checked.
POSIFGOX	Positive if any GO output has been transmitted whose status must be checked.
PROCTR	Assembly program counter.
RECSIZE	Contains a number that designates the number of data words per record in the T1 and T2 files. The actual record size will be two words greater to facilitate the two control words.

SHOESIZE	Its value represents the maximum size logical record that may be accommodated in the physical binary record under construction.
SPCTR	Special program counter used in processing data statements only.
TRAADR	Transfer address contains the object program transfer address if the program unit being compiled is a main program.
WRITECNT	Contains a count of the number of records written since the last rewind on the files T1 and T2.
XBINOUT	EXU XBINOUT will result in a skip if any form of binary output (BO or GO) is required; otherwise, it will result in a NOP.
XFILXREF	EXU XFILXREF is set to skip if external references are to be added to the XREFLIST in the CPAD subroutine or set to a NOP if external references are not to be filed in the CPAD subroutine.
XLIST	EXU XLIST will result in a skip if an object listing is required; otherwise, it will result in a NOP.
XMAINP	EXU XMAINP will result in a skip if the program unit being compiled is a main program; otherwise, it will result in a NOP.
XMONITOR	EXU XMONITOR will result in a skip if the compiler is in a MONITOR interfaced form; otherwise, it will result in a NOP. The only use of NOP is for a free standing version for debugging purposes.

T2 FILE (File Pass 2 Output/Pass 3 Input)

T2 is the file generated by pass 2 and input to pass 3. T2 file may overflow to the X1 tape. In pass 3 the GNIW subroutine is used to get the next word from the T2 file. The following are the item forms:

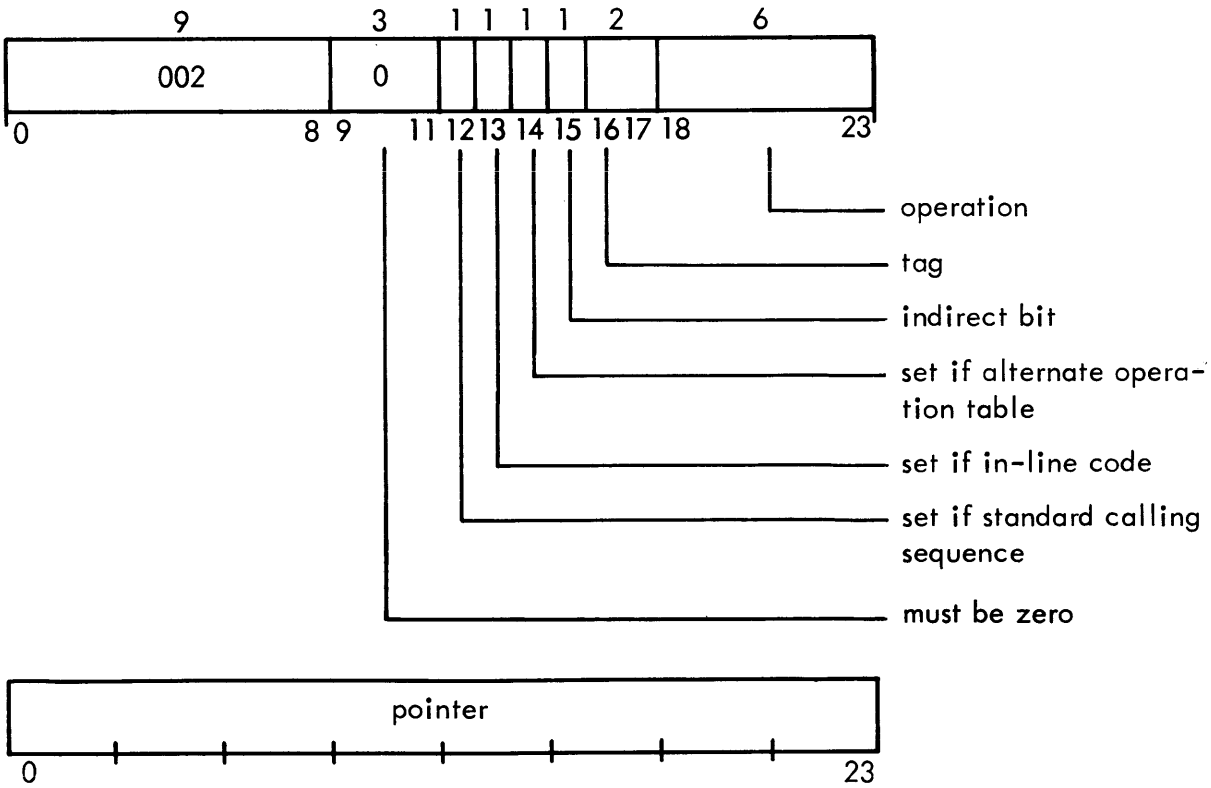
Constant



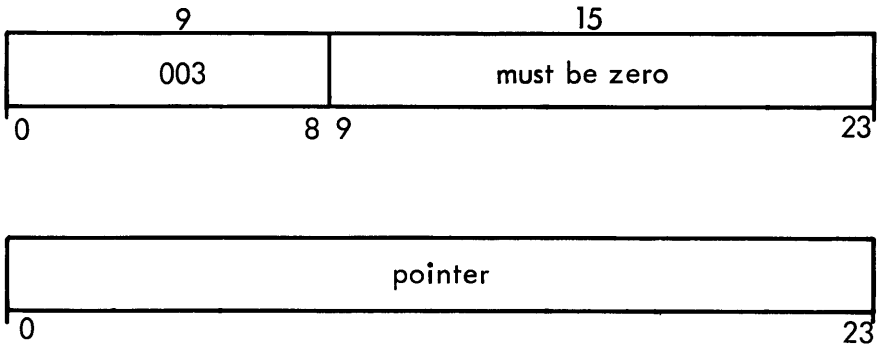
The next n words are absolute and of the following format:

α	Format
0	octal
1	BCD
3-7	not defined

Instruction

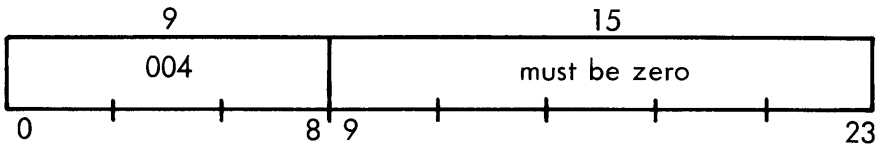


Definition



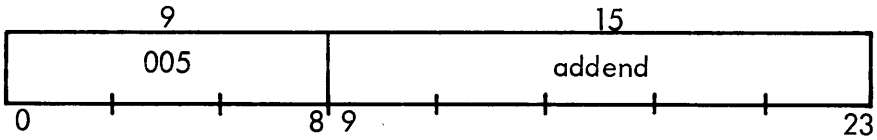
(will not be present if no listing)

Use Program Counter

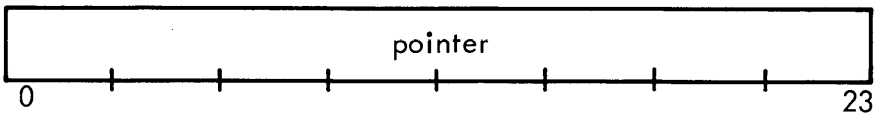


(use program counter and restore to last)

Use Special Location Counter

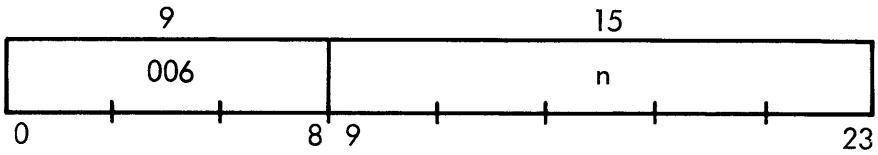


(use special location counter)



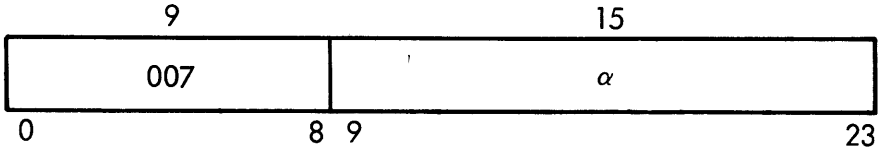
(points to symbol table)

Increment Location Counter



Increase current location counter by n

Special Instructions



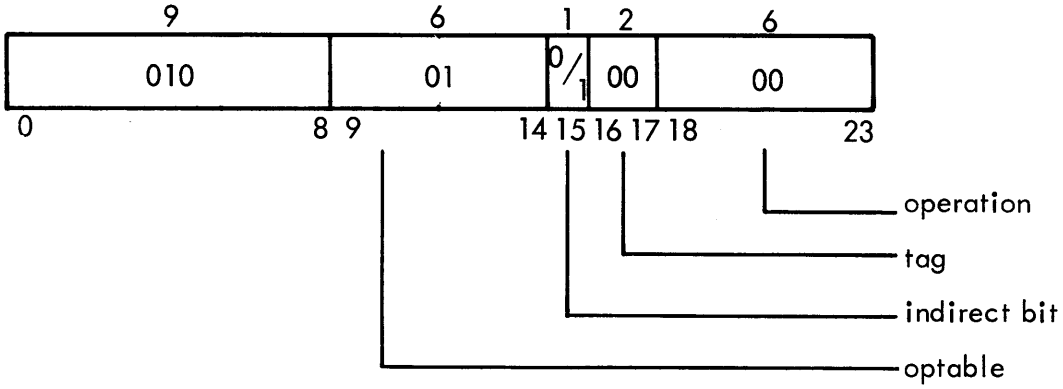
α is address of a variable length item of the following form:

a. 24-bit absolute instruction

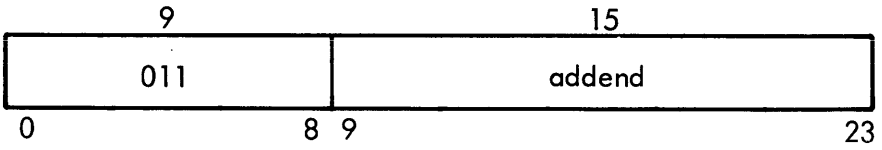
b. 4-character operation mnemonic

c. packed character string terminated by a $ to be used in the symbolic address field

Real-Time PZE (indirect if protected)

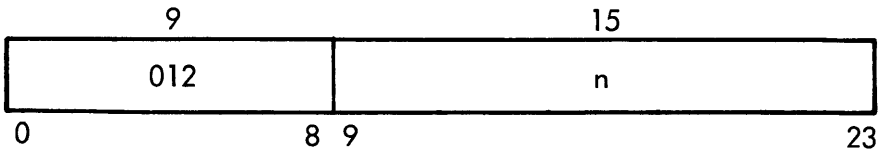


Addend



Addend applies to subsequent instruction.

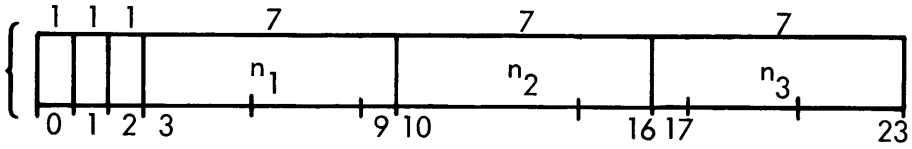
Line



n words follow in compressed format.

Each line consists of a set of control words followed by a character string. The following describes the compressed format:

1 or more control words

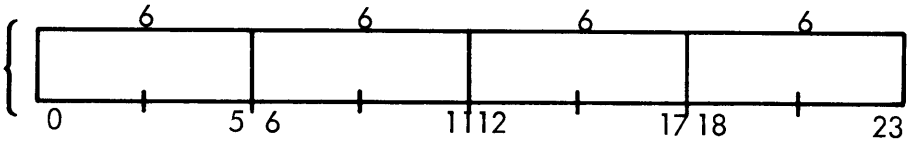


Bit 0 = 1 last control word

Bit 1 = 0 n_1 blanks; n_2 characters; n_3 blanks

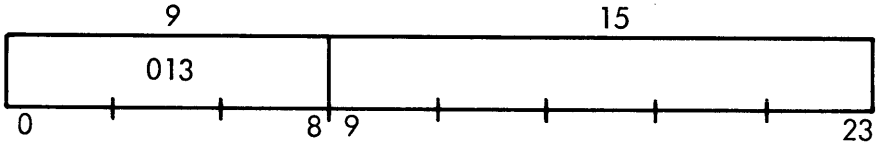
= 1 n_1 characters; n_2 blanks; n_3 characters

0 or more words of character string



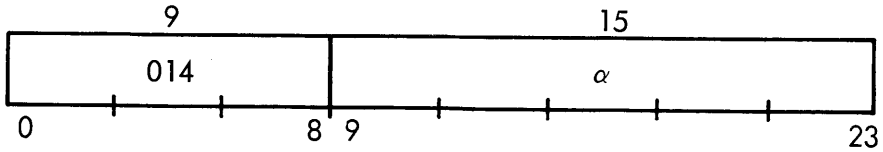
Note: Two or fewer contiguous blanks are normally best left in the character string from an efficiency standpoint.

End T2

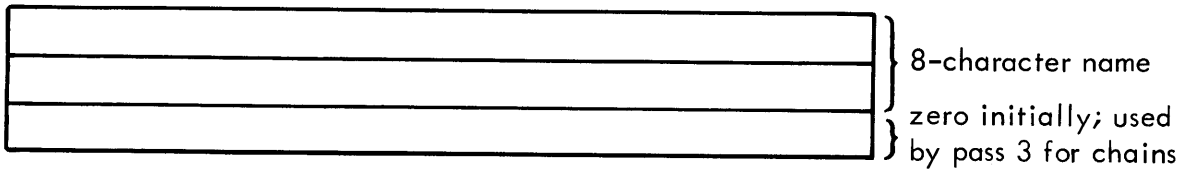


end of T2

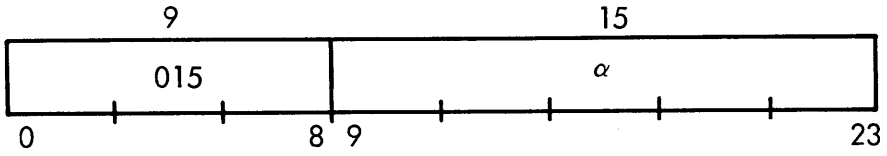
Special BRM



α is address of a 3-word item of format

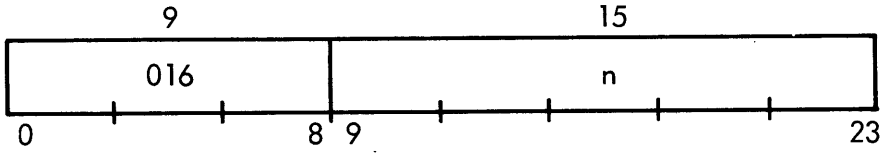


Special BMA



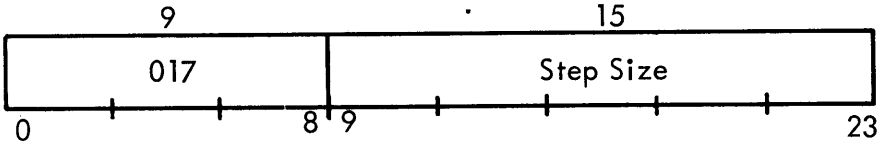
α is same as above

Shift Instruction

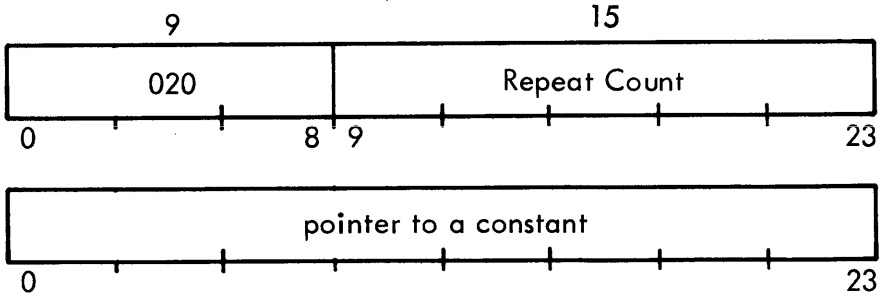


The next n words are shift instructions

Step



Repeated Load

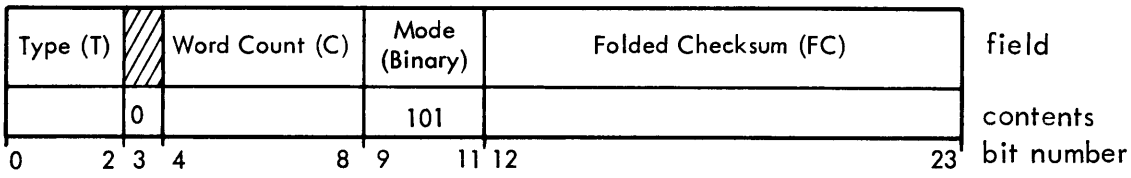


SDS STANDARD BINARY LANGUAGE

The following description specifies the subset of the standard binary language for the SDS 9300 Computers that may be generated by the compiler.

In the following description of the language, a file is the total binary output from the compilation of one program or subprogram. A file is both a physical and a logical entity since it can be subdivided physically into unit records and logically into information blocks. While a unit record (in the case of cards) may contain more than one record, a logical record may not overflow from one unit record to another.

1. CONTROL WORD - first word in each type of record



T	Record Type
000	Data record (text)
001	External references and definitions, block and program lengths
010	Not used
011	End record (program or subroutine end)
100, 110, 111	Not used
101	Data Statement record

C = total number of words in record, including Control Word

Note that the first word contains sufficient information for handling these records by routines other than the loader (that is, tape or card duplicate routines). The format is also medium-independent, but preserves the mode indicator positions desirable for off-line card-handling.

An exclusive OR checksum is used. If the symbol -- is used to denote exclusive OR, and W_i denotes the i-th word in the record ($1 \leq i \leq C$), then

$$FC = (W_1)_{0-11} \text{ -- } (C)_{0-11} \text{ -- } (C)_{12-23} \text{ -- } 07777$$

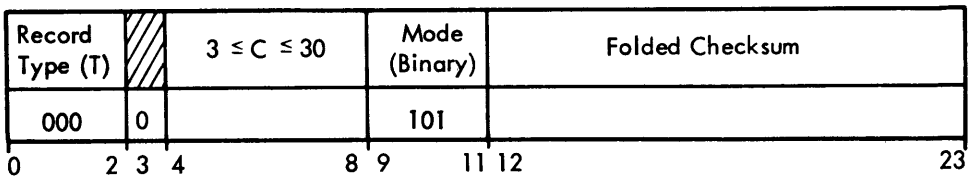
where

$$C = W_2 \text{ -- } W_3 \text{ -- } \ldots \text{ -- } W_c$$

2. DATA RECORD FORMAT (T = 0)

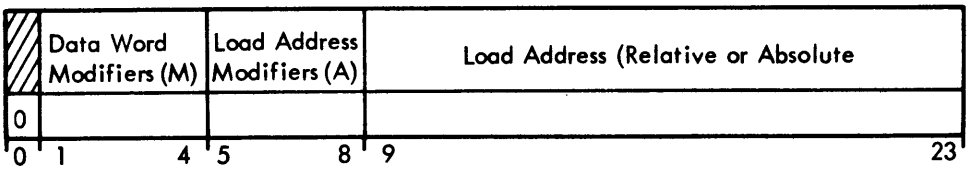
Word 1

Control Word



Word 2

Load Address Word

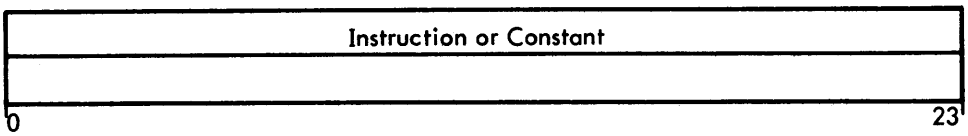


The presence of bits in field M indicates the presence of words n+3, n+4, n+5, and n+6 (shown below):

If bit 4 is a 1, word n+3 (load relocation) is present.
If bit 3 is a 1, word n+4 (common relocation) is present.

Word 3

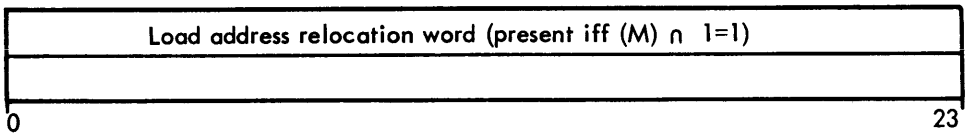
Data Word 1



Words 3 through n+2 contain instructions or constants (where 1 n 24)

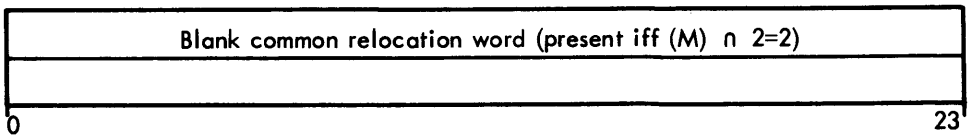
Word n+3

Load Relocation



Word n+4

Common Relocation



Words n+3 through n+6 are modifier words. Each bit in each of these words corresponds to a data word (bits 0 through 23 correspond to word 3 through n+2, respectively). A bit set to 1 indicates that the specified data word required modification by the loader. There are two types of modification (and hence two possible modifier words that are indicated in data records. Presence of a modifier word is indicated by the M (data word modifier) field in the load address word.

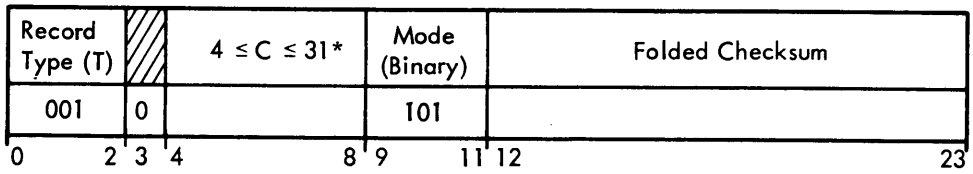
The load address is subject to modification as indicated by the A field of the load address word as follows ((A) = 0 means absolute):

(A) ∩ 1 = 1, current load relocation bias is added to load address

3. EXTERNAL REFERENCES AND DEFINITIONS, BLOCK AND PROGRAM LENGTHS (T = 1) (Includes labeled COMMON, blank COMMON and program lengths)

Control Word

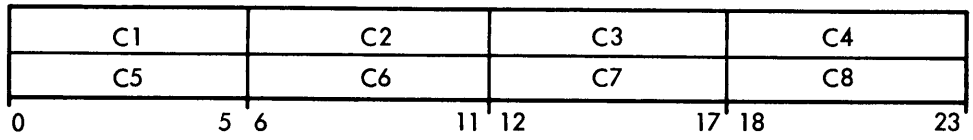
Word 1



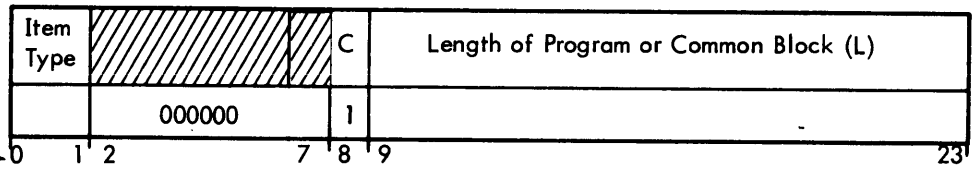
*From 1 to 10 items per record.

Common Length or Program Length Item

1- to 8-Character Label



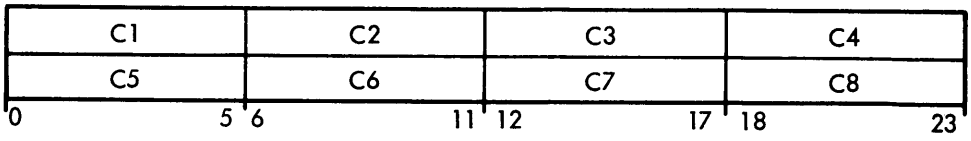
Length Word



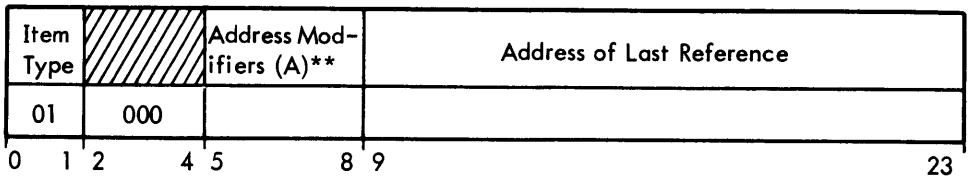
C = 1 if (L) is length of a labeled common block.

External Reference Item

Label



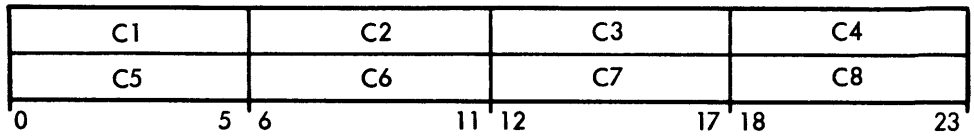
Chain Word



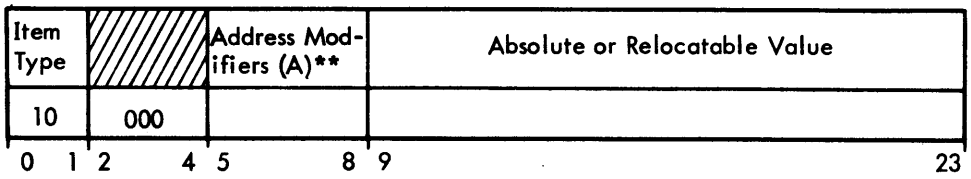
**See data record, load address word, for interpretation.

External Definition Item

Label



Value Word

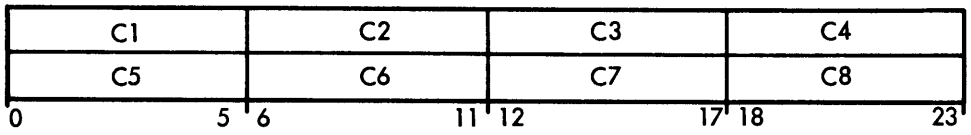


**See data record, load address word, for interpretation.

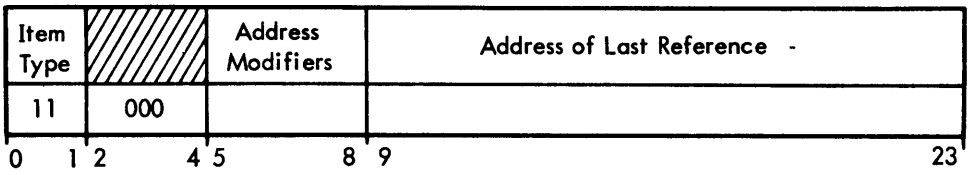
External symbolic definitions include subroutine "identification" as a subset and require no special treatment of subroutines with multiple names.

External Reference with Addend Word*

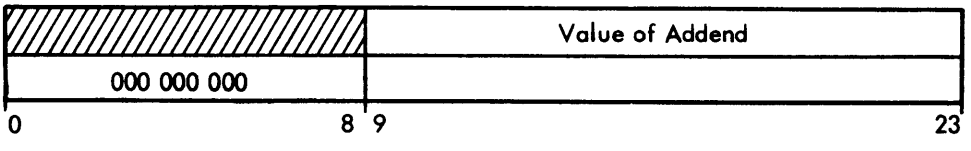
1- to 8-Character Label



Chain Word



Addend Word



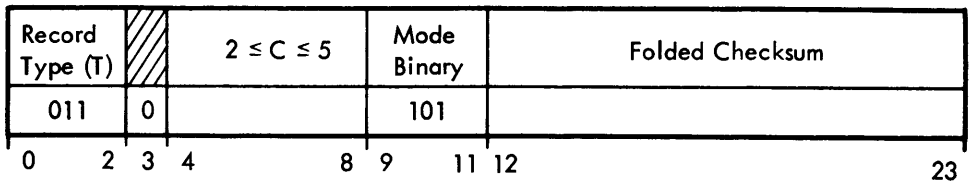
*One of these items for each unique reference; e.g., each of the following references is represented by a separate item:

A+5, B+5, B+6, C+2, C+5

4. END RECORD (T=3)

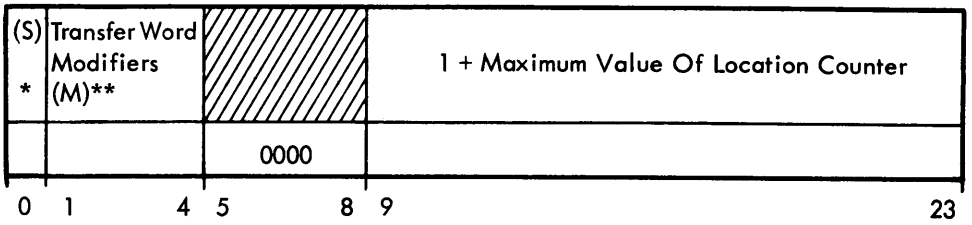
Word 1

Control Word



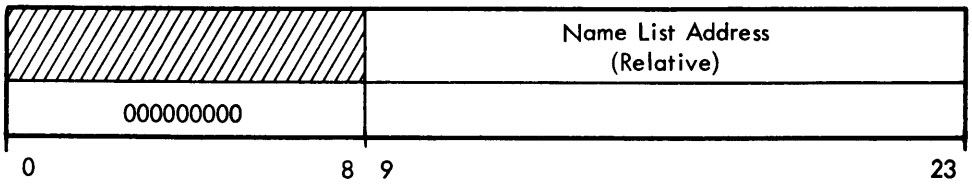
Word 2

Length of Program



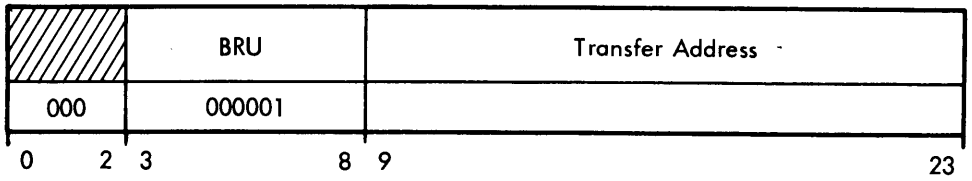
Word 3

Name List Location Word*



Word 4

Transfer Word *



*If S = 1, word 3 is the Name List Location Word and word 4 is the Transfer Word.

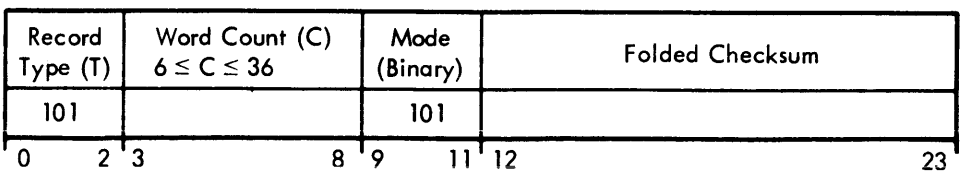
If S = 0, word 3 is the Transfer Address Word; the Name List Location Word is omitted.

**See data record description for interpretation.

5. DATA STATEMENT RECORD FORMAT (T=5)

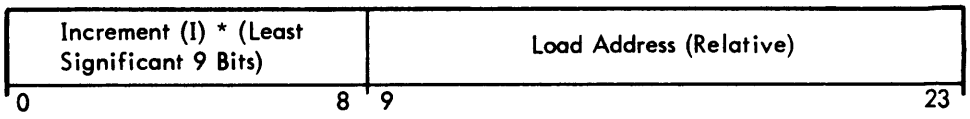
Word 1

Control Word



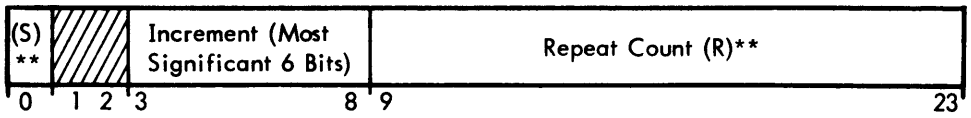
Word 2

Load Address Word



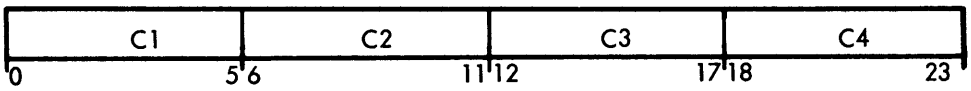
Word 3

Repeat Count Word

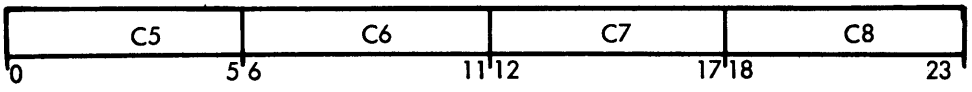


Word 3+S

Common Block Label

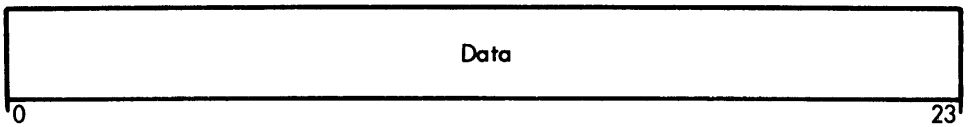


Word 3+2S



Word 4+2S

Data Word 1



⋮

Words 4+2S through C contain constants.

*The increment (I) is added to the relative load address to obtain the next relative load address for a repeat load.

**If S = 1, words 6 through C ($6 \leq C \leq 36$) are loaded relative to the labeled common block origin.

If S = 0, words 4 through C ($4 \leq C \leq 36$) are loaded relative to the subprogram origin.

***Data words 4+2S through C are repeatedly loaded (R) times in increments of (I).

SDS PROGRAM LIBRARY
PROGRAM DESCRIPTION

IDENTIFICATION: 9300 FORTRAN IV Compiler Debugging System

PURPOSE: To aid in debugging of the 9300 FORTRAN IV compiler.

STORAGE: Approximately 06100 relocatable locations including:

IOASTE	200-word Instruction or Address Search Table (1 word per entry)
FREE	400-word insertion block used to store all snapshots and insertions. Each insertion requires 2 + n words from the insertion block, where n is the number of words logically inserted. Each snapshot requires 3 +2n words from the insertion block where n is the number of memory blocks to dump.
HISL	596-word block used to record the recent history of the compiler control.

USE: Switch Settings:

Switch	Interpretation
1	Set - Instruction/address search for trap Reset - Bypass instruction/address search for trap
2	Set - Automatic history print on table cycle Reset - No automatic history print on table cycle
3	Set - Terminate current request Reset -
5	Set - Build history Reset - Bypass building of history
6	Set - Retrieve control immediately Reset -

History

Optionally, a history of the program flow is maintained. There is room in memory to maintain a history equivalent to one printed page. An H request will print the current history. The option also exists for automatically printing the history every time the history table cycles, thus,

USE: (cont)

producing a complete history. Table 1 contains a sample history print.

History will not be maintained at levels below the level specified in THRLEV (threshold level) which may be altered while debugging.

Trapping

When in the trapping mode, the debugging system will type the following control line and transfer control to the typewriter:

Tnnn lllll mmmm

where:

T	indicates Trap,
nnn	is the level number in decimal,
lllll	is the location in octal,
mmmm	is the mnemonic code for the instruction to be executed.

At this point the user may type in any of the valid requests.

The trapping mode may be entered in any of the following ways:

1. Setting Sense Switch 6.
2. Executing a Trap Enter Instruction.
3. Returning to a higher level at which trapping had not been terminated.
4. Exhausting a trap skip count.
5. Executing an instruction which is in the Instruction/Address Table in the proper form.
6. Executing a typewriter snapshot.

Once in the trapping mode, the following means of exiting are provided:

1. Executing a Trap Exit Instruction.
2. Typing a trap skip count.
3. Typing a level trap exit.

USE: (cont)

Input Request Rules

All requests except the continue request begin with a 1 to 4-character request name.

The complete request is read before any part of it is performed.

Elements of an input list are separated by commas (,).

Blanks, carriage returns, and tabs are ignored.

All requests are terminated with a period (.).

The delete character deletes the request.

The following syntactic elements form parts of the requests:

number	if preceded by the digit 0, then octal; else decimal
operators	+ add - subtract * indirect addressing of value so far; e.g., 010 + 8 * + 3 where 020 contains 040000107 0107 contains 00000225 denotes 0230. ~ flags addressed. # pointer
expression	consists of a string of numbers, symbols, and operators
block	memory block or list name or $ list name ($ denotes dump list only to first reserve.)
memory block	expression/expression or expression The memory block is defined as expression 1 through (expression 1 + (expression 2 - 1)).

Any of the functional descriptions used in the request descriptions which designate a location, number, or word may be an expression.

USE: (cont)

Request Descriptions

.	Continue; i.e., execute the instruction and trap next instruction.
X number.	Skip the trapping of the next n instructions to be executed at this level, then resume trapping.
X.	Discontine trapping at this level.
G location.	Go to location specified (interpretively).
A location, word list.	Alter the contents of memory beginning at the location specified.
CRD.	Causes the card reader to be the input device.
TYP.	Causes the typewriter to be the input device.
R location list.	Remove the snapshot, trap enter or trap exit instruction from the locations specified
H.	Print history page.
P.	Eject page on printer.
IB location, word list. IA location, word list.	Insert logically before/after the location specified the instructions in the word list.
DT block list. DP block list.	Dump on typewriter (DT) or on the printer (DP) the blocks specified. A block may be either a memory block or a compiler list.
TN location list. TX location list.	Insert logically before the locations specified trap enter (TN) or trap exit (TX) instructions.
SST location, life, block list. SSP location, life, block list.	Insert logically before the location specified a typewriter (SST) or a printer (SSP) snapshot. When the snapshot is executed, the blocks specified will be dumped on the typewriter/printer. The snapshot will automatically be removed after being executed the number of times designated by life. Typewriter snapshots also enter the trapping mode whereas printer snapshots do not. An S will be typed on a typewriter snapshot control line instead of the T for trap control lines.

USE: (cont)

SEP memory block, value 1, value 2, mask.
SUP memory block, value 1, value 2, mask.
SET memory block, value 1, value 2, mask.
SUT memory block, value 1, value 2, mask.

Search(S) the memory block specified for all words that are equal (E) or unequal (U) to value 1 through value 2 using the mask specified. All successful searches are either printed (P) or typed (T).

ITA word list.
ITD word list.

Instruction table add (ITA) or delete (ITD) the words specified.

When the routine is not in the trapping mode and prior to the execution of each instruction, a search is made of the instruction table to determine whether the current instruction should cause the trapping mode to be entered. Each word in the instruction table contains three parts:

type - bits 0, 1

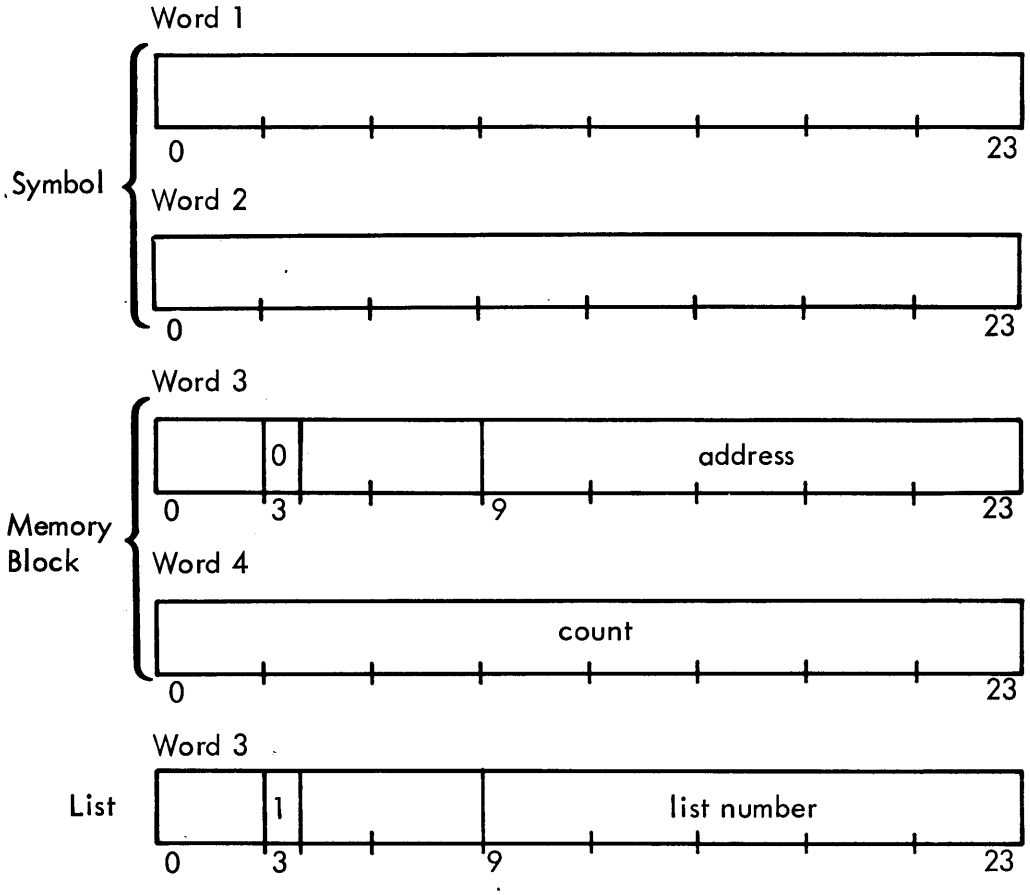
operation code - bits 2 - 8

effective address - bits 9 - 23

The trapping mode will be entered if the current instruction is in the instruction table in the proper form, i.e.,

type = 0 not valid
1 effective address match
2 operation code match
3 effective address and operation code match

USE: (cont) Symbol Table (Entry Format - 4 words per entry)



Symbols

R	Dummy can be used as relocation register.
THRLEV	History threshold level.
ALL	Designates all lists to be dumped in a dump request.
EA	Effective Address register.
NDCP	Number of dump columns to print.
NDCT	Number of dump columns to type.
INSTR	Instruction Register.
LOC	Location Register
SYMTC	contains the negative of the number of symbol table entries minus 1.

USE: (cont)

Error Messages

Error messages are typed on the typewriter in the following form:

ERROR llllll eeee

where:

llllll is the octal location of the error call.
eeee is an error type mnemonic.

The following errors may occur:

Mnemonic	Meaning and Action
MACH	Either machine error or a list entry is in location 0; terminate dumping of current list.
Z CT	Zero count in a memory block dump request; terminate request.
IREQ	Illegal request name; request terminated.
R TL	Request too long (exceeds 80 characters); request terminated.
N DP	Attempt to remove a debugging POP which does not exist; continue same request.
ITOV	Instruction/Address Table overflow; continue same request.
NO I	Attempt to delete from the Instruction/Address Table an entry which does not exist; continue same request.
STX 1 STX 2	Syntax error; request terminated.
NSYM	Symbol not found in symbol table; request terminated.
F OV	Free block overflow (insertion block); request terminated.
NO A	No address found in symbol table corresponding to assembled snapshot request; this part of request ignored.

Table 1
Sample Three Columns of a 6-Column History Print

Execution is columnwise.

repeat count	level number	location group	repeat count	level number	location group	repeat count	level number	location group
1	29	00300	1	29	00400	1	29	00600-00603
2	29	00400-00401	1	29	00500-00501	1	29	00600
1	29	00400	1	29	00600-00603	1	29	00700-00702
1	29	00500-00501	1	29	00600	2	29	00702
1	29	00600-00603	1	29	00700-00702	1	29	00704
1	29	00600	2	29	00702	2	29	00100-00102
1	29	00700-00702	1	29	00704	1	29	00100-00104
2	29	00702	2	29	00100-00102	1	29	00200
1	29	00704	1	29	00100-00104	1	29	00300
2	29	00100-00102	1	29	00200	2	29	00400-00401
1	29	00100-00104	1	29	00300	1	29	00400
1	29	00200	2	29	00400-00401	1	29	00500-00501
1	29	00300	1	29	00400	1	29	00600-00603
2	29	00400-00401	1	29	00500-00501	1	29	00600
1	29	00400	1	29	00600-00603	1	29	00700-00702
1	29	00500-00501	1	29	00600	2	29	00702
1	29	00600-00603	1	29	00700-00702	1	29	00704
1	29	00600	2	29	00702	2	29	00100-00102
1	29	00700-00702	1	29	00704	1	29	00100-00104
2	29	00702	2	29	00100-00102	1	29	00200
1	29	00704	1	29	00100-00104	1	29	00300
2	29	00100-00102	1	29	00200	2	29	00400-00401
1	29	00100-00104	1	29	00300	1	29	00400
1	29	00200	2	29	00400-00401	1	29	00500-00501
1	29	00300	1	29	00400	1	29	00600-00603
2	29	00400-00401	1	29	00500-00501	1	29	00600
1	29	00400	1	29	00600-00603	1	29	00700-00702
1	29	00500-00501	1	29	00600	2	29	00702
1	29	00600-00603	1	29	00700-00702	1	29	00704
1	29	00600	2	29	00702	2	29	00100-00102
1	29	00700-00702	1	29	00704	1	29	00100-00104
2	29	00702	2	29	00100-00102	1	29	00200
1	29	00704	1	29	00100-00104	1	29	00300
2	29	00100-00102	1	29	00200	2	29	00400-00401
1	29	00100-00104	1	29	00300	1	29	00400
1	29	00200	2	29	00400-00401	1	29	00500-00501
1	29	00300	1	29	00400	1	29	00600-00603
2	29	00400-00401	1	29	00500-00501	1	29	00600

Table 1
Sample Three Columns of a 6-Column History Print (cont.)

1	29	00400	1	29	00600-00603	1	29	00700-00702
1	29	00500-00501	1	29	00600	2	29	00702
1	29	00600-00603	1	29	00700-00702	1	29	00704
1	29	00600	2	29	00702	2	29	00100-00102
1	29	00700-00702	1	29	00704	1	29	00100-00104
2	29	00702	2	29	00100-00102	1	29	00200
1	29	00704	1	29	00100-00104	1	29	00300
2	29	00100-00102	1	29	00200	2	29	00400-00401
1	29	00100-00104	1	29	00300	1	29	00400
1	29	00200	2	29	00400-00401	1	29	00500-00501
1	29	00300	1	29	00400	1	29	00600-00603
2	29	00400-00401	1	29	00500-00501	1	29	00600

Table 2
Compiler Interfaces and Intra Global Symbols

Interfaces

Compiler interpreter calls the debugging system as follows:

```
BRM     F4DEB

return
```

Initialization of the Debugging System (namely erasing history, setting input device to typewriter) may be accomplished as follows:

```
BRM     INIDEBUG

return
```

Registers B, X1, X2, and X3 are saved upon entry and restored prior to return.

A G (Go to) request will change the contents of X3 (location counter) and return via:

```
BRU     INTERP
```

instead of performing the normal return. In this case registers B, X1, and X2 are not restored.

X3	bits 0-8	1
	bits 9-23	location counter
X2	bits 9-23	operation code (0-0177)
X1	bits 9-23	effective address

The debug operations are

0174	TN	(Trap Enter)
0175	TX	(Trap Exit)
0176	SSP	(Snap Shot Print)
0177	SST	(Snap Shot Type)

The following global locations must be defined in the compiler:

2LEVEL	Level number
BASE	Table of list bases
TOP	Table of list tops

Table 2
Compiler Interfaces and Intra Global Symbols (cont)

BOTTOM	Table of list bottoms
START	Table of list starts
INTERP	Entrance to interpreter

Intra Global Symbols

SPACE
PAGE
PLINE
ICPI
DCPI
STC
ICPSUB
SCPSUB
OOSUB
OASUB
ODSUB
ILINE
LINE
PRTY
SYMTAB
SYMTB
SYMTC
ERRORSUB
BL
NL
NDCT
NDCP
DUMPM
DUMSTE
HISTOR
THRLEV
HP
2LOC
RELREG
REGX1
REGX2
REGX3
NEXTRQ
XAHP
XNTERM

READER COMMENT FORM

XEROX

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

- ☐ Learning
- ☐ Installing
- ☐ Operating
- ☐ Reference
- ☐ Maintaining
- ☐ Sales

Is the material presented effectively?

- ☐ Fully covered
- ☐ Well illustrated
- ☐ Clear
- ☐ Well organized

What is your overall rating of this publication?

- ☐ Very good
- ☐ Fair
- ☐ Very poor
- ☐ Good
- ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the XDS Software Improvement or Difficulty Report (1188) instead of this form.

Thank you for your interest.

Fold and fasten as shown on back.
No postage needed if mailed in U.S.A.

Your name and return address.

2190(5/71) Xerox

STAPLE

STAPLE

FOLD

FIRST CLASS
PERMIT NO. 229
EL SEGUNDO, CALIF.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

XEROX

701 South Aviation Boulevard
El Segundo, California 90245

ATTN: PROGRAMMING PUBLICATIONS

CUT ALONG LINE

FOLD