160-A FORTRAN MAINTENANCE MANUAL

CONTROL DATA CORPORATION
3330 Hillview Avenue
Palo Alto, California

November 1963

Pub. No. M0503

Section 1.1

General Description

INTRODUCTION

This introduction will describe the general construction of the entire 160A FORTRAN System.

The compiler, distinguished from the interpreter, is composed of two passes. In addition, there is a preliminary section which allows for changes from the standard compiling procedure. Specifically, routines which bring in the source code and write the memory map may be changed. This preliminary loads Pass I, the principal part of the compiler. Finally, the source input routine is loaded.

Pass I scans source statements one at a time. When a legal statement is read, the compiler generates object code in final form if possible. In some cases, for instance transfers, only relative addresses can be known at this time. As soon as the compiler encounters an illegal statement, all previously generated code is destroyed and a diagnostic code is output to the scratch tape. While no further object code is output to the scratch tape, the compiler continues to process each statement for further diagnostics.

Pass II, the second part of the compiler, is essentially a loading and relocating pass. When Pass II begins, the compiler has available all of the information it needs to lay out core for execution. Specifically it knows which interpreter modules and functions (I/O and library) it will need. A memory map is prepared giving all object code locations of variables, constants, statement labels, and subprograms. All identifier information necessary for relocation is saved in condensed form. The object code is then loaded and all relocatable addresses are updated. Finally the interpreter and its modules are loaded. If diagnostics were generated, these last three steps are bypassed and, instead, the diagnostic information is output and the system halts.

The interpreter consist of a main fixed part and modules for floating point arithmetic, Boolean arithmetic, and FORMAT control. These modules are loaded only if required by the source program. At the end of Pass II, control is given to the interpreter and execution begins. Interpretive object code instructions are executed sequentially until an interpretive control instruction changes the sequence. Transfers to library functions and input-output drivers are effected through a transfer vector resident in the interpreter.

MEMORY LAYOUT DURING COMPILATION and EXECUTION

During Pass I, Bank 1 contains working storage and the processors for most statements. Bank 0 contains the input/output buffer, the source code input routine, the system input routine, the algebraic scan and various utility routines. As statements are processed, a symbol table, called Idlist, is formed starting downward from the highest numbered location in memory. This table records all statement labels, variables, constants and other identifiers encountered by the compiler. The area in bank 1 starting from the end of the processors and ending with the beginning of Idlist (or, in a three or more bank system, the end of bank 1 if Idlist is not in bank 1) is available for processing statements. This area is increased if there are no EQUIVALENCE statements and is further increased if no COMMON declaration is made. It holds lists for data declarations (SUBROUTINE parameters, COMMON, DIMENSION, and EQUIVALENCE statements) until they are all processed for a given routine. The effective area is decreased, temporarily, during the processing of nested DO-loops.

During Pass II, bank 0 contains the coding for Pass II and a condensation of the IDLIST in the locations which will ultimately be occupied by the Interpreter and library routines. Object code is loaded into its execution locations, all relocatable addresses are fixed, and, finally, the Interpreter and library functions are loaded over Pass II.

The interpreter memory is organized as follows: In Bank 0, beginning at location 0000, are low core, the library function transfer vector, the I/O buffer, the fixed interpreter, the floating interpreter, the format interpreter, the boolean interpreter, (these last three only as needed), the required I/O routines, a routine called INPUT which does input conversion, library functions and object code. Data storage is assigned by the compiler from the end of the last available bank. Between the end of the object code and this data is an area called erasable storage. The data assignment is in two parts: COMMON and variables local to subroutines. The first four cells assigned are for a variable called IO and the constant "1". IO is needed only when array I/O by name is used, but this variable is nevertheless always assigned.

Two points might be mentioned at this time. First, arrays are stored forward in core, i.e. x(1) has a smaller location number than x(2). Secondly,

all functions and I/O routines which end up in Bank 0 will be loaded beginning at an even cell while those in Bank 2 will begin at an odd. An integer constant or variable requires two 160-A words, floating point requires three. The area between the object code and the data storage is called "erasable". Room is needed here for linkage between subroutines and functions. If a function is used, space for a minimum of 32 erasable locations is allocated by the compiler. If only subroutines are used, one erasable location will be set aside for each subroutine and another for each formal parameter used. Each erasable location requires three 160-A words. Since no flow analysis is done by the compiler, room will be set aside for the worst case. One or two locations are set aside to insure that in the first bank occupied by erasable, the number of locations available is divisible by three.

Schematically, the memory layout, during compilation, Pass I and Pass II and execution, is as follows for n + 1 banks of memory:

0,0000	1,0000	...	n,7777
(Compiler Pass 1	)(Processing)(	Idlist	)
(Compiler PassII)(Condensed Idlist	)(Object Code)(Unused)	(Data + COMMON)	
(Interpreter+modules)(Library Routines)	(Object Code)(Erasable)	(Data + COMMON)	

COMPILER PASS I

The coding for Pass I occupies all of Bank 0 and 3/4 of Bank 1. Statements are brought in one at a time, blanks removed (except for FORMAT statements) and moved into Bank 1. If the statement has a label, this identifier is entered in the symbol table called Idlist. As the statement is cracked, all new variables, constants, etc., are entered in the Idlist. If this list gets too large, i.e., if it is impossible to move a statement into Bank I and to work on that statement, compilation cannot continue.

All items are entered in the Idlist as they are encountered together rather than in separate tables. The entry includes a notation as to the length of the entry. This depends on two factors: The type of entry and the length of the identifier. As variables and formats are encountered their object code locations may be determined. Only transfer points and internal stores are undefined. A 2-character name or statement label requires one word, a three or four character identifier, two words, and a five or six character identifier, three words. Since formats and statement labels are different types, they may have the same number. Similarly, the name used in an assigned GO TO statement may be the same as a variable. A level number is assigned to the subroutine according to the order of its occurrence. All entries into the list are identified by level; the main program may or may not be level 1. Constants and the variable I/O described above are assigned to level 0 and may be referenced by any subroutine. All other identifiers except subroutine and function names are local to a subroutine.

Fixed point quantities are represented by two words. The high order bit in each word is for the sign. Boolean constants also occupy two words and once the compiler has formed such a constant, it is indistinguishable from an integer constant. Floating point quantities occupy three words. They are stored by the

compiler in BCK (See STM 160-A-17). In the QWIK system these BCK numbers are converted to a 36-bit binary quantity whose format is the same as a 7090 word.

As mentioned, the symbol table may overflow. This may occur in two ways. First, any given statement which is processed may be too large to be cracked. The compiler will quit on this statement but attempts to go on to the remaining statements. However, if it cannot move the statement into working storage, the compiler stops. It is still possible, however, to obtain a memory map.

In Pass I, statements are scanned using the following procedure. All comment cards are skipped. The first statement entered is checked for numerical COMMON. If present, a bound is set as described in the common processor below. The section of code used for this checking and other initialization is then released to working storage. It is necessary that the compiler read ahead one card so that it may check for continuation cards. As soon as a statement is isolated, it is checked to see if it is FORMAT. If it is, Idlist is checked to insure that it contains an entry for this FORMAT statement. The relative object code location for the statement is computed and stored in the entry. Note that it is possible that this statement number may have already occurred and is present in the Idlist as an unassigned quantity. It may also be doubly defined; this causes a diagnostic.

To determine whether the non-format statement is an algebraic statement, it is scanned for an equal sign. Only one other kind of statement has an equal sign, the "DO" statement, but it also must have commas outside of parentheses. Having identified an algebraic statement, three lists are created. First, a Operand-Operator list, MATHST. All operand references in this list point to relative locations in a second list. This second list BOPLST, contains the object code locations for all operands in the statement. The final list, BOXLST, contains indexing information. The algebraic translator then generates the appropriate sequence of one-, two-, or three-word commands. Ordinary arithmetic commands

referencing memory require two words; references to formal parameters or pseudo-accumulators used by the interpreter for temporary storage ordinarily are one word instructions. A subscript expression is analyzed into a fixed part and a part depending on variables. The fixed part is compiled into the command and the variable part is converted into an indexing expression. Rather than calculate the value of an index each time one is used, the compiler sets up a pseudo index register (pseudo B-box) for each subscript expression. For each integer variable appearing in subscript expressions, an UP subroutine is created which describes its position in each of the b-boxes in which it appears. The UP routine also contains the last value of the variable. When a value is assigned to the variable, an UP command is executed which causes each pseudo b-box it references to be modified according to the increment.

If not an algebraic statement, a scan is made of the first two letters. In many cases the first two letters and length are sufficient to define the statement given that it is a legal FORTRAN statement. This does lead to some anomalies. For instance PANTS become equivalent to PAUSE. Several of the statement types (GO TO, IF, etc.) refer to statement labels which may not as yet be defined. An Idlist entry will be made for these unassigned labels and when they are encountered, the list will be updated with their object code locations.

The SUBROUTINE statement, if present, is processed first. If there are array formal parameters, the DIMENSION statement must be in core at the same time. The formal parameters will be processed with references to the DIMENSION statement. The COMMON statement, which must be next after the SUBROUTINE statement, also requires that the DIMENSION statement be in core. Finally, the EQUIVALENCE statement, which follows the DIMENSION statement, is processed referring to the DIMENSION statement. All variables in the DIMENSION statement which have not yet been processed will be placed in the Idlist. It is possible to have more than one DIMENSION statement but the variables in this statement must be local and

must not appear in the EQUIVALENCE statement. Since the DIMENSION statement triggers the indexing, it is necessary that each program have a DIMENSION statement (if there is going to be one) before any integer variables are encountered. When the first COMMON statement in the source program is processed, a bound for the COMMON area is computed. Any subsequent COMMON statement requiring more storage area will cause a diagnostic to be produced.

For many of the non-algebraic FORTRAN statements, ASSIGN, computed GO TO, etc., the compiler generates macro calls to the interpreter. In executing an I/O statement, the interpreter transfers control to an input/output macro. The list has been compiled as a series of loads for output and series of stores for input. The I/O macro processes the information in the list according to the format statement, transferring control to the actual I/O routine when required. In order to process the list, the normal linkage within the interpreter has been modified. For this reason, at the termination of the I/O statement, a macro called IOT is called to restore the interpreter. In implementing the "DO" loop, a count of the number of times through the loop is calculated. This count is a negative quantity and is defined as (M2-M1)/M3, for the statement DO N I= M1, M2, M3. This count and the increment may not be changing during execution of the loop. This means that while M3 may be changed, it will not effect the increment. The compiler will not allow the increment to be a negative constant, but the interpreter does allow a variable M3 to take on negative values. Initialization of the loop is accomplished by interpretive arithmetic coding; incrementation is done by a macro.

COMPILER PASS I ROUTINES and STORAGE AREAS

BANK 0

Low Core

OUTBUF: Output buffer for compiled code

BINARY: Binary tape routine

INBUFF: Input buffer for a source record

READST: BCD source input routine

I/O TBL: Table of input output equipment names.

PRIMER: Read the first source card, go to (BEGN) in bank 1

NXTSTT: Read next complete statement, check to see if it is FORMAT, move it into processing area in bank 1 (eliminating blanks unless FORMAT). If statement is too large to fit in processing area, halt. If statement is FORMAT, set flag and exit to FORMAT in bank 1. Otherwise, go to NEWSTT.

NEWSTT: Process statement number if present. Test to see if statement contains an equal sign and does not contain any commas outside parentheses. If so, exit to algebraic processor at PREALG. Otherwise type statement according to the first two letters and exit to the appropriate processor in bank 1.

L(CLAS: Classify character in A according blank (0), special character (1), number (2), letter I through N (3), other letter (4).

FORMOP: Classify identifier by first character. If blank, then error unless at the end at the statement being processed. If special character, store translation into algebra string, put 0 in A and exit. If number, isolate complete constant, convert it from BCD to binary, type it as integer or floating, ensure that it is in IDLIST, and exit with type in A. If letter, set fixed or floating type. Pack characters into name until special character is encountered. If this character is a left parenthesis, then either an array IDLIST entry has been made for this identifier or the previous character is an F and this is a library function or else there is an error. For any legtimate identifier ensure that it is in IDLIST, and exit with type in A.

LIB-FN: Search IDLIST for previous occurence of named library function. If found, exit. If not, increase library function counter, make IDLIST entry and exit with function number in A.

PACKID: Pack identifier, two characters per word appending final blank if necessary, into cells IDEN through (LETEND). Compute number at words required, exit.

LSTSPY: Check IDLIST for previous occurence at the proper level of the identifier in IDEN through (LETEND). If found, exit with IDLIST type in A; if not, exit with zero.

MAKEID: Create data address and IDLIST preamble for simple variable or constant. Jump to NTRID to enter identifier. Exit with list type A.

NTRID: Enter identifier (including preamble) into next available IDLIST locations.

SUBSCR: Analyze subscript expression. Create constant additive and index expression. If the subscript is not constant, ensure that proper pseudo b-box entry is in idlist.

ALGSTR: Refine algebraic expression. At entry, expression consists of the BCD source input string. Jumps are made to routines to isolate and translate identifiers and operators and to form subscript expressions. At exit, the expression has been replaced by three lists to be used by the algebraic translator. The first, MATHST, consists of a one word-per-element translation of the original expression. For operand description it points to the second list, BOPLST. This contains object code location or formal parameter position for each of the designated operands, and, in the case of variable subscript expressions, also contains the constant part of the expression and a pointer to the third list, BOXLST, which contains IDLIST locations for the appropriate pseudo b-boxes.

STNOCK: Set beginning and find end of a statement number, set list type to 13, and jump to LBLCHK.

STNOID: Pass through leading zeros in a statement number; jump to STNOCK.

LBLCHK: Pack statement number, search IDLIST for it. If not found, create preamble and jump to NTRID to enter identifier.

MAKCOM: Set flags so that proper command will be made. (This is used principally to make store commands in algebraic replacement statements and to generate loads and stores during I/O list processing). Put the command away in OUTBUF and exit.

MAKSTO: Jump to MAKCOM with store designator in A. Save object code location for possible DO-loop incrementation. If store was of an integer variable and there are dimensioned variables, generate an UP command.

PREALG: Process arithmetic replacement statements. Jump to ALGSTR to refine algebraic string, then to ALGBRA to translate the algebraic expression, and finally to MAKSTO to make the appropriate store command.

STATNO: Check first non-blank character found in columns 1-5 of a source statement. It must be B for Boolean, O for octal, S for symbolic, or a number for the first digit in a statement label. Insure that it is in IDLIST, but has not previously been assigned a relative object code location. Zero out L(CURR, used by diagnostics processor, which counts number of statements since last label.

TILT: First time through rewind tape 2. Create a diagnostic message and dump it onto tape 2.

VNUMCN: Convert numbers from BCD to fixed or floating binary representation. TILT if magnitude is out of range.

(FMBK1: Effect a pseudo jump return from bank 1 to bank 0. Routine contains an ordered list of all bank 0 routines jumped to from bank 0. List position is in (TOBK0; the specific call is used to transfer the contents of A.

VADXT: Integer add and subtract routine. Operands are in OPJ and ACCJ at entry, result is found in ACCJ. Sign of A on entry determines which operation is to be performed.

VMLTIN: Integer multiply routine. Operands are in OPJ and ACCJ at entry, result goes to ACCJ.

ALGEBRA: Translate an algebraic expression. Put the coding away into OUTBUF. See ALBEGRAIC TRANSLATOR for details.

I = J,K: Generate coding for DO-loop preamble or DO-implied I/O list. Exit to SVCODL in MAKSTO where UP commands are generated as needed.

FORMOT: Output a FORMAT statement after dumping OUTBUF if there was anything in it.

BNK1VC: Exit to routine in bank 1 which branches to various processors depending on contents of A.

PUTWAY: Put one word into object code buffer (OUTBUF). When it is filled, dump it out on tape 2.

FINISH: Save flags, load Pass II and jump to it.

BANK 1

B(Z), A(Z): Relative object code location immediately following last store generated. This is used by DO-loop processor as the location to jump to after incrementation.

(TOBKO: Routine used in conjunction with (FMBK1 in bank 0 to effect a pseudo jump return from bank 1 to bank 0.

CMNVEC: Distributor for routines in bank 1 jumped to from bank 0.

FORMAT: Pack FORMAT statement 2 characters per word. Ensure that label is in IDLIST and has proper object code address. Go to FORMOT in bank 0 to output the statement.

L(ST): Generate coding fro STOP n instruction by jumping to (PAUSN to generate a PAUSE n instruction and following this by a "transfer back one location" instruction.

L(PA): Jump to (PAUSN to create a PAUSE n instruction and put it away.

L(FU): Give a diagnostic if a FUNCTION subprogram is encountered.

(PAUSN: Isolate octal number n and generate an HPR n instruction

L(D): Process a "DO N I = M1, M2, M3" instruction. Jump to STNOID to process statement number N. Refine the rest of the statement using ALGSTR. Jump to INCSAV to save incrementation information.

INCSAV: Jump to I = J,K to generate coding for DO-loop preamble. Save all information necessary for DO-loop incrementation in the first locations of the processing area. Move location of beginning of area, L(PROC, to reflect this.

L(CO): If next character is M, branch to COMMON processor, L(COM). Otherwise it must be CONTINUE statement.

L(CON): Process CONTINUE statement. If statement is numbered, check to see if it is a terminal statement of a DO-loop. If so, make incrementation coding by jumping to MAKINC.

COEXIT: Normal exit from processors. Increase increment from last label and exit to NXTSTT in bank 0 to read and process next statement.

L(TILT: Exit from bank 1 routines to diagnostic routine, TILT, in bank 0.

L(GO): Generate coding for ordinary GO TO and for computed and assigned GO TO statements. If assigned GO TO, label is terminated either by blank or comma. If computed GO TO, generate macro which contains number of branches, the location of the control variable, and, for each of the branches, two words giving the bank and location of the IDLIST entry for the branch number.

MAKINC: Generate incrementation macro for DO-loops and for DO-implied I/O lists. Restore the beginning of the processing area, L(PROC, to the beginning of the beginning of the incrementation information.

EQUPNO: Find library number for I/O equipment referenced. If it is a standard I/O call, find the name in I/OTBL. If not, isolate the name and pack the identifier. In any event, jump to LIB-FN to obtain library vector number and exit with it in A.

RELESE: Release processing area which has been used temporarily to hold formal parameter, DIMENSION, and COMMON lists until they have been processed. If there are no EQUIVALENCEs, make the space used for this processor available; if in addition COMMON is not used, free this space also.

NXTSPG: Prepare for next subprogram by increasing LEVEL count and initializing various switches. Make an entry for a statement numbered 0 at this level and exit to NXTSTT to process next statement.

L(EN): If statement is END FILE, branch to that processor. Otherwise, process END statement. If a subroutine is just compiled and no RETURN was encountered, generate a RETURN macro. If a main program and the last statement was not STOP, generate a STOP 0 command. In any event, check to see if the next statement to be processed is all blank. If not, go to NXTSPG. If so, exit to FINISH.

OUTPUT: Generate output macro, set I-OR-O to O.

INPUT: Generate input macro, set I-OR-O to I.

L(PU): Set NXTTYP to standard punch. Jump to OUTPUT. If PUNCH FLEX set parameter and merge with RDFLEX. If not, set A to NXTTYP designator and merge with READCD.

L(PR): Set A to standard printer and merge with L(PU).

L(WR): Create output macro. IF WRITE OUTPUT TAPE, set A to 15 if standard equipment is being used or to zero for non-standard and merge with TAPEIO. IF WRITE TYPE, set parameter and merge with RDFLEX. Otherwise set binary tape function write, load A with equipment number, and merge with FM(OUT.

REWIND: Set tape function rewind, load A with equipment number, and merge with FM(OUT.

ENDFIL: Generate output macro, set tape function end file, load A with equipment number, and merge with FM(OUT.

L(BA): Generate input macro, set tape function bakcspace, load A with equipment number, and merge with FM(OUT.

L(RE): If READ or REWIND, exit to READ. Otherwise, generate a return macro if in a subroutine.

READ: Generate input macro. If REWIND, jumpt to REWIND. If READ FLEX, exit to RDFLEX. If READ INPUT TAPE, exit to BCDTPI. If READ TYPE, exit to RDTYPE. If READ n or READ (, exit to READCD. Otherwise, set binary tape function read, load A with equipment number and exit to FM(OUT.

FM(OUT: Create appropriate coding for binary tape function on given tape. Exit to list processor.

RDTYPE: Set type parameter, get library vector number, and merge with RDLABL.

RDFLEX: Set flex parameter, get library vector number, and merge with RDLABL.

BCDTPI: Check for alphabetic READ label. If so, exit to READCD. Otherwise, set A to 12 if standard equipment is being used or to zero if nonstandard and merge with TAPEIO.

TAPEIO: Get library vector and put it and tape number into equipment description. Merge with RDLABL.

READCD: Get library vector number and put it into equipment description. Merge with RDLABL.

RDLABL: Isolate format label and ensure that it is in IDLIST. If alphabetic label, make certain that object code address is in FORMAT label entry. Exit to LIST.

LIST: Put away first two words of I/O macro which describe equipment, driver, and location of FORMAT statement or binary tape function. If there is no list, exit to IO-END. Otherwise generate loads if output or stores if input for the elements in the list. If parentheses are encountered, generate DO-loop initialization and incrementation coding. If an unindexed array name is found, exit to ARYNAM. When list is exhausted, exit to IO-END.

IO-END: Generate macro for I/O END.

L(IF): Generate macros for IF(SENSE SWITCH i), IF ACCUMULATOR OVERFLOW, IF QUOTIENT OVERFLOW, or IF DIVIDE CHECK. Use ALGSTR and ALGBRA to generate coding for the expression in an arithmetic IF, then generate the macro. Use a common routine for putting out the branch locations.

(TOPTO: Routine used by many of the processors in bank 1 to shorten coding necessary to get to the PUTWAY routine in bank 0.

(OCTAL: Pack four octal digits into a word and put it away as the next word of object code.

L(AS): Generate coding for ASSIGN n TO i statement.

STTYPE: Check next statement for type. Exit with parameter in A: 1 for COMMON, 2 for DIMENSION, 3 for EQUIVALENCE, 0 for anything else.

NEWDAT: Create new end of data address by reducing DATEND by the number of words in NOTINT.

NTRDIM: Make IDLIST entry for array.

L(DI): Process DIMENSION statement. Isolate identifiers and subscript maxima, compute number of elements in the arrays, and pack this information at the beginning of the processing area. If there is an unprocessed COMMON list, jump to COMNID to process it. If there are SUBROUTINE parameters, jump to SUBRID to process them. If the next statement to be processed is EQUIVALENCE, exit without completion of array processing. If not, use NTRDIM to make appropriate IDLIST entries. Exit to RESTOR beginning of processing area.

L(SU): Process SUBROUTINE statement. Ensure that subroutine name is in IDLIST and give it the proper relative object code location. Create a packed list of parameter names at the beginning of the processing area. If the next statement to be processed is COMMON or DIMENSION, exit without processing the parameters. If not, jump to SUBRID to process the parameters, release the processing area and exit.

NTRVBL: Create IDLIST entries for simple variables in COMMON, EQUIVALENCE, or SUBROUTINE statements. Use NTRID to make the entries. Make U? commands for integer variables if there are arrays in this subprogram.

SUBRID: Process list of formal parameters for SUBROUTINE. Count parameters and jump to MATCHK to effect appropriate entry.

MATCHK: Check next identifier to see if it is in the unprocessed array list. If so, set type for COMMON or parameter and jump to NTRDIM to make an IDLIST entry. If not, jump to NTRVBL to make simple variable entry.

L(CA): Process CALL statement. Generate coding for evaluating actual parameters given as arithmetic expressions, for loading locations of first element of an array if its name is given, or for loading the value of a simple variable. In each case, generate a store into the next available erasable locations. Generate the return jump coding necessary for jumping to the subroutine. For any simple variable actual parameter, generate a load of the appropriate erasable location and a store into the proper memory location using MAKSTO.

NAMPRM: Used by L(CA) to isolate the next actual parameter and refine it using ALGSTR.

ARYNAM: Used by LIST when it encounters an unindexed array name in an I/O list. Coding is created as though the array were indexed by a variable called IO which takes on values 1 through the total number of elements in the array. I/ODO is used to generate initialization coding, a dummy list is entered into the processing area and an exit is taken back to LIST.

L(COM): Process COMMON statement. Pack the COMMON identifiers into a list starting at the beginning of the processing area. If the next statement to be processed is DIMENSION, exit without processing the COMMON list. Otherwise, jump to COMNID to do the processing. If there are subroutine parameters, jump to process them.

COMNID: Process COMMON list. If this is the first COMMON statement, the length for COMMON is computed. If not a check is made to make certain that the original COMMON bound has not been exceeded. As each name in the list is isolated, jump to MATCHK to make the appropriate IDLIST entry.

EQNUM: Used by equivalence processor to compute object time addresses. If the identifier is an element of an array, the address is increased or decreased by the element position according to whether the identifier is the left or right half of an equivalence pair.

L(EQ): Process EQUIVALENCE statement. Isolate identifiers, use EQNUM to compute object code addresses, and use MATCHK to make appropriate IDLIST entries. If there are unprocessed array names, process them.

(BEGN): This is the master initialization routine and its space is given up to the processing area as soon as it is used. Variable object code and IDLIST addresses depending on the number of banks in the object and compiling computers are set. If the first non-comment card is a numeric COMMON card, the COMMON bounds are set at this time. Exit is to NXTSPG.

TABLE OF CONTENTS

SECTION 1.2

Specifications for BINARY

BINARY is a closed subroutine to do all binary (odd parity) tape handling operations required within the compiler. All bank settings must be zero upon execution of the routine. BINARY is included in the initialization routines. The two BINARY routines are used as follows: BIN for 163 magnetic tapes or equivalent. BIN7 for 1607 magnetic tapes.

The routine will be entered by a JPR to the first location with A containing a parameter requesting these functions:

A = 0 - write tape 2 from 100 to 200

A = 1 - read tape 2 into 100 to 220

A = 2 - write an end of file record on tape 2

A = 3 - rewind tape 2

A = 4 - rewind tape 1

A = 7777 - search forward for an end of file record on tape 1

A = address (anything else) - load one record from tape 1 beginning at the address in A upon entry to the routine.

Error conditions will be handled as follows:

Entry parameter = 0

End of tape - stop. If restarted, return

Parity error - try to rewrite the record three times

If error persists, skip tape and try three more times.

If error persists, stop. There is no provision for recovery.

Entry parameter = 1 or "address" -

End of tape - stop. If restarted, return.

End of file - return with A negative.

Parity error - try three times then stop. There is no provision for recovery.

Entry parameter = 2, 3, or 4

No error checking is done.

The normal return for the functions is to the address provided by the JPR instruction with A set as follows:

0 - A = 0

1 - A = 0

2 - A ≠ 0

3 - A ≠ 0

4 - A ≠ 0

7777 - A ≠ 0

Address- A = 0

A bootstrap loader will be included as a bioctal tape. This loader sets all banks to zero, rewinds both tapes, reads the first record from the systems tape (tape 1) and transfers control to the initialization routine that was read. As explained in the write up of the initialization routines, the contents of the A-register has meaning to this loader

Procedure: BIN

1. Enter by JPR with A = entry parameter.

   Set START = entry parameter.

   Entry parameter for tape 1 operation? No, go to 3.

2. Create codes for tape 1 operations. Go to 4.

3. Create codes for tape 2 operations.

4. Select odd parity (binary mode).

   Entry param = 0? Yes, go to 21.

5. Entry param = 2? Yes, go to 22.

6. Entry param = 3? Yes, go to 23.

7. Entry param = 4? Yes, go to 25.

8. Entry param = 7777? Yes, go to 17.

9. Set to bypass write end of file error procedure.

   Set read code in execution sequence.

   Tape 1 operation? Yes, go to 12.

10. Set buffer limits 100 to 220.

11. Create INP or OUT instruction and store in execution sequence.

12. Execute preset sequence.

    Check status. Are there errors? No, (A = 0) go to 18.

13. End of tape response? Yes, go to 24.

14. End of file response? Yes, (A non zero) go to 18.

15. Is this the third parity error on this operation? Yes, go to 19.

16. Backspace, go to 12.

17. Search forward for end of file on tape 1. (A non zero).

18. Return.

19. Is this a read operation or the second try on a write operation? Yes, go to 24.

20. Reset counters, backspace, write end of file, go to 16.

21. Set write code in execution sequence. Preset INSTR to create OUT instruction. Go to 10.

22. Write end of file on tape 2. (A non zero). Go to 18.

23. Rewind tape 2. (A non zero). Go to 18.

24. Stop. If restarted, go to 18.

25. Rewind tape 1. (A non zero). Go to 18.

Procedure: BIN7

1. Enter by JPR with A containing the entry parameter. Set exit address and initialize counters. Create select codes for the requested tape. Select the tape first as a write tape then as a read tape, tape 1 cannot be tested for write ready. Generate mask. Wait for the selected tape to be ready. Was write tape 2 operation requested? Yes, go to 15.
2. Was the read tape 2 operation requested? Yes, go to 10.
3. Was the write end of file operation requested? Yes, go to 14.
4. Was the rewind tape 1 or 2 operation requested? Yes, go to 11.
5. Was the search forward for end of file operation requested? Yes, go to 12.
6. Read the selected tape into the area specified by the entry parameter. Wait ready. Was an end of file record read? Yes, set A = 10 and go to 19.
7. Are there parity errors on the operation? No, go to 19 (A = 0).
8. Is this the third consecutive error? Yes, go to 18.
9. Backspace the selected tape. Wait ready. Go to 6.
10. Set entrance parameters to read into locations 100 to 220. Go to 6.
11. Rewind the selected tape. Go to 19 (A $\neq$ 0).
12. Read one record from the selected tape and wait ready. Was this an end of file record? Yes, go to 19 (A $\neq$ 0).
13. Go to 12.

14. Write end of file record on selected tape. Wait ready. Go to 19. (A = 0).

15. Write tape 2 from 100 to 220 and wait ready. Were there parity errors on this operation? No, go to 19 (A = 0).

16. Backspace the selected tape and wait ready. Is this the third consecutive error? No, go to 15.

17. Write end of file record, wait ready, reset counters. Is this the third consecutive error sequence? No, go to 16.

18. Stop. If restarted, go to 19 with A = 0 unless altered while the program was halted.

19. Return.

The READ Statement routine is a closed subroutine, entered by a JPR to the first location, which will read one 72 decimal character input record. All bank settings must be zero when the routine is entered. If necessary, the input codes are converted to BCD codes, and are stored (one character per word) from INBUFF to INBUFF + 72. The contents of INBUFF +72 will always equal a BCD blank. INBUFF is a symbolic location within the compiler that is the first location of the input buffer.

READST requires no parameters. There are nine routines for statement input, which may be used interchangeably. They are in binary format on the systems tape without indentifiers. They require the following equipment and are named as follows:

Paper tape (flex)	FLEXIN
088 card reader	RDCARD
167 card reader	RD167 or DBH167
167 card reader, source list on 166 printer	RD167CS
163 magnetic tape unit (tape 3) or equivalent	RD163C
1607 magnetic tape unit (tape 3)	RD1607C
405 card reader	DB405C
167 card reader, source list on 1612 printer	RD167CP

If a normal record has been read, the normal return is to the location provided by the JPR instruction

Procedure: FLEXIN

1. Enter by JPR. Initialize and load the buffer with BCD blanks. Select paper tape reader.
2. Set case to lower.
3. One character to A.
4. Is it a case code? Yes, set case flag to proper case and go to 3.
5. Is character a backspace or delete code? Yes, go to 3.
6. Is character a tab code? Yes, begin storing at sixth location in buffer. Go to 3.
7. Is it a carriage return? Yes, return (A = 0).
8. Convert flex code to BCD. Store converted character. Increase counter. Have 72 characters been stored? No, go to 3.
9. Read to a carriage return. Return (A = 0).

Procedure: RD167C - RD167CS - RD167CP

1. Enter by JPR.
2. Wait for card reader ready. Set return address.
3. Select single cycle read.
4. Input two columns. Check for reader failure. Failure, display status and halt. Clear A-reg. and run to continue (for amp. failure replace last card in hopper).
5. Convert and store the two columns in buffer. Set LWA + 1 of buffer to BCD blank.
6. Read one column.
7. Convert to BCD and store lower 6 bits in buffer.
8. Have 62 columns been read? No, go to 6.
9. Lock out timing fault. Return (RD167C). RD167CS go to 10. RD167CP go to 13.
10. Pack buffer. Wait ready on 166 printer.
11. Output packed buffer on 166.
12. Restore buffer (unpack) and return.
13. Wait ready on 1612 printer.
14. Output buffer on 1612 and return.

Procedure: (RDCARD)

1. Enter by JPR.
2. Wait for card reader ready. Select primary read. Initiate buffer input. Initialize to convert columns 1-36.
3. Wait for current row to be completely read.
4. Set COLUMN to address of column 1 or 37.
5. Set WORDER to word to be converted.
6. Is word negative? Yes, go to 12.
7. Set A = 0.
8. Store in buffer (for none row) or add to buffer (for succeeding rows). Is this the end of the row being converted? Yes, go to 15.
9. Is this the last bit of the word being converted? Yes, go to 14.
10. Shift the word being converted one place to the left. Go to 6.
11. Is the value of the current row non zero? Yes, go to 8.
12. Is the previous value of the column equal to zero? Yes, set A=12. Go to 8.
13. Set A = 20. Go to 8.
14. Increase address of word to be converted. Go to 5.
15. Skip columns 37-84 if first half. Reset address of word to be converted; reset 13 counter. Is row value 40? Yes, go to 18.
16. Set STORER to RAI if not already set. Subtract one from row value. Is resulting row value positive? Yes, go to 3.
17. Set row value = minus (40). Go to 3.

18. Is row value = 60? Yes, reset column address to column 1 or 37. Set all zero value columns to 20. (BCD blank). Go to 20.

19. Set current row value = 60. Go to 3.

20. Has the second half of the card been converted? Yes, go to 22.

21. Reset to convert columns 37-72. Transfer remaining image to the end of the buffer area. Set 73rd character = blank. Go to 4.

22. Return.

Procedure: RD163C

1. Enter by a JPR.
2. Initialize, select even parity (BCD).
3. Set 73rd character = BCD blank.
4. Read tape 3 (6 bit) from Inbuff to Inbuff + 110.
5. Status response = 0? Yes, go to 10 (A = 0).
6. End of file? Yes, go to 2.

6a End of tape? Yes, go to 11.

7. Backspace. Is this the third try? Yes, go to 9.
8. Go to 4.
9. Is this second time? Yes, halt. Go to 10.

9a No, backspace three records or to load point and reposition tape. Go to 4. There is no provision for recovery.

10. Return.
11. Rewind unload and halt. Run if A was cleared. Go to 2. Otherwise go to 10. (A non zero)

Procedure: R1607C

1. Enter by JPR. Initialize counters, set exit address, select read tape 3 and wait for read tape ready.
2. Read one record and wait for tape ready. Are there tape errors? Yes, go to 5.
3. Unpack and store characters in buffer area. Set A = 0.
4. Return.
5. Is the end of file indicator set? Yes, go to 2.

5a End of tape? Yes, rewind and halt. Run, if A was cleared,go to 2. Otherwise, go to 4.

6. Backspace tape and wait ready. Is this the third error? No, go to 2.
7. Stop. If restarted, go to 4.
8. Set A non zero. Go to 3

Procedure: DB405C - DBH167

1. Enter by JPR.
2. Set return address and counter.
3. Is a card image waiting in buffer? Yes, go to 12.
4. Wait ready. JPR to 5.
5. Set up buffer entrance and exit registers.
6. Check status. Zero? Yes, go to 11.
7. Check for hopper empty. Yes, go to 14.
8. Reader failure? Yes, go to 9, in the case of DB405C also get card to secondary hopper.
9. Halt with status + 1000 in A register.
10. Select single cycle read. Go to 6.
11. Select single cycle read. Set buffer bank control to zero. Initiate buffered input. Return to main routine.
12. Set flag for no card image waiting in buffer. JPR to 5.
13. Reset flag for card image waiting in buffer.
14. Unpack card image and return.

VARIABLE AND CONSTANT STORAGE

Generally, each floating point variable or constant requires three words of storage; each fixed point variable or constant requires two. If an array has dimensions D1, D2, D3 stated in a DIMENSION statement, then it will occupy D1*D2*D3 "words" of storage where a "word" is 2 or 3 words in the 160-A depending on the mode.

Level:

If a main program is thought of as a special type of subprogram, then associated with each subprogram is its order in the source deck. This will be referred to as the "level of the subprogram". Since the same variable name can be used in different subprograms to refer to different quantities, each name will be further identified by giving it the level of its subprogram. Constant will also be given level zero, and the same constant (NOT statement number) occurring in two programs will cause only one storage assignment for the value. Constant which have the same value but different representations will similarly be identified.

Variable names for formal parameters in SUBROUTINE subprograms will cause no reservation of storage space even if these names also occur in a DIMENSION statement in the subprogram.

It is possible to reuse storage space a program through the use of EQUIVALENCE statement. Formal parameters may not appear in EQUIVALENCE statements. Two or more programs can refer to the same variables (without including them in a parameter list) by placing them in a COMMON statement.

STORAGE REQUIREMENTS FOR STATEMENTS

SYMBOLS:

Let H be an address in a (up to 8-bank) 160-A then b(H), 3 bits, is the number of the bank and A(H), 12 bits, is the location within the bank. Generally for H a fifteen bit integer, B(H) represents the upper 3 bits and A(H) the lower 12.

MACRO OPERATIONS:

CMPUTD: List following transfer contains information for computed GO TO.

IF: Next 4 words contain 3 conditional transfer addresses depending on value in the accumulator.

IFOV: Next 3 words contain two alternative branches depending on over flow indicator.

IFDVCK: Divide check branch.

INCR: Next 5 words contain information for increasing and testing for DO-Looping

CALL: Next words contain information for calling a subroutine.

RETURN: Do necessary subroutine return operations.

VARIABLE IN ARRAYS:

Let $ be a matrix with dimensions D1$, D2$, D3$. If $ is floating, m$=3; otherwise, m$=2. Subscript expression: The address of $ (i1*j1+k1, i2*j2+k2, i3*j3+k3) is $\{L\ \$(1,1,1) + (((k3-1)*D2\$+(k2-1))*D1\$+(k1-1))*m\$\}$ $+i3*\{D2\$*D1\$*m\$\}$ $+i2*\{D1\$*j^2*m\$\}$ $+i1*\{m\$*j1\}$

The parts in curly brackets are computed by the compiler; the rest is a subscript expression computed during execution and held in a pseudo B-box.

Let f3 = D$*D1*m$*j^3

f2 = D1$*m$*j^2

f1 = m$*$j^1$

These values are computed by the compiler then a B-box is described by:

f3, i3; f2, i2; f1, i1

and the value of the B-box is the current value of

f1*i1+f2*i2+f3*i3

STATEMENT	PROGRAM	TOTAL
GO TO n	GOx b(n) always IDLIST location	2
	A (n)	
ASSIGN i TO n	CAO b(Ti) where Ti: GO b(i)	
	A (Ti) A(i)	
	S20 b(n)	
	A (n)	4
GO TO n, (n_1, n_2,..., n_m)	GOx b(n)	
	A (n)	2
GO TO (n_1, n_2,..., n_m	TRM CMPUTD	
	m,o b(i)	
	A (i)	
	$b(n_1)$	
	A $b(n_1)$	
	⋮	
	$b(n_m)$	2m + 3
IF (a) n_1, n_2, n_3	"a" is in accumulator	
	TRM IF	
	$b(n_1)$; $b(n_2)$; $b(n_3)$; 0	
	A (n_1)	
	A (n_2)	
	A (n_3)	5
IF ACCUMULATOR OVERFLOW n_1, n_2	TRM IFOV	
or	$b(n_1)$; $b(n_2)$; 0,0	
IF QUOTIENT OVERFLOW n_1, n_2	A (n_1)	
	A (n_2)	4
IF DIVIDE CHECK n_1, n_2	TRM IFDVCK	
	$b(n_1)$; $b(n_2)$; 0;0	
	A (n_1)	
	A (n_2)	4
PAUSE or PAUSE n	HPR 0	1
	HPR n	
STOP or STOP n	TRB 1	2
DO n i = m_1, m_2, m_3	CAO $b(m_1)$	
(if no m_3, then $L(m_3)$ = L(1))	A (m_1)	
	S20 b(i)	
	A (i)	
z	UPx b(ii)	
	A (ii)	5
	⋮	
n	TRM INCR	

STATEMENT	PROGRAM	TOTAL
	b(i); b(m_2); b(m_3); b(z)	
	A (i)	
	A (m_2)	
	A (m_3)	
	A (z)	12
CONTINUE	none	0
END (in Subroutine)	none	
or same as	RETURN	?
END (Main Program)	none	
or same as	STOP	?
RETURN	TRM RETURN	1

MISCELLANEOUS STORAGE REQUIREMENTS

1. DIMENSIONed variables

   Each different subscript expression is contained in a pseudo B-box whose coding requires 4, 5 or 6 words for 1, 2, or 3 dimensional arrays. The coding is as follows for a B-box for an element of a 3 dimensional array:

```
XB      0;  0  ;  0  ; b(B)
           A(B)
    b(f1);b(f2); b(f3); 3     last octal digit gives dimensionality
          A(f1)
          A(f2)
          A(f3)
```

2. To keep all B-boxes current, each time the value of an integer not in an array is redefined (by occurring on the left of an equal sign, by replacement of a formal parameter during call of a subroutine, by trasmission through an input list, or by valid incrementation during a DO-loop), a 2 word command UPx is invoked. Suppose I is an integer-valued variable which occurs in the expression of the third subscript in B-box BO and of the first subscript in B3. Then UPx (I) will refer to the following routine.

```
(I)     0; 0; 0; b (I)      I:   location of current value of I
           A  (I)
        0; 0; 0; b(I0)      I0:  last value of I at previous UPx (I)
           A  (I0)               occurrence
        3; 0; 0; b(XBO)     third subscript of BO
           A (XBO)
        1; 0; 0; b(XB3)     first subscript of B3
           A (XB3)
```

The (I) routine will require 4 + 2 m locations where m is the number of occurrences of I in different subscript expressions. If I occurs in none, the routine (I) will consist solely of a flag. The first word of the last subscript reference to I will be negative.

3. ASSIGN

   Each statement number referred to in an ASSIGN statement will cause a 2 word GO TO command to be generated.

ID LIST TYPES

01	Floating variables
11	Floating constants
05	Floating variable subprogram arguments
03	Floating arrays
07	Floating array-type subprograms arguments
02	Fixed point variables
12	Fixed point constants
06	Fixed point subprogram arguments
04	Fixed point arrays
10	Fixed point array-type subprogram arguments
13	Labels (statement numbers)
14	Subprogram names
17	Format Statements
15	Library function names
16	Pseudo B-boxes
20	ASSIGN Transfers
21	Up subroutine
22	Unused up-subroutine
23	Incrementation information

1. Variables (not in arrays)

   WORD1: 3; Level; Length of entry - 3 (really(distance to next entry)-3)

   WORD2: 01,02,05, or 06; 0; b(object code location)

   WORD3: (object code location)

   Succeeding Words: Alphanumeric Identifier

2. Arrays

WORD1: 4+d; Level; Length of entry - 3

WORD2: 03,04,07, or 10; b (number of elements in array); b(object code location)

WORD3: A(object code location of first element)

WORD4: A(number of elements in array)

WORD5: b(D1); b(D2); 0; d

WORD6: A(D1) if d ≠ 1

WORD7: A(D2) if d = 3

Succeeding Words: Alphanumeric identifier

where d = number of dimension of the array

and D1, D2 are the first two dimensions

3. Constants

WORD1: 3; 0; Length of entry (2 for fixed, 3 for float)

WORD2: 11 or 12 ; 0; b (object code location)

WORD3: A(object code location)

Succeeding Words: Value of the constant (2 or 3 words)

4. Labels (statement numbers) or Subroutine names

WORD1: 3 ; Level ; Length of entry - 3

WORD2: 13 or 14 ; 0 ; b (object code location)

WORD3: A(relative object code location)

Label (in internal format)
or alphanumeric identifier

5. Library Function Names

WORD1: 2; 0; Length of entry

WORD2: 15; Count of number of library functions thus far encountered

Succeeding Word: Alphanumeric identifier

6. Format Statements

WORD1: 3; Level; Length of entry - 3

WORD2: 17; 0; b(object code location)

WORD3: A(object code location)

Succeeding Words: Label (in internal format)

7. Pseudo B-boxes

WORD1: 2 ; Level ; Length of entry (1, 4, 6)

WORD2: 16; b(F1); b(i1)

A(F1) These are IDLIST location for i

A(i1)

b(F2); b(i2); b(F3); b(i3) Missing if 1-dimensional

A(F2) Missing if 1-dimensional

A(i2) Missing if 1-dimensional

A(F3) Missing if 2-dimensional

A(i3) Missing if 2-dimensional

8. ASSIGN transfers: ASSIGN i to n

WORD1: 4 ; Level; 1

WORD2: 20 ; b (IDLIST entry for i); b (T (i))

WORD3: A(T(i))

WORD4: A(IDLIST) entry for i)

9. Up Subroutine UP (i) Generated during 2nd pass

WORD1: 0, 0, 0, length -3

WORD2: 21 or 22, B (IDLIST LOC of i), B(OBJ, LOC, UP-SUBT)

WORD3: IDLIST LOCATION FOR $\overset{o}{n}$

WORD4: OBJECT CODE LOC. UP SUBR. (7777 if not used)

10. Incrementation Information

WORD1: 3; Level ; 0

WORD2: 23; 0; B(INCR + 1)

WORD3: A(INCR + 1)

LSTSPY: Searches IDLIST for identifier already formed which is the same as one currently being processed. If it finds no previous entry, it creates one. In any event, it returns with object code location of identifier in all those cases where it can have this information.

Entry: Identifier type is in LSTTYP, beginning location is in LETBEG (=NUMBEG), ending location + 1 is in LETEND. For variables, LSTTYP at entry is 0, 1 if mode is floating, fixed. Current level is in LEVEL.

Method: Starting with entry at location LASTID in bank LSTBNK, find entry of same level, type, and name as identifier being processed. If lower level or end of storage is encountered with no match, then go to MAKEIDentifier list entry subroutin and return as though identifier had been found.

Exit: Bank of object code location is in B(OBJ), relative address is in A. Bank of idlist entry is last digit of memory location SICIDl; beginning address of idlist entry is in LOC(ID.

Procedure: (LSTSPY)

1. Initialize: LOC(BK = LSTBNK, NEXTNAME - LASTIDentifier.
2. If idlist type is variable name, set mask to use only last bit for mode; otherwise look at all 5 bits of type.
3. Set relative address of beginning of next entry and next bank.
4. Has available storage been exceeded: Yes, go to 18.
5. No, set indirect bank to B(ID).
6. Compute address of beginning of next entry (NXTNAM).
7. Is it in next bank? Yes, increase bank indicator LOC(BK.
8. Set beginning of IDLIST name.
9. Does IDLIST level agree with LEVEL? Yes, go to 12.
10. Is constant being looked for? No, go to 18.
11. Yes, is this IDLIST entry for a constant? No, back to 3.
12. Is type (masked) of IDLIST entry same as type being sought? No, back to 3.
13. Yes, do names agree? No, go to 3.
14. Yes, put unmasked type into LSTTYP.
15. Store object code data bank in B(OBJ).
16. Load object code relative address.
17. Return
18. Go to MAKEID, return with B(OBJ) set and with relative address in A. Back to 17.

SUBSCRipt processor

Have just encountered array name X

If subscript expression is (m1*I1±n1, m2*I2±n2, m3*I3±n3)

where X is D1 x D2 x D3 and 1X1 is 2(3) for fixed (floating)

Need base: $L\left[X(1,1,1)\right]$

Need additive: $\left\{\left[(\pm\ n3-1)\times D2\pm n2-1\right]\ \times D1 \pm n1-1\right\} \times 1X1$

Need B-box: 1X1xm1 for I1, (1X1 x m2) x D1 for I2, $\left[(1X1\ \times m3) \times D2\right] \times D1$ for I3

1.) Initialize: J = 1, ADDTVE = 0

2.) Fetch NUMber of DIMensions, create proper representation of D1 and D2, create NUMber of WoRDs per entry, by-pass left parens.

3.) FORM next OPerand. If integer constant (12) go to 5)

4.) If integer variable (02 or 06), then set M(J) = 1 and go to 7)

5.) Check next symbol. If right parens or comma, set M(J) = 0 and IDI(J) = 0. Store constant as N(J). Go to 11)

6.) Constant is M(J). If next symbol is asterisk, FORM next OPerand. Must be integer variable (02 or 06).

7.) Store IDI(J), the idlist location of variable I(J).

8.) Is next symbol right parens or comma? Yes, set N(J) = 0 and go to 11)

9.) Next symbol must be plus or minus. Save this.

10.) FORM next OPerand. Must be integer constant (12). This is N(J) and next symbol must be comma or right parens.

11.) ADD(J) = (=N(J)-1) x NUMWRD. L = -J

12.) L = L + 1. If L = 0, go to 14). (K = J + L)

13.) Otherwise, M(J) = M(J)*D(K) and ADD(J) = ADD(J)*D(K). Go to 12)

14.) ADDTVE = ADDTVE + ADDCJ)

15.) Was last symbol encountered a right parens.? Yes, go to 17)

16.) J = J + 1. Gotto 3).

17.) Form B-box LISTSPY for previous occurence and STore in Identifier List if necessary.

18.) Exit with additive in ADDTVE and Idlist location of pseudo-B-box in LOC(BK and LOC(ID

ALGebraic STRing pre-processor

Scan BCD input string (blanks eliminated) from left to right forming MATHSTring by replacing operators by their internal designations (0-14), replacing variable names and constants either by 2-word descriptions if space allows (which it always will if the identifier consists of at least two characters) or else by one word descriptions:

Two-word description: [Idlist type, o, b(Idlist loc)] [A(Idlist location)]

One-word: [Idlist type, BCD character] for non-integer idlist type or [Idlist type, 40 + BCD character] for integer

Note that any floating constant requires at least two source BCD characters.

During this scan, indexing functions are computed by jump to SUBSCRipt calculation and are left as three word descriptions.

Three-word description: [0,b(additive),0,b(ID for B-box)] [A(additive] [A(ID for B-box)]

This three-word package is placed immediately after the array identifier it modifies. Each array mentioned makes this requirement. There is always room for this since the shortest index specification, viz. (I), requires three BCD characters. (An array name mentioned in an arithmetic statement without any further specification will be taken to be the first element of the array). These B-box occurences will be counted.

Library function names are at least 2 BCD characters in length. They will be replaced by one word descriptions of the following form:

[1 (0 or 1) 1 0 11, Number in function library]

i.e., [43, n] for floating functions

and [73, n] for integer-valued functions

A pseudo-end (0011) is inserted after the last character.

Second Scan - MATH STring collapsed on itself

Initialize: ERàSable LOCations beginning = Beginning of OPerand LiST

= Location of pseudo-end

and L CHI) = MATHST = L(PROC

Fetching will occur through L(CHI) and storing through MATST.

BOXLiST is set to min {first available IDLIST location; 1,7776} minus twice the number of B-box references (not necessarily the number of different B-boxes) in the statement.

1.) If $0 \leq C[L(CHI)] \leq 15$, then store $C[L(CHI)]$ in MATHST.
If "end" (0006), go to 14. If "start" (0015) set BINIT equal to MATHST GO to 10)

2.) If $C[L(CHI)] < 0$, then library function so shift around 6 and mask off original sign bit. Store in MATHST and go to 10)

3.) To get here must have operand of some type. Is it a one-word description? Yes, go search IDLIST for it and return with IDLIST location and create a 2-word description.

4.) Is the operand the name of an array? Yes, go to 11)

5.) Is 2-word operand entry already in operand list? Yes, go to 9)

6.) No, are there two words available between ERSLOC and BOXLST? Yes, go to 8)

7.) No, try to condense by moving words from L(CHI) through ERSLOC to the area starting at MATHST. Note relocation of "end" and set BOPLST accordingly. Set L(CHI) = 1+ MATHST. Change ERSLOC to give new location of pseudo-end.

8.) Add new operand entry to list.

9.) Create 12 + relative operand designation and add to one word entry in MATHST.

10.) Increase L(CHI) and MATHST; go to 1)

11.) Isolate B-box information. Check through existing B-box list. Has it occured previously? Yes, go to 13).

12.) No, add new B-box to list.

13.) Create relative B-box designation and 2 word operand list entry. Go to 5).

14.) Condense string by moving L(CHI) through ERSLOC to area starting at MATHST. Note relocation of BOPLST and ERSLOC.

15.) JPR to ALGeBRA. On return, exit to caller.

MAKe a STOre command

DIMensioned variable SWitch is 0 if no arrays in this subprogram

BMODE is 0 for floating, 5 for fixed

BMDSW is 0 for store in same mode as arithmetic, 2 for changed

BOXSW contains number of index register for store (0 if none)

STOBNK contains 20 + storage bank for idlist entry for integer variable

STOADD contains relative address for idlist entry for integer variable.

Procedure:

1.) Set BSTOre SWitch to 13. JPR to MAKe COMmand with 13 in A.

2.) Was store of an integer? No, go exit

3.) Are there dimensioned variables? No, so no UPP's. Go exit.

4.) Was store indexed? Yes, go exit.

5.) No, generate UPP command (7600) with bank of idlist entry for the integer variable as low order bits. PUTaWAY.

Next word of UPP command is relative address of idlist entry. PUT this aWAY also. Go exit.

PRE-ALGebra

On entry, a string of BCD characters exists in bank 1 extending from L(PROC to L(BUFL

1.) JPR to ALGebraic STRing processor. On return, all object code for the right side of the arithmetic statement has been put away.

2.) JPR to MAKe a STOre command and an UPP if necessary.

3.) EXIT

NTRID - creates all idlist entries which consist of PReAMBLe followed by IDENtifier. The preamble starts in PRAMBL and has N(PRAM elements to it. The identifier starts in IDEN and extends to (not through) the address found in ID*END. The first word of preamble looks like [NCPRAM, level, 0]. At entry, the last identifier had been located at LSTBaNk address LASTID.

At exit, the LaSTBaNK, LASTID has been corrected and the entry made.

1. Compute length of complete entry = LSTLNG.
2. Is LSTLNG > LASTID? No, go to 4.
3. Yes, LSTBNK = LSTBNK-1, TEMP = LASTID, LASTID = -LSTLNG, LSTLNG = LSTLNG + TEMP
4. Set L(CHAR = LASTID, set indirect bank to LSTBNK.
5. Place LSTLNG-3 as last 5 bits of first word of preamble.
6. S = -NCPRAM, I = 1
7. L(CHAR [I] = PRAMBL [N(PRAM-S+1]
8. I = I + 1.S=S+1. If S ≠ 0, go to 7
9. *S = IDEN-ID*END(=MAX)
10. L(CHAR I = IDEN [MAX-S + 1]
11. I = I + 1. S-S + 1. If S ≠ 0, go to 10.
12. Set indirect bank to 1 and exit.

MAKEID - creates idlist preamble for simple variables (not formal parameters) and for constants. This routine calls NTRID which puts the entry into idlist.

1. If LSTTYP = 11 or 12, level should be 0.
2. PRAMBL = 3; level; 0
3. Compute address of next data entry. Last entry was at B(DATL,DATEND. If DATEND 3 (or 2 for non-floating) then B(DATL = B(DATL-1 and DATEND = DATEND-3 (or 2)
4. PRAMBL+1 = LSTTYP,0, B(DATL [Note: Set LSTTYP=2 if LSTTYP = 0]
5. PRAMBL+2 = DATEND
6. N(PRAM = 3
7. JPR to NTRID
8. Load A with LSTTYP and exit.

STATNO: Preliminary processor for all statements whose first five characters are not blank. On entry, L(CHI) contains indirect bank location of first non-blank character and A contains that character minus BCD blank (20).

1. Is character numeric? No, go to 3.
2. Yes, go check idlist for statement number. If it does not already exist, make an entry (STNOID). In any event place relative object code address B(CODL, CODEND in the idlist entry. Set LAST NO to location of beginning of idlist entry. Set L(CURRent, the distance from last label to current statement, to zero. EXIT.
3. Is character S? Yes, go to SYMBOLic coding processor.
4. Is character O? Yes, go to (OCTAL to putaway octal coding.
5. Is character B? Yes, put zero in SWBOOL, blank out character located by L(CHI) and go to Q.STNO.
6. No, TILT with NOSTAT.

STNOID: Makes preamble of statement label entries.

Calls = NTRIDentifier to place statement label in list.

1. Form label identifier. Search idlist for previous occurence. If found, go exit at 7.
2. Otherwise PReAMBLe = 3,LEVEL,0
3. PReAMBLe + 1 = 13, 0, 0
4. PReAMBLe + 2 = 0 until relative object location is assigned.
5. N(PRAM = 3 elements to preamble.
6. Go eNTeR IDentifier in idlist.
7. Exit

MAKe COMmand is a routine which generates interpretive commands not occuring in ordinary algebraic expressions. Specifically it is used to generate stores after evaluation of an algebraic statement, to generate stores and fetches of elements of input-output lists, and to provide the means of implementing a set of symbolically code (S) instructions. The operations that can be performed are load, store, add, subtract, multiply, and divide both in fixed and floating mode.

On entry to MAKCOM, the A contains the command type BOXLST must contain first locatior of B-box list (Bank 1)

BOPLST must contain first location of operand list (Bankl)

BSCNLC = L(PROC must contain location of one-word description (Bank 1)

BMODE must contain mode (0 for floating, 5 for integer)

BMODE must be set to agree with the mode of the operand for all commands except store after arithmetic statement which can generate a convert accumulator command.

On exit, the appropriate commant has been PUTaWAY.

Procedure:

1.) Store command type in ORDTYP.

2.) Set BSCNLC equal to L(PROC

3.) Zero out FORward-BAcKward scan switch, BWRONG which is used for **, and FIRST which will make subroutine BPTSUB do the PUTaWAY without a log.

4.) JPR to BSCANK for mode agreement check.

5.) JPR to BSCAND for index register usage.

6.) JPR to BSCANE. This has three exits. The first after the JPR says pseudo-accumulators or temporary erasable are used. The A-register is zero at this exit. Exits two and three have non-zero A-register and mean normal fifteen bit operand from exit two and either 1 or 3 word command from exit three.

7.) If exit one is achieved or zero in exits two or three, then TILT.

8.) If exit two, go to 10.)

9.) If exit three, go to 13.)

10.) If the command is a store, set BSTOSW to 26.

11.) If the command is a load, set LODFLG to 10.

12.) Set JPR to go to BSCANG which processes two word commands. Go to 14.)

13.) (If command is load or store then BTOSW and LODFLG are proper.) Set JPR to go to BSCANF which processes one and three word commands.

14.) Store LODFLG in BLODSW. Load A-register with command type. JPR to appropriate processor.

15.) Go to put away the command (JPR to BPTSUB).

16.) Zero out LODFLG, BSTOSW.

Return

LIST: String extends from [L(CHI)] to L(BUFL-1 in bank 1. I-OR-O contains BCDI or BCDO according as input or output is to be done. Routine must make appropriate stores (using MAKSTO) or loads (using ALGBRA). In addition, indexing control information must be generated. Caution: Special care must be taken to insure preservation of and restoring of low core values used as locators since a great many routines may be used by this processor.

1. Save L(PROC in L)PROC.
   Replace contents of L(BUFL by BCD, and increase L(BUFL. Save in SWBOOL.
2. Is list exhausted, ie., does L(CHI) = SWBOOL? Yes, restore L(PROC and exit.
3. Set MATHST = L(CHI). Go to FORMOP.
4. Is list element operator? Yes, go to 14.
5. No means operand. Is next character [L(CHI)] a ,? No, go to TILT
6. Set L(CHAR = MATHST. Set L(PROC=L(BUFL=SWBOOL. Store = in L(BUFL. Increase L(BUFL.
7. Move L(CHAR into L(BUFL. Increase L(BUFL.
8. Increase L(CHAR. L(CHAR = L(CHI)? No, go to 7.
9. Save L(CHI) in L)CHI(. Go to ALGSTR. If operand is array called by name, go to ARYNAM. Otherwise, if input is to be done, go to 11.
10. (Output): Increase L(PROC. Go to ALGBRA. Decrease L(PROC. Go to 12.
11. (Input): Set BMODE so no conversion of accumulator is done. Go to MAKeSTOre.
12. Restore L(BUFL(=L(PROC).
13. Set L(CHI) = 1 + L)CHI(. Go to 2.

14. Is operator)? No, go to 16.

15. Yes, save 1+L(CHI) in L)CHI(. Go MAKeINCrementation coding. Go to 13.

16. Is operator (? No, go TILT.

17. Set PARENS = 0. Save L(CHI) in L)CHI(.

18. Increase L(CHI). Is next op (? No, go to 20.

19. Yes, increase PARENS, go to 18.

20. Is it)? No, go to 23.

21. Yes, decrease PARENS. Is it non-negative? No, go to TILT.

22. Yes, go to 18.

23. Is it ,? No, go to 25.

24. Yes, save L(CHI) in L(CHAR. Go to 18.

25. Is it =? No, go to 18.

26. Is PARENS=0? No, go to 18.

27. Increase L(CHI). Is next character)? No, go to 27.

28. Set L(BUFL = L(PROC = SWBOOL. Save L(CHAR in L)CHAR. Increase L(CHAR.

29. [L(CHAR] goes to L(BUFL.

30. Increase L(BUFL. Increase L(CHAR. Does it = L(CHI)? No, go to 29.

31. Set L(CHAR to L)CHAR.

32. [L(CHAR] goes to L(CHI).

33. Decrease L(CHI). Decrease L(CHAR. Does it = L)CHI(? No, go to 32.

34. Save L(CHI) in L)CHI). Go to ALGSTR.

35. Set LPROCI = L)PROC-6. Go to INCSAV. Set L)PROC = L(PROC.

36. Set L(CHI) = 1 + L)CHI). Go to 3.

ARYNAM - Input/Output of Arrays with Implicit Indexing

The initialization routine (BEGN) creates IDentifier LIST entries for the integer 1, a level 0 indexing variable IØ, and for fixed and floating one-dimensional indexing functions using the variable IØ. It also plants proper last-bank references in the list called ARYDØ, a dummy list of the form ordinarily generated by explicit indexing.

1. Correct operator string which has been phonied to specify array name not followed by subscripting.
2. Create decrement, 2 for fixed point array, 3 for floating.
3. Put address of proper indexing function in B&XLST.
4. Create proper operand list entry, either
   b. (first address of array-decrement),0,0,0;A (first address-decrement) or b (-decrement),0, parameter number; A (-decrement).
5. Fetch number of elements in the array, convert into 22-bit format, and search IDLIST for it, making entry if necessary.
6. Set STØrage ADDress, STØrage BaNK to IDLIST locations for the dummy index IØ. Set locaters as though algebraic pre-processing (ALGSTR) has just been done.
7. Move the list at ARYDØ into the processing area.
8. Save the IØ indexing information (I/ØDØ).
9. Put right parenthesis at end at string.
10. Restore locaters and exit back to the list processor.

MATCHK - COMMON, DIMENSION, EQUIVALENCE identifier processor

Given an identifier in packed BCD in low core cells IDEN through ID*END - 1, this routine checks the unprocessed array list to determine a match, and if there is one, goes to eNTeRDIMension to make the appropriate entry. If not, a simple variable entry is made through eNTeRVariaBLe.

1. Save MATHST, the locater for the next item to be processed.
2. Are there dimensioned variables, i.e., is DIMSW ≠ 0? If not, go to 13)
3. If so, set L(CHI) to the beginning of the array list, MATBEG.
4. Set L(CHAR at the beginning of the next array list entry. Set NØTINT to the beginning of the candidate identifier.
5. Do the names agree so far? If not, go to 10)
6. Look at next words of both names. Is candiate name complete? No, go to 5)
7. Yes, is array name complete? No, go to 10)
8. Yes, correct LiSTTYPe and go eNTeRDIMensioned quantity.
9. Restore MATHST and exit.
10. Find end of array identifier
11. Skip to beginning of next array.
12. Are array names exhausted? No, go to 4).
13. Identifier must be for simple variable so go to eNTRVariaBLE.
14. Go to 9)

EQNUM - modify base address for equivalence variable if there is an additive. Amount of modification is number of words per entry times value of integer, and direction is increase for left half of equivalence pair, decrease for right half. Fifteen bit base is found in EQUADD, EQBANK, next character is found through L(CHI).

1. Is there modification to be done, i.e., is next character (? no, exit.
2. Save beginning and end of equivalence name.
3. Set beginning of number.
4. Check for digits to find end of number.
5. Next character should be) or else go to error in L(EQ) routine.
6. Store number of words per entry, NØTINT, in ØP,ØP+1.
7. Set integer flag, -1, and go convert the number.
8. Move converted number to arithmetic accumulator, ACCJ, and multiply by number of words.
9. Move result to ØP, move base to ACCJ.
10. Go add or subtract depending on the equivalence flip=flop EQFLIR.
11. The result replaces EQUADD, EQBANK.
12. If memory overflow, go to error in L(EQ).
13. Bypass ending).
14. If right equivalence go to 16).
15. Exit.
16. Restore beginning and end of equivalence name. E xit

L(EQ) - processor of equivalence statements

For each equivalence pair, check for previous occurence at left-hand name and make idlist entry as needed. Save location of variable (or first word of array) in EQBANK, EQUADD. Check for additive and increase the base address as necessary. Check for previous occurence of right-hand name, and, if it has previously occurred, call it an error. Check for additive and decrease the base address as necessary. Then make the entry for the right-hand name.

1. Skip letters in the name EQUIVALENCE. Set L(CHI) to the next character. Set A to (.
2. Next character, [L(CHI)], must be the same as A. If so, go to 4.
3. Error: reset the flip-flop, EQFLIP, to starting position and store it also in EQUFLP. Exit to error routine, L(TILT.
4. Isolate the variable name, fix its type as fixed or floating (LSTTYP), and store number of words per element in NØTINT.
5. Save location of character ending name.
6. If left half, go to EQNUM to check for additive.
7. Pack identifier and search IDLIST.
8. Has it occurred before? No, go to 11).
9. Yes, is identifier for right half? Yes, to to.3).
10. Otherwise, go to 12).
11. Make appropriate entry using routine MATCHK.
12. If left hand of equivalence, go to 19.
13. If right hand, restore end of name locater in L(CHI).
14. Is next character)? No, go to 3).
15. Increase L(CHI). Are all characters exhausted? Yes, to to 22.

16. Is next character , ? No, go to 3).

17. Increase L(CHI). Set specific cell to).

18. Flip-flop EQFLIP. Load A with contents of specific cell. Go to 2).

19. For left hand of equivalence, fetch base address and put it in EQBANK, EQUADD.

20. Check for additive using EQNUM. Restore locater of end of name.

21. Set specific cell to , . Go to 18).

22. Are there dimensioned variables? No, exit to process them.

23. Otherwise, normal exit to CØEXIT.

Subroutine FORMOT

1. Enter by JPR
2. Move B(DATL, DATEND, L(PROC to NTEMP1, NTEMP2, L(CHI) respectively.
3. Dump buffer.
4. Set OUT = FIRST
5. C(OUT = 1(NTEMP1) 00 000 000, OUT = OUT + 1
6. C(OUT) = C(NTEMP2)
7. OUT = OUT + 1
8. C(OUT) = C(L(CHT)), C(FIRST) = C(FIRST) + 1
9. L(CHI) = L(CHI) + 1
10. If L(CHI) = L(BUFL) go to 17.
11. NTEMP2 = NTEMP2 + 1
12. If NTEMP2 ≠ 0, go to 15.
13. Shift ENDSWC, if -, go to 3.
14. NTEMP1 = NTEMP1 + 1, NTEMP2 = -1, go to 7.
15. OUT = OUT + 1
16. If OUT = FIRST + 80D go to 3, otherwise go to 8.
17. Dump buffer, reset NDSWC, Exit.

160-A Fortran - Compiler Pass 1

Subroutine PUTWAY

1. Enter by JPR, word in A-register
2. Save word in TEMP
3. If OUT ≠ FIRST go to 5
4. C(OUT) = 0, OUT = OUT + 1
5. If OUT ≠ LAST go to 8
6. Dump buffer
7. OUT = FIRST, go to 3.
8. C(OUT) = C(TEMP)
9. OUT = OUT + 1
10. C(FIRST) = C(FIRST) + 1
11. CODEND = CODEND + 1
12. If CODEND ≠ 0 go to 14
13. B(CODL = B(CODL + 1
14. Exit

SECTION 1.3

160-A FORTRAN ALGEBRAIC TRANSLATOR

INPUT INFORMATION

The algebraic translator derives the necessary information to encode an algebraic statement for the generation of object code from three lists.

1. Statement scan list
2. Operand information list
3. B-box information list

Statement Scan List:

This is a list, beginning with a start code and terminating with an end code, of the necessary operation indicators that define an algebraic statement. These operation indicators are represented by the following symbols and mnemonics,

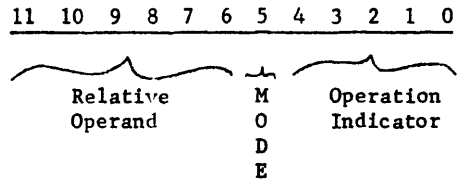
+; -; *; /; **; (;); ;, ; FUNCTION; START; END; OPERAND.

The operand indicator is one of two types.

1. Normal Fortran operand
2. Call Subroutine operand

For both cases the operand is a relative operand location which refers to a position in the Operand Information List containing more specific information about the true Fortran operand.

The Statement Scan List is composed of 1 word (12-bit) quantities in the following format.

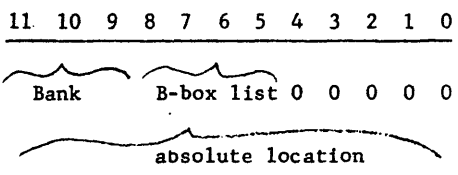


Operand Information List:

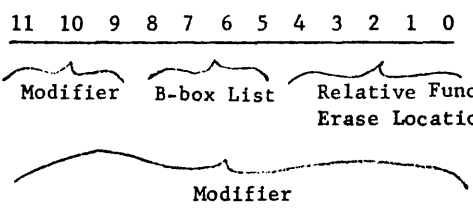
This list has 2-word (24 bit) quantities which contain the necessary information for referencing Fortran operands in the preparation of object code. There are two possible formats for this list.

1. Normal Fortran Operand format
2. Call Subroutine Operand format

The normal format contains the bank setting of the required operand, the absolute location of the required operand and a relative B-box List location if the operand is to be index register modified.

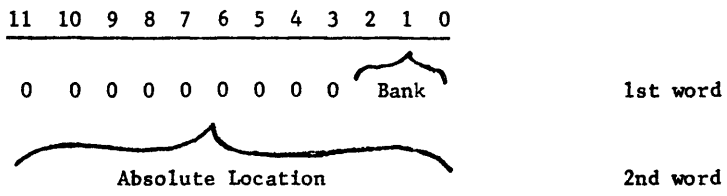


The call format contains the relative function erase operand location, and a 15-bit modifier if the operand is to be index register modified.



B-box Information List:

This list has 2 word (24 bit) quantities which contain 15 bit absolute B-box locations.



SECTION 1.4

160-A FORTRAN

ALGEBRAIC TRANSLATOR

The algebraic translator is divided into four major sections.

1. A section designed to delimit and isolate a parentheses group for processing.
2. A section designed to generate the necessary linkage (object code) if the delimited parentheses group is a function group.
3. A section designed to generate the object code necessary in implement the meaning of a parentheses group.
4. A section designed to process a Boolean statement.

The four sections stated above, various associated subroutines, and some necessary definitions will be described in the following pages. The actual step by step logical procedure of the translator is described in a second document entitled "160-A Fortran Algebraic Translator Flow Analysis".

Contained within the first three major sections is the fourth section, the necessary logic to perform the translation if the encountered statement is defined as Boolean.

Definitions

For this document the list of words below will have the meanings stated in the associated description.

1. OPERATION INDICATORS: Any one of the following symbols or mnemonics.

**	END	FUNCTION
*	)	(
/	,	START
+	PSEUDO END	
-		

2. OPERATOR: Any one of the following operation indicators: **; *; /; +; -.
3. ENDING INDICATOR: Any one of the following operation indicators: END;); ,; PSEUDO END.
4. STARTING INDICATOR: Any one of the following operation indicators: FUNCTION; (; START.
5. OPERAND: A constant or variable.
6. ALGEBRAIC STATEMENT: Any normal, meaningful grouping of operation indicators and operands. The first symbol must be START and the last symbol must be END. Imbedded blanks are acceptable. This grouping will, in general, represent the right hand side of an algebraic equation.
7. PARENTHESES GROUP: A subsection of an algebraic statement which has one of the following pairs of delimiters.

    a. START; END
    b. (;)
    c. (; ,
    d. ,;)

8. FUNCTION GROUP: A parentheses group preceded by a FUNCTION operation indicator.

9. SUBGROUP: A subsection of an algebraic statement which has one of the following pairs of delimiters.

   a. START; END

   b. START; $\pm$

   c. (;)

   d. (; $\pm$

   e. $\pm$;)

   f. $\pm$; ,

   g. ,; $\pm$

   h. $\pm$; $\pm$

10. PARENTHESES GROUP LEVEL OR LEVEL: Essentially a parentheses group, but used to describe the hierarchy or order of evaluation (encoding) of parentheses groups. For instance, the innermost parentheses group level of an algebraic statement will be encoded first.

11. ENCOMPASSING GROUP: The next higher level parentheses group from the current delimited parentheses group.

12. EVALUATE OR ENCODE: Create object code from source language.

13. MODE: The type of arithmetic, floating point or integer, associated with an operand, parentheses group, or an algebraic statement.

14. GENERATE: Create object code.

15. HIERARCHY LEVELS: Within a subgroup there are three levels of evaluation. They are in order of importance **; *, /; +, -. The normal sequence of evaluation is as follows.

    a. Scanning from left to right evaluate the ** operator.

    b. Scanning from the ** operator to the left, evaluate all *, / operators.

    c. Scanning from the ** operator to the right, evaluate all *, / operators.

    d. When all subgroups have been evaluated in the sequence a, b, c, scan from left to right to evaluate the +, - operators.

16. PSEUDO ACCUMULATORS: Intermediate storage to be used when subgroups inside of parentheses are being evaluated.

17. CONVERT: For mixed arithmetic statements an operand or result of an evaluation may have to be converted (i.e. floated or fixed depending on the direction of conversion).

Parentheses Groups

The evaluation of algebraic statements with parentheses groups and subgroups as the main units of evaluation will be given in the following description.

1. Delimit the next parentheses group and encompassing group.
2. With the parentheses group delimited in (1) to define the level of evaluation, delimit the next subgroup of the encompassing group.
3. If there is a parentheses group contained in the delimited subgroup of (2) go to (6).
4. Evaluate the subgroup (return transfer to Within Parentheses Group (W.P.G.)).
5. Go to (12).
6. Evaluate the parentheses group (return transfer to W.P.G.).
7. If the evaluated parentheses group is a function group do a return transfer to Function Groups (F.G.).
8. If the subgroup in which the evaluated parentheses group is contained does not embody one of the following criteria go to (11).

    a. More than one parentheses group.

    b. A ** operator

9. Generate a STORE ERASE command.
10. Go to (1).
11. Finish evaluating the subgroup (return transfer to W.P.G.).

12. Interrogate the encompassing group. If it has been completely evaluated go to (1).

13. Go to (2).

14. Go to (3).

Function Groups

1. If this function is not an Absolute Value Function go to (4).
2. Generate a SET SIGN PLUS command.
3. Go to the Return Exit.
4. If this function is the same as the last encountered function go to (7).
5. If this function is not one of a series of stacked function (functions within functions go to (7)).
6. Generate an INCREMENT FUNCTION ERASE COUNTER command.
7. If this is a single valued function go to (10).
8. Generate a STORE FUNCTION ERASE command.
9. If this is not the terminal function parameter go to Parentheses Group (P.G.).
10. Generate a TRANSFER (to function) command.
11. If this function is one of a series of stacked functions, generate a DECREMENT FUNCTION ERASE COUNTER command.
12. Go to the return exit.

Within Parentheses Groups

1. Delimit the next subgroup. Scan from left to right.
2. If the subgroup does not contain a ** operator go to (9). For the following discussion assume the example A ** B.
3. Generate a LOAD B command and a STORE PSEUDO ACCUMULATOR command.
4. Generate a LOAD A (or a LOAD AND CONVERT A if A is an integer quantity and B is a floating point quantity).
5. Generate a TRANSFER TO POWER command.
6. If the results of the power calculation need converting, generate a CONVERT command.
7. If there is more than one ** operator in the subgroup go to (19).
8. Assuming a scan from the ** operator to the left, go to (9) to evaluate that portion of the subgroup.
9. If the subgroup does not contain any *, / operators go to (21).
10. Interrogate the next operation indicator. If it is not a * or / go to (14).
11. If mixed arithmetic is necessary, generate a LOAD AND CONVERT command.
12. Generate the appropriate MULTIPLY, DIVIDE, or INVERSE DIVIDE (for a right to left scan) command.
13. Go to (10).
14. If the scan had been in the normal forward direction (left to right), go to (17).
15. If the initial operator of the subgroup had been a - operator generate a CHANGE SIGN command.

16. Assume a forward scan and go to (10).
17. If there are any intermediate results in a Pseudo Accumulator generated, generate an ADD PSEUDO ACCUMULATOR command.
18. If there are no more subgroups left to be evaluated go to (23).
19. Generate a STORE PSEUDO ACCUMULATOR command.
20. Go to (1).
21. Scanning from left to right interrogate the first encountered operator.
22. Generate either a LOAD or a LOAD NEGATIVE command depending on the sign of the first operator.
23. Interrogate the next operation indicator. If it is an ending indicator go to the Return Exit.
24. Generate the appropriate ADD or SUBTRACT command.
25. Go to (23).

Boolean Statement Evaluation

Boolean statements do not follow the same set of rules for evaluation as a normal algebraic statement. There are three major differences.

1. Operator hierarchy
2. Parentheses group combination
3. Mixed arithmetic

Operator Hierarchy:

The normal algebraic operators have the following associated meaning within a Boolean statement.

** Shift

* And

/ Exclusive OR

+ Inclusive OR

- Load Complement

Within these five operators there is three levels of hierarchy. They are in their order of evaluation importance:

1. -
2. **, *
3. +, /

There may be only one - operator within a subgroup. If one does exist it has to be the leading operator. Non-adherence to these two rules will create object code which is essentially nonsense.

Parentheses Group Combination:

There is no attempt to optimize the object code generated after a parentheses group evaluation. A STORE ERASE command will always be generated. That is intermediate storage will always be used, whereas in a normal algebraic statement intermediate storege will be avoided if possible within the rules of the evaluation algorithm.

Mixed Arithmetic:

Differentiation between the two modes of operands, integer and floating point, will be avoided. That is operands of both modes may be interspersed within a Boolean statement without any difference in evaluation.

Within the limits of the above three differences, statement evaluation is identical to a normal algebraic statement. The logic needed to differentiate between the two types of statements is interspersed, at the appropriate places, within the algebraic translator.

160-A FORTRAN ALGEBRAIC TRANSLATOR

List of Subroutines

The following is a list of all the subroutines used in the translation of an algebraic statement. Reference to these subroutines is made in the section entitled "160-A Fortran Algebraic Translator Flow Analysis". A description of the input and output quantities for the individual subroutines subsequently follows:

BSCANA: OPERATOR SCAN

BSCANB: SCAN FOR ERRORS

BSCANC: MODE DETERMINATION

BSCAND: INDEX REGISTER TEST

BSCANE: OPERAND TYPE TEST

BSCANF: ARITHMETIC FUNCTION ERASE COMMAND

BSCANG: ARITHMETIC COMMAND

BSCANH: PARENTHESES GROUP DETERMINATION

BSCANJ: OPERATOR DETERMINATION

BSCANK: MODE EQUALITY TEST

BSCANL: RELATIVE OPERATOR DETERMINATION

BSCANM: ACC_n/ERASE ARITHMETIC COMMAND

BSCANN: STORE ACC_n/ERASABLE TEST

BSCANO: OPERAND INTERROGATION

BSCANQ: OPERAND LOCATION DETERMINATION

BSCANR: LOAD DRIVER ROUTINE

BSCANS: STORE ERASABLE ROUTINE

BSCANT: PARENTHESES GROUP STORE TEST

BSCANN: MODE COMPARISON

BPTSUB: PUT AWAY DRIVER

BSCANA: OPERATOR SCAN

INPUT:

1. BINITA = Initial group location.

OUTPUT:

1. BTRMNL = Terminal subgroup location.
2. BTRMOP = Terminal subgroup operator.
3. BH23SW = Status of **, *, / operators.

$$= \begin{cases} 0 \text{ if not encountered} \\ 0 \text{ if encountered} \end{cases}$$

4. BH23SW = Status of *, / operators.

$$= \begin{cases} 0 \text{ if none encountered} \\ \text{location if encountered} \end{cases}$$

5. B+-SW = Status of leading lower level operator.

$$= \begin{cases} 4 \text{ for no leading operator} \\ 4 \text{ for + leading operator} \\ 5 \text{ for - leading operator} \end{cases}$$

6. B*/SW = Location of leading lower level operator.

$$= \begin{cases} \text{(BINITA) for no leading operator} \\ \text{list location for } \pm \text{ leading operator} \end{cases}$$

7. BCOUNT = Octal count of the number of elements in the subgroup.

(excluding parentheses group elements).

8. FRSTOP $= \begin{cases} \text{(BINITA) for a 1st element operator} \\ 0 \text{ for a 1st element operand} \end{cases}$

9. BPARSW $= \begin{cases} 0 \text{ for no encountered parentheses group} \\ \neq 0 \text{ for an encountered parentheses group} \end{cases}$

10. BPWRSW = number of ** operators in subgroup.

11. BH3Sw = Status of ** operators.

$$= \begin{cases} 0 \text{ if none encountered} \\ \neq 0 \text{ if encountered} \end{cases}$$

BSCANB: ERROR SCAN

INPUT:

1. Initial list location.

OUTPUT:

1. Either a proper exit indicating no errors, or an error exit to TILT with the proper error indicator in the Accumulator. The types of errors are listed below:

   Ending operator.

   Two adjacent operands.

   Two adjacent operators.

   Unequal (and).

   Two ** operators in sequence.

   Initial **, *, or /.

   Misplaced commas.

   Compiler error.

BSCANC: MODE DETERMINATION

INPUT:

1. Accumulator = List starting location.

OUTPUT:

1. BMODE = $\begin{cases} 0 \text{ for floating mode} \\ 5 \text{ for integer mode} \end{cases}$

BSCAND: INDEX REGISTER TEST

INPUT:

1. BSCNLC = operand location.
2. BINDSW = current relative I.R. address.
3. BOPLST = Initial operand list location.
4. BOXLST = Initial B-box list location.

OUTPUT:

1. BINDSW = New relative I.R. address.
2. Generated Load I.R. command, if one was necessary.
3. BOPRLS = Absolute location of operand.
4. BOPRLS+1 = Absolute location + 1 of operand.
5. BOXSW = { 0 for no I.R. reference; rel. B-box location for an I.R. reference }

BSCANE: OPERAND TYPE TEST

INPUT:

1. BSCNLC = Operand location.
2. SWBOOL = 0 for Boolean statement.
   0 for normal statement.

OUTPUT:

1. BLODSW = 0
2. ACCTYP = 7777
3. BVARTP = { 0 for ACC_n operand; 7777 for Erasable operand; absolute location for Erasable (Boolean statement); 0 for normal operand; 0 for Function erasable operand }
4. BANK = { Rel. location for an erasable operand.; 0 for an erasable operand (Boolean statement); Bank (0-7) setting for a normal operand }
5. ERSBIT has the nth (n = 1 - 12) bit cleared for an erasable operand.

6. Accumulator = $\begin{cases} 0 \text{ for } ACC_n \text{ or Erasable operand} \\ \neq 0 \text{ for normal or Function erasable operand} \end{cases}$

7. One of 3 returns to main program.

   A. Return: ACC_n or Erasable operand

   B. Return + 1: Normal operand

   C. Return + 2: Function erasable operand

BSCANF: ARITHMETIC FUNCTION ERASE COMMAND

INPUT:

1. One of the following operator codes will appear in the ACCumulator.

   2 multiply

   3 divide

   4 add

   5 subtract

   13 store

2. BSCNLC = Operand Scan list location

3. BOXSW = $\begin{cases} 0 \text{ no B-box used} \\ \neq \text{B-box used} \end{cases}$

4. BMDSW = $\begin{cases} 0 \text{ no conversion necessary} \\ 2 \text{ conversion necessary} \end{cases}$

5. FORBAK = $\begin{cases} 0 \text{ forward direction scan} \\ 7777 \text{ backward direction scan} \end{cases}$

6. BLODSW = 0 no load command
3 load command

7. BMODE = 0 floating point
5 integer

8. BWRONG = $\begin{cases} 0 \text{ general operand class} \\ 7777 \text{ ** operand} \end{cases}$

9. BOPRLS = Erasable relative location

10. BOPRLS + 1 = Modifier value (low 12 bits)

11. Bank = Modifier value (hi 3 bits)

12. BSTOSW = {0 no store option
    {13 store option

13. BOOLSW = {0 Boolean statement
    {≠ 0 normal statement

OUTPUT:

1. Generated proper 1, or 3 word command

   A. 1-word: NO CALL SUBR.

   B. 3-word: CALL SUBR.

BSCANG: ARITHMETIC COMMAND

INPUT

1. One of the following operators will appear in the ACCumulator.

   2 MPY

   3 DIV

   4 ADD

   5 SUB

   13 STORE

2. BMODE = {0 floating point
   {5 integer

3. BMDSW = {0 no convert
   {2 convert

4. BLODSW = {0 no load command
   {10 load command

5. BSCNLC = {Operand scan list location
   {0 forward direction scan

6. FORBAK = {0 forward direction scan
   {7777 backward direction scan

7. BWRONG = {0 general operand class
   {7777** operand

8. BANK = Operand bank setting

9. BOXSW = { 0 no B-box used
           { ≠ 0 B-box used

10. BOPRLS+1 = Operand absolute location

11. BSTOSW = { 0 no store option
             { 13 store option

12. BOOLSW = { 0 Boolean statement
             { ≠ 0 normal statement

OUTPUT:

A. Generated proper 2 or 4 word command.

BSCANH: PARENTHESES GROUP DETERMINATION

INPUT:

1. ACCumulator = starting scan location.

OUTPUT:

1. BCOMAS = relative function parameter location.

2. BRTPAR = { 0 terminal operator not a right parentheses
           { ≠ 0 terminal operator a right parentheses

3. BFUNSW = { 0 parentheses group not a function group
            { Function location for a function group

4. BENDSW = { 0 end of statement not encountered
           { ≠ 0 end of statement encountered

5. BINITA = Parentheses group start location

6. BINITC = Encompassing parentheses group start location

BSCANJ: OPERATOR DETERMINATION

INPUT:

1. BSCNLC: Initial scan location

2. FORBAK = { 0 forward direction scan
            { 7777 backward direction scan

OUTPUT:

1. BSCNLC = Operator Scan location
2. BOPLOC = Operator Scan location
3. BOPRTR = Operator indicator
4. BOPRSV = Operator indicator
5. BVARCD = { 0 no operand encountered
   { ≠ 0 operand encountered
6. There are one of 4 exits
   A. Return exit : *, / operator
   B. Return exit + 1: +, - operator
   C. Return exit + 2: Ending operation indicator
   D. Return exit + 3: Starting operation indicator

BSCANK: MODE EQUALITY TEST

INPUT:

1. BSCNLC = operand location
2. BMODE = { 0 floating point
   { 5 integer
3. BOOLSW = { 0 Boolean statement
   { ≠0 normal statement

OUTPUT:

1. BNDSW = { 0 for modes equal
   { 2 for modes not equal
   { 0 for a Boolean statement
2. **OPT = has ** option type

BSCANL: RELATIVE OPERATOR DETERMINATION

INPUT:

1. ACCumulator = Operation code
2. BMDSW = { 0 for modes equal
   { 2 for modes not equal

3. FORBAK = { 0 forward direction scan
   { 7777 backward direction scan

4. BLODSW = { 0 no load command
   { 3, 10 load command

OUTPUT:

1. The Accumulator will contain the necessary logical combination of the 4 input quantities.

BSCANM: ACC_n/ERASABLE ARITHMETIC COMMAND

INPUT:

1. One of the following operator codes will appear in the ACCumulator.

   2 MPY

   3 DIV

   4 ADD

   5 SUB

2. BLODSW = 0

3. FORBAK = { 0 forward direction scan
   { 7777 Backward direction scan

4. BVARTP = { 0 ACC_n operand
   { 7777 Erasable operand

5. BSCNLC = Scan list operand location

6. BANK = Relative erasable operand location

7. ACCN = ACC_n operand location + 1

8. BMODE = { 0 floating point
   { 5 integer

9. BOPRLS+1 = Location (ACCN)

10. ACCTYP = { 7777 ACC_n type commands
    { 0004 Accumulator type commands

OUTPUT:

1. Generated proper 1-word command.

BSCANN: STORE ACC_n/ERASABLE TEST

INPUT:

1. BOPTST = $\begin{cases} 0 & \text{Previous command not function erasable} \\ \neq 0 & \text{Previous command function erasable} \end{cases}$

2. BLODSW = $\begin{cases} 0 & \text{store } ACC_n \text{ command} \\ \neq 0 & \text{store operand command} \end{cases}$

3. ACCN = 1, 2, 3 or ACC_n indicator

4. BOOLSW = $\begin{cases} 0 & \text{Boolean statement} \\ \neq 0 & \text{normal statement} \end{cases}$

OUTPUT:

1. Either one of the following as output.

   A. For a previous function erasable command, a one word store command is generated.

   B. For a previous non-function erasable command,

   $$\text{BNSW} = \begin{cases} 1 & \text{for ACC1 storage} \\ 2 & \text{for ACC2 storage} \\ 3 & \text{for ACC3 storage or operand storage (if BLODSW } \neq 0) \end{cases}$$

   C. For a Boolean statement a STORE ERASE command is generated.

BSCANO: OPERAND INTERROGATION

INPUT:

1. BSCNLC = operator scan location.

2. FORBAK = $\begin{cases} 0 & \text{forward scan} \\ 7777 & \text{backward scan} \end{cases}$

OUTPUT:

With the given input, this routine referencing the routines:

1. BSCAND

2. BSCANE

3. BSCANF

4. BSCANG

5. BSCANK

6. BSCANM

7. BSCANN

8. BSCANQ

9. BSCANR

will do the following:

1. Set the BNSW store switch or generate a STORE ACCN if necessary.

2. Check mode of operand.

3. Load the index register, if necessary.

4. Convert the operand if necessary.

5. Generate the proper command.

BSCANQ: OPERAND LOCATION DETERMINATION

INPUT:

1. BSCNLC = Potential operand scan location

OUTPUT:

1. BSCNCL = True operand scan location.

BSCANR: LOAD DRIVER ROUTINE

INPUT:

1. BSCNLC = Operand scan location

OUTPUT:

With the given input, this routine referencing this routines:

1. BSCAND

2. BSCANE

3. BSCANF

4. BSCANG

5. BSCANK

6. BSCANM

7. BSCANQ

will do the following:

1. Check mode of operand.

2. Load the index register, if necessary.

3. Convert the operand, if necessary.

4. Generate the proper load command.

BSCANS: STORE ERASABLE ROUTINE

INPUT:

1. ERSBIT with relative erase bit set

OUTPUT:

1. ERSBIT with relative erase bit cleared.

2. ERASE operand indicator to the scan list

3. STORE ERASE command.

BSCANT: PARENTHESES GROUP STORE TEST

INPUT:

1. BMODE = 0 floating point
5 integer

2. BINITC = Initial location of encompassing parentheses group.

3. BSCNLC = Right parentheses indicator

4. BOPLOC = Right parentheses indicator

5. BINITA = Left parentheses indicator

6. BINIT = Start of statement

7. BEND = End of statement

BSCANU: MODE COMPARISON ROUTINE

Input:

1. BMODE = 0 floating point / 5 integer
2. BMDSAV = 0 floating point / 5 integer

Output:

1. Generated CONVERT ACCUMULATOR command if BMODE $\neq$ BMDSAV and BMDSAV1 $\neq$ BMODE. Where BMDS V1 is the mode for the encompassing parenthesis group.

8. BENDSW = 0 end of statement not reached
   ≠0 end of statement reached

9. BOOLSW = 0 Boolean statement
   ≠0 normal statement

OUTPUT:

1. One of the two returns

   A. Start of Scan: Generate a STORE ERASE COMMAND

   B. RETURN : Continue evaluation

2. Right and left parentheses indicators cleared.

BSCANU: ——→

BPTSUB: PUT AWAY DRIVER SUBROUTINE

INPUT:

1. BNOWDS = 1, 2, 3, or 4 number of words that make up the command.
2. BWORD1, BOWRD2, BOWRD3, BWORD4 contain the command information.
3. BNSW = 0, 1, 2, 3 = pseudo accumulator bits for the previous generated command.
4. Specific Cell = 0 first time thru routine
   ≠0 not first time thru routine

OUTPUT:

1. GO to PUTAWAY to output the previously generated command.
2. BOPTST = 0 (erasable storage option switch)
3. BNSW = 0
4. BNOWDS = 1
5. Specific ≠0

SECTION 1.5

ALGEBRAIC STATEMENT TRANSLATOR

The translator, is, in general, divided into three portions:

1. A program to isolate and create the object code linkages between parentheses groups.
2. A program to create the necessary object code for those parentheses groups that are function groups.
3. A program to create object code for an isolated parenthesis group.

In general, the Algorithm is as follows:

1. Isolate an innermost parenthesis group.
2. Evaluate (generate object code) the isolated group.
3. If the group isolated is a function group generate the necessary STORE, FUNCTION, ERASE and TRANSFER commands.
4. If it is necessary generate a STORE ERASE command for the results of the parenthesis.
5. If more object code can be generated before a STORE ERASE is necessary, generate this object code then generate the STORE command.
6. Continue at (1) until the statement is exhausted.

A more detailed analysis of the three sub-algorithms follows.

Parentheses Groups

1. Isolate a parenthesis group for evaluation. Also determine the delimiters for the next encompassing higher level parentheses group.

2. Using the higher level deliminters isolate a subgroup. A subgroup is defined as having one of the following pairs of deliniters:

   A. (,)

   B. (, $\pm$

   C. $\pm$, $\pm$

   D. $\pm$,)

3. If the subgroup does not contain a parentheses group, do a R. J. (return Jump) to W.P.G. (within parentheses groups) to evaluate the subgroup. GO TO (8).

4. If the subgroup does contain a parenthesis group, do an R.J. to W.P.G. to evaluate the parenthesis group isolated in step (1).

5. If the parenthesis group was a function group, do an R.J. to F.G. (function groups).

6. Scan the rest of the subgroup. If it contains a ** evaluation or another parenthesis group, or if the statement is Boolean, generate a STORE ERASE command. GO TO (1).

7. If the subgroup did not contain a ** operator do an R.J. to W.P.G. to evaluate the rest of the subgroup, if possible (/, + and /, / combinations not valid in a right to left scan).

8. If the end of the statement has been encountered go to the final exit.

9. If the subgroup just evaluated was the last one contained inside the deliminters of the higher level parentheses group determined in step (1), go to (1).

10. Do a mode comparison check between the current parenthesis group and the mode of the encompassing group. If they differ generate a Float command.

11. Generate a STORE ERASE command. Go to (1).

Function Group

1. Check the function switch. If it references the 0th relative function, generate a SET SIGN PLUS command. Go to R.J. exit.

2. If this is a single parameter function go to (6).

3. If this function parameter does not belong to the same function as the last parameter evaluated, generate an INCREMENT FUNCTION ERASE command.

4. Generate a STORE FUNCTION ERASE command.

5. If this is not the last parameter of the current function go to P.G. (parentheses groups) step (1).

6. Generate a TRANSFER FUNCTION command.

7. If an INCREMENT FUNCTION ERASE command was generated for this function generate the corresponding DECREMENT FUNCTION ERASE command.

8. Go to the R.J. exit.

Within Parentheses Groups

1. If this routine is being entered either from step (7) of the P.G. evaluation go to (13).

2. Isolate the next subgroup.

3. Does this subgroup contain a ** operator? If it does not go to (7).

4. Evaluate the A ** B group.

5. Does the isolated subgroup contain another ** operator? If it does not go to (13).

6. Generate a STORE ERASE command. Go to (2).

7. Does the current subgroup contain any * or / symbols? If not go to (12).

8. Generate object code for the subgroup.

9. If it is necessary to accumulate a sum from previous subgroup evaluations generate an ADD ACC_n command.

10. Set an accumulate switch.

11. If there are no more * or / operators left in the parentheses go to (12). Generate a STORE ACC_n command. Go to (2).

12. Generate object code for the + and - operators finishing the parenthesis group evaluation. Go to ^{R}.J. exit.

13. Evaluate the subgroup in a reverse direction until a $\pm$ or a starting symbol is encountered. If a /, * or /, / combination is encountered exit to P.G. (11).

14. Generate a CHANGE SIGN command if a - sign was encountered. Go to (8).

In addition to the outlined algorithm three other major coperations occur.

1. Loading of Index Registers when necessary

2. Generation of conversion commands for use with mixed arithmetic.

3. Termination of the evaluation and setting of a switch when a parenthesis group is encountered as an operand.

ALGBRA: Scan statement for errors.

1. BINIT --→ BINITB.
2. Use the low order 4 bits of (BINITB) to preset a 20-way transfer switchboard.

4-BITS	MEANING	ROUTINE
00	Blank	B0
01	**	B1
02	*	B2
03	/	B2
04	+	B4
05	-	B4
06	End	B6
07	)	B7
10	,	B10
11	Error	BBERR
12	Error	BBERR
13	Function	B0
14	(	B14
15	Start	B15
16	2-word var.	B16
17	2-word var.	B16

BBERR: Compiler error.

1. Load "Compiler Error Indicator".
2. Go to TILT.

BO: Increase location counter.

1. BINITB + 1 --> BINITB, BENDSW
2. Go to ALGBRA, 2.

B1: Power operator.

1. If there is not a preceding power operator go to (4).
2. Load "Double Power Indicator".
3. Go to TILT.
4. Set double power switch (DBLPWR).
5. Go to B2, 2.

B2: *, / operators.

1. Zero power switch (BPWRSW).
2. If this operator is not a leading operator go to (5).
3. Load "Illegal Leading Operator Indicator".
4. Go to TILT.
5. If there is not a preceding operator go to (8).
6. Load "Preceding Operator Indicator".
7. Go to TILT.
8. Zero variable switch, (BVARSW)

   Zero right parentheses switch, (BPARSW)

   Set operator switch, (BOPRSW).

9. If this statement is not a Boolean statement go to B0.

10. Modify operator indicators according to the following table:

    1 --> 2

    2 --> 3

    3 --> 4

    4 --> 5

    5 --> 1

11. Go to B0.

B4: +, - operators.

1. Zero power switch, (BPWRSW)

   Zero leading operator switch, (BFIRSW).

2. Go to B2, 5.

B6: End indicator.

1. If there is not a trailing operator go to (4).

2. Load "Trailing Operator Indicator".

3. Go to TILT.

4. If there are equal left and right parentheses go to BOUT.

5. Load "Unequal Left and Right Parentheses Indicator".

6. Go to TILT.

B7: Right parentheses indicator.

1. Reduce parentheses count (BPRCNT).

2. If there are more right parentheses than left parentheses go to B6, 5.

3. Set right parentheses switch (BPARSW).

4. Go to B15, 3.

B10: Comma indicator.

1. Set first operation switch (BFIRSW).
2. If there is a bad comma inserted in the statement go to B15, 2.
3. Load "Bad Comma Indicator".
4. Go to TILT.

B14: Left parentheses indicator.

1. Increase parentheses count (BPRCNT).
2. Set first operation switch (BFIRSW).
3. If a left parentheses follows a right parentheses go to B16, 3, if not go to B15, 3.

B15: Start indicator.

1. Set first operation switch, (BFIRSW)
   Zero parentheses count switch (BPRCNT).
2. Zero right parentheses switch (BPARSW).
3. Zero double power switch (BPWRSW)
   Zero operator switch, (BOPRSW)
   Zero variable switch (BVARSW).
4. Go to B0.

B16: Variable indicator.

1. Zero first operation switch.
2. If there is not a preceding variable go to (5).
3. Load "Double Variable Indicator".
4. Go to TILT.

5. Zero operator switch, (BOPRSW) set variable switch (BVARSW).

6. Go to BO.

BOUT: Begin cracking statement

1. Zero switches.

   a. First object code output (Specific Cell).

   b. Erasable bit indicator (ERSBIT).

   c. Index Register setting (BINDSW).

   d. Store Option switch (BSTOSW).

   3. Temporary function list (ERSLOC).

2. Determine mode of statement (JPR BSCANC).

3. Preset mode of encompassing parentheses group: BMODE --- BMDSA.

4. PRESET Function Erasable Statement counter (32 --- LRASE).

5. Go to BPG.

BPG: Parentheses group determination.

1. Zero direction switch (FORBAK).
2. Preset Pseudo Accumulator Counter (2 --- ACCN).
3. Isolate a parentheses group (JPR BSCANH).
4. Determine next subgroup (JPR BSCANA).
5. Go to BPGYB.

BPGYB: Continue Scan / Erasable Store

1. If one of the following conditions exist go to BPGYA.
   a. The contents of the previous parenthesis were not exhausted.
   b. There is a parenthesis group in the current subgroup.
   c. The end of the statement terminates this subgroup.
2. Zero the right and left parenthesis deliniters of the previously executed parenthesis group.

BPGYA: Evaluate Subgroup.

1. Save terminal operator of subgroups (BTEMP2 --- BTRMOP).
2. Save terminal location of subgroups (BINITB --- BTRMNL).
3. Pseudo end inidcator to terminal operator.
4. If there was not a parentheses group in the encountered subgroup go to (5).
5. Return jump to subgroup evaluation (JPR BWPG1).
6. Go to (12).
7. Return jump to parentheses group evaluation (JPR BWPG1).
8. If this is a Function Group return jump to BFG.
9. Do a parentheses group store check (JPR BSCANT).
10. Replace the currently evaluated parentheses group with a pseudo end indicator.
11. Return jump to subgroup evaluation (JPR BWPG1).
12. Scan for next subgroup (JPR BSCANA).

13. If the subgroups have not been completely encoded go to BPGDA.

BPGDB: Exhaustion of encompassing parenthesis group.

1. If the end of the statement has been encountered go to the FINAL EXIT.
2. Zero the contents switch () -- BPGYO)
3. Go to BPG.

BPGDA: Prepare for next scan.

1. Set the Information In ACC switch (BSTRSW).
2. Do a mode comparison check (JPR BSCANU).
3. If there is a parenthesis group in the next subgroup to be evaluated go to BPGZ.
4. Go to BPGYB.

BPGZ: Reduce the scan locater

1. Reduce the location counter (BSCNLC - 1 -- BSCNLC).
2. Go to BPGG.

BPGG: Store Erase Command

1. Generate a Store Erase command (JPR BSCANS).
2. Reser the contents switch (1 -> BPGYC)
3. Go to BPG.

BFG: Function Group Evaluation.

1. If this is not the first function encountered go to (3).
2. Initialize function erasable counter (32D --> LERASE).
3. Zero the terminal operator of the group (either a comma or a right parentheses).
4. If this function is not the 0th relative function go to (7).
5. Generate a SET SIGN PLUS command.
6. Go to Return Exit.
7. If this function is not the first in a series of functions (functions within functions) go to (10).
8. Increment function list locator (ERSLOC).
9. Go to (15).
10. If this function is not the same as the last encountered function go to (17).
11. Generate INCREMENT FUNCTION ERASABLE COUNTER command.
12. Increment the "statement function erasable counter" (LRASE).
13. If the "program function erasable counter" (LERASE) has a smaller count than the "statement function erasable count" (LRASE) go to (15).
14. LRASE --> LERASE.
15. Increment function list locator (ERSLOC).
16. Save function location in list. BFUNSW --> List.
17. If there is list space still available go to (20).
18. Load "No More List Space Indicator".
19. Go to TILT.
20. Save the current number of function parameters in the function list. BCOMAS --> List.

21. If this is a single valued function go to (24).

22. Generate STORE FUNCTION ERASE command.

23. If this is not the terminal function parameter go to BPG.

24. Decrement function list locator (ERSLOC).

25. Generate a TRANSFER TO FUNCTION command.

26. If this is a function inside a another function (series of functions) go to (28).

27. Go to the Return Exit.

28. Generate a DECREMENT FUNCTION ERASABLE COUNTER command.

29. Go to the Return Exit.

BWPG1Z: Parentheses groups evaluation exit.

1. Restore terminal operator.

2. Zero Information In ACC (BSTRSW) switch.

3. Go to the Return Exit.

BWPG: Parentheses group evaluation.

1. Zero Load and Convert switch (BWRONG).

2. The mode of previous evaluation goes to the mode of current evaluation BMDSW1 --> BMODE.

3. If the Information In ACC switch (BSTRSW) is set go to BWPGAR.

4. If a backward direction evaluation is being performed go to BWPGAK.

5. Determine the mode of the parentheses group (JPR BSCANC).

6. Go to BWPGZY.

BWPGZY: Hierarchy determination.

1. Determine next operator to be processed.
2. Set operator switch (BOPRTR) to ADD.
3. Set Boolean first operator switch (BOPFST) to Inclusive OR.
4. If the next operator is a ** go to (7).
5. Zero the Load and convert switch (BWRONG).
6. Go to BWPGA.
7. Operator location to scan location (BSCNLC) and operator location (BOPLOC).
8. If this is not a Boolean statement go to (11).
9. Set operator switch (BOPRTR) to LOAD COMPLEMENT.
10. Go to BWPGAH.
11. Zero ** operator (A ** B).
12. Set Load and Convert switch (BWRONG).
13. Generate a LOAD B command. (JPR BSCANR).
14. Generate a STORE ACCn command. (JPR BSCANN).
15. If this is an (INTEGER) ** (FLOATING) option zero the Load and Convert switch (BWRONG).
16. Generate a LOAD A command. (JPR BSCANR).
17. Generate the proper TRANSFER TO POWER command.
18. Set the proper bit in the Power Option Used test word (PWRBIT).
19. If the mode of the power option equals the mode of the encompassing arithmetic go to (21).
20. Generate a CONVERT ACCUMULATOR command.
21. Go to GWPGAK

BWPGAS: Reverse Scan Check

1. If the previous operator - current operator do not have either the combination /, * or /, / go to BWPGAH.
2. Increment the scan location (BSCNLC + 1 -> BSCNLC).
3. Restore the terminal operator (BTRMOP -> (BTRMNL)).
4. Go to BPGG.

BWPGA: Process *, / operators

1. If there are no * or / operators in the subgroup go to BWPGBA .
2. Set the scan location, (BSCNLC) to the location of the first *, / operator.
3. Alter the scan location if there is a leading +, - operator.
4. Generate a LOAD command (JPR BSCANR).
5. Go to BWPGAB.

BWPGAB: Increment locater.

1. Increment the scan location (BSCNLC).
2. Go to BWPGAK.

BWPGAH: Interrogate Operand.

1. Interrogate operand and generate a multiply or divide command (JPR BSCANO).
2. Go to BWPGAK.

BWPGAK: Interrogate the next operator.

1. Determine the next operator (JPR BSCANJ).
2. If the next operator is
   a. *, /: Go to BWPGAS
   b. +, -: Go to (3).
   c. End : Go to (9).
   d. Start: Go to (7).
3. If the scan is in a forward direction go to (10).
4. Zero the +, - operator.
5. If the operator is + go to (7), if it is - go to (6).
6. Generate a CHANGE SIGN command.
7. Reset the direction switch (FORBAK) to forward (= 0).

8. If there was only one ** operator in the subgroup, go to BWPGAB.

9. Store an operand indicator in the scan lid. Go to BWPGAP.

10. If this is a boolean statement go to (13).

11. If the Information In ACC switch (BSTRSW) is not set go to (13).

12. Generate an ADD command. (JPR BSCANM).

13. Set the Information In ACC switch (BSTRSW).

14. Scan the current subgroup. (JPR BSCANA).

15. Go to BWPGAR.

BWPGAR: *, / operator check.

1. If there are *, / operators left in the subgroup go to BWPGAP, if not go to BWPGZY.

BWPGAP: STORE ACCN command.

1. Generate a STORE ACCN command. (JPR BSCANN).

2. Go to BWPGZY.

BWPGBA: +, - operates interrogation.

1. Reset scan location (BSCNLC) to start of subgroup.

2. Scan for next operator (JPR BSCANJ).

3. If the operator is a +, - go to (7), if it is an end code go to (4).

4. If a single operand is left in the subgroup go to (8).

5. Zero ending code.

6. Go to BWPGIZ. (return Exit).

7. If a leading +, - was not encountered go to (14)

8. If this is not a Boolean statement go to (11).

9. Set the operator switches (BOPRTR and BOPRSV) equal to the leading operator of the subgroup.

10. Go to (12).

11. Set the operator switches (BOPRTR and BOPRSV) to an ADD indicator.
12. Reset scan location (BSCNLC) to start of subgroup.
13. Go to (15).
14. Zero the operator indicator.
15. If the Information In ACC switch (BSTRSW) is set go to (21), if not go to (16).
16. Increment the scan locator (BSCNLC) to the operand location.
17. Generate a "Load" command. (JPR BSCANR).
18. Set the Information In ACC switch (BSTRSW).
19. Increment the scan locator.
20. Go to (2).
21. Interrogate operand and generate a command (JPR BSCAND).
22. Go to (19).

BPTSUB: Put Away Driver Subroutine.

1. If this is not the first time thru this routine for the current statement go to (7).
2. Set the first time switch (Specific Cell).
3. Zero the Use Pseudo ACC switch (BNSW).
4. Zero the Eraseable or Function Erasable Command Generated switch (BOPTST)
5. Set Number of Words Count (BNOWDS) to 1.
6. Go to the Return Exit.
7. Output the next word of the command (JPR PUTAWAY).
8. If there are more words to output go to (7), if not go to (3).

BSCANA: Subgroup Scan.

1. Set starting scan address. BINITA + 1 --> BINITB.

2. Set *, / location address. BINITA + 1 --> B*/SW.

3. Set leading +, - operator address. BINITA + 1 --> FRSTOP.

4. Zero ** address locator (BH3SW)

   Zero *, / address locator (BH2SW)
   Zero temporary cell (BTEMP1).
   Zero Items In Subgroup counter (BCOUNT)

   Zero parentheses indicator (BPARSW).

   Zero ** counter (BPWRSW)

   Zero **, *, / indicator (BH23SW)

5. Preset leading +, - operator switch (B+-SW) to an ADD.

6. If the low order 4 bits of (BINITB) is zero go to A0.

7. Use bits 1, 2, 3 of (BINITB) to preset a 10 way switchboard.

3-BITS	MEANING	ROUTINE
0	**	A1
1	*, /	A2
2	+, -	A4
3	END,)	Return exit.
4	', PSEUDO END	A0
5	FUNCTION	A0
6	(, START	A14
7	OPERAND	A0

A0: Increase location counter.

1. Increment subgroup element counter (BCOUNT) by 1.
2. Increment the scan address (BINITB) by 1.
3. Go to BSCANA, 6.

A1: ** operator.

1. If a ** operator has already been encountered in the subgroup go to (3).
2. Set ** address with current scan location. BINITB -- BH3SW.
3. Increment the ** counter (BPSRSW) by 1.
4. Set the **, *, / indicator (BH23SW.)
5. Go to A0.

A2: *, / operators.

1. If the previous operator is a /, set an indicator in the upper half of the operator scan location.
2. If a *, / operator has already been encountered in the subgroup go to A1, 4.
3. Set *, / address with current scan location. BINITB --> BH3SW.
4. Go to A1,4.

A4: +, - operators.

1. If a **, *, / has been encountered (i.e. BH23SW is set) go to the Return Exit.
2. Set *, / preceding location address with current scan location BINITB --- B*/SW.
3. Reset leading +, - operator switch (B+-SW) with the current operator.
4. Go to A0.

A14: Left parentheses operator.

1. Scan the subgroup until the matching right parentheses has been encountered.
2. Go to A0.

BSCNCZ: Return Exit.

1. Set floating point mode indicator. 0 --> (BMODE).
2. Go to the Return Exit.

BSCANC: Mode Determination.

1. Set scan address locator (BINITB) with initial scan address.
2. Zero parentheses counter (BTEMP1).
3. Increment parentheses counter (BTEMP1) by 1.
4. Increment location counter (BINITB) by 1.
5. Interrogate the current operation indicator (low order 4-bits).
   a. If it is an arithemtic operator go to (4).
   b. If it is an ending indicator go to (6).
   c. If it is a starting indicator go to (3).
   d. If it is an operand go to (7).
6. If the parentheses count (BTEMP1) is zero go to the return exit, if it is not go to (4).
7. If the operand is of a floating point mode go to BSCNCZ, if of an integer mode go to (8).
8. Set integer mode indicator. 5 --> BMODE.
9. Go to (4).

BSCAND: Index register test.

1. Interrogate the relative operand location.
   a. If it is an Erase or Pseudo Accumulator operand go to the return exit.
   b. If it is a Normal or Function Erase operand go to (2).

2. Operand list location to BOPRLS, BOPRLS + 1.

3. Interrogate the operand information bits. B-box indicator bits --> BOXSW.

4. If the operand does not have a B-box associated with it go to the Return Exit.

5. If the B-box associated with the operand is identical to the last encountered B-box go to the Return Exit.

6. Generate a LOAD INDEX REGISTER command.

7. Go to the Return Exit.

BSCANE: Variable type test.

1. Zero the load command switch (BLODSW).
   Zero the variable type switch (BVARTP).

2. Set pseudo accumulator type switch. 7777 --> ACCTYP.

3. Interrogate the relative operand location.
   a. If it is a pseudo accumulator operand go to the Return Exit.
   b. If it is an Erase operand go to (4).
   c. If it is a Normal Function Erase operand go to (11).

4. Zero the proper bit in erasable bit indicator (ERSBIT).

5. If this is a Boolean Statement go to (6), if not go to (9).

6. Change the relative erase operand location to an absolute two word operand location.

7. Zero the B-box indicator (BOXSW).

8. Go to (14).

9. Set the variable type switch (BVARTP) to minus 0 (7777).

10. Go to the Return Exit.

11. Operand bank setting --> BANK.

12. If the operand needs a 3-word command go to (15).

13. If the operand bank setting is zero go to (16).

14. Go to the Return + 1 exit.

15. Go to the Return + 2 exit.

16. If the absolute operand location is 32 go to (15), if not go to (14).

BSCANF: Function Erase Command Generator.

1. Return jump to BSCANL.

2. Preset command locator instruction.

3. If this is not a STORE command go to (12).

4. If a mode conversion is not necessary go to (12).

5. Generate a CONVERT ACCUMULATOR command.

6. Go to (12).

7. If this is a LOAD command go to (12).

8. If this is not a Boolean statement go to (12).

9. Increment the "Number of Words in Command" count (BNOWDS) by 1.

10. Increment the command locator instruction by 13.

11. Go to (10).

12. Compute the proper mode indicator bit for the current command.

13. Store the command in the output area: BWORD1, BWORD3.

14. Store the relative operand location in the output area: BWORD2, BWORD4.

15. If this is a Boolean statement (except for LOAD and STORE command) go to (14).

16. BWORD1 + (BWORD2) --- BOWRD1, BWORD3.

17. If this is not a 3 or 4 word command go to the Return Exit.

18. Compute a 3 word command (op. code 75) and place it in the output area BWORD1, BWORD2.

19. Increment the Number of Words In Command count (BNOWDS) by 2.

20. Go to the Return Exit.

BSCANG: Normal operand command generator.

1. Return jump to BSCANL.

2. Preset command locator instruction.

3. If this is not a STORE command go to (7).

4. If a mode conversion is necessary increment the command locator instruction by 2.

5. Set mode bit for the command.

6. Go to (12).

7. If this is a LOAD command go to (11).

8. If this is not a Boolean statement go to (11).

9. Increment the command locator instruction by 31.

10. Go to (12).

11. Increment the command locator instruction if a LOAD AND CONVERT command is to be generated.

12. Send the Operation Code + Bank Setting + Index Register Option Bit to the output area (BWORD1).

13. Send the absolute operand location to the output area (BWORD2).

14. Go to the Return Exit.

BSCANH: Parentheses Group Determination.

1. Initial scan location --> BTEMP1.
2. Zero right parentheses switch (BRTPAR),

   Zero function switch (BFUNSW),

   Zero end switch (BENDSW).
3. Interrogate the lower order 4 bits of (BTEMP1).
4. If they indicate an arithemtic operator go to H0.
5. Use the low order 4-bits of (BTEMP1) to set a 10 way switchboard.

4-BITS	MEANING	ROUTINE
06	END	H6
07	)	H7
10	,	Return Exit
11	PSEUDO END	Return Exit
12		Return Exit
13	FUNCTION	H13
14	(	H14
15	START	H15
16	2-word operand	H0
17	3-word operand	H0

H0: Increment location counter.

1. Increment location counter (BTEMP1) by 1.
2. Go to BSCANH,3.

H6: End indicator.

1. Set end switch (BENDSW).
2. Go to the Return Exit.

H7: Right parentheses indicator.

1. Set the right parentheses switch (BRTPAR).
2. Go to H10.

H13: Function indicator.

1. Set the function switch BTEMP1 --> BFUNSW.
2. Go to H14,2.

H14: Left parentheses indicator.

1. Zero the function switch (BFUNSW).
2. The previous parentheses group starting location goes to the encompassing group starting location. BINITA --> BINITC.
3. The current scan location goes to the parentheses group starting location. BTEMP1 --> BINITA.
4. Go to H0.

H15: Start indicator.

1. The statement starting location goes to the encompassing group starting location. BTEMP1 --> BINITC.
2. Go to H14,3.

BSCANJ: Operator determination.

1. Zero the Preceding Operand switch (BVARCD).
2. Use bits 1, 2, 3 of (BSCNLC) to set an 8 way switchboard.

3-BITS	MEANING	ROUTINE
0	BLANK, **	J0
1	*, /	BSCNJY
2	*, -	J4
3	END,)	J6
4	), PSEUDO END	J6
5	FUNCTION	J14
6	(, START	J14
7	OPERAND	J16

JO: Increment location counter.

1. Increment the scan locator (BSCNLC) by
   a. +1 for forward direction.
   b. -1 for backward direction.
2. Go to BSCANJ,2.

J14: Left parentheses and start indicators.

1. Increment the Return Exit address by 1.
2. Go to BSCNJY.

BSCNJY: Save scan location.

1. Save the scan location in the operator location switch (BSCNLC --> BOPLC
2. Go to the Return Exit.

J16: Operand indicator.

1. Set Preceding Operand switch (BVARCD).

2. Go to JO.

BSCANK: Mode Equality Test.

1. Compare the mode of the operand with the mode of the parentheses group in which the operand is contained. If they are
   a. Equal, Zero the mode equality switch (EMDSW).
   b. Unequal, set to 2 the mode equality switch (EMDSW).
2. If this is not a Boolean statement go to the Return Exit.
3. Zero the mode equality switch (EMDSW).
4. Go to the Return Exit.

BSCANL: Relative operator Determination.

1. If the current operator is not a divide operator go to (3).
2. If the direction of the scan is
   a. Forward : the operator remains a normal divide.
   b. Backward : the operator changes to an inverse divide.
3. If the LOAD switch (BLODSW) is not set go to the Return Exit.
4. If the Mode Equality switch (BMDSW) is not set go to the Return Exit.
5. Change the LOAD operator to a LOAD and CONVERT operator.

BSCANM: Pseudo Accumulator and Erasable Arithemtic.

1. Return jump to BSCANL (JPR BSCANL).
2. Preset an arithemtic command locator.
3. If this is a Pseudo Accumulator command go to (10). If an Erase command go to (4).
4. Set the Erase-Function Erase operand switch (BOPTST).
5. If the mode of the operand and the mode of the encompassing subgroups are equal go to (8).
6. Generate a CONVERT ACCULULATOR command.
7. Increment the Number Of Words In Command count (BNOWDS) by 1.
8. Set the mode indicator bit for the generated command.
9. Go to (12).
10. Decrement by one the Pseudo Accumulator counter (ACCN)
11. Set the mode indicator bit for the generated command.
12. Generate the proper command.
13. Go to the Return Exit.

BSCANN: Store Pseudo Accumulator Exit.

1. If this is not a Boolean statement go to (5).
2. Generate a STORE ERASE command (JPR BSCANS).
3. Reset the initial operator of the subgroup in the statement scan list.
4. Go to the Return Exit.
5. If the previously generated command was an Erase or Function Erase command go to (9).
6. Generate the proper bits for the Pseudo Accumulator bit indicator (BNSW).
7. Increment the Pseudo Accumulator counter (ACCN).
8. Go to the Return Exit.
9. If this is a Store In Operand Register command go to (8).
10. Generate a STORE PSEUDO ACCUMULATOR command (JPR BSCANM).
11. Go to (7).

BSCANO: Operand Interrogation.

1. Increment the scan location (BSCNLC) to enable operand references.
   a. +1 for a forward scan.
   b. -1 for a backward scan.
2. Determine the operand location (JPR BSCANQ).
3. Do a mode equality test (JPR BSCANK).
4. Do an index register test (JPR BSCAND).
5. If the operand does not need converting to to (10).
6. Generate a LOAD AND CONVERT command (JPR BSCANR).
7. Do a pseudo accumulator store check (JPR BSCANN).
8. Generate the necessary (dummy) command (JPR BSCANM).
9. Go to (12).
10. Check the operand type (JPR BSCANE).
11. If the operand type is a
    a. Pseudo Accumulator or Erase go to (8).
    b. Normal operand do a return jump to BSCANG.
    c. Function Erase do a return jump to BSCANF.
12. Zero the operand indicator. Zero the operator indicator.
13. Go to the Return Exit.

BSCANQ: Operand Location Determination.

1. If the contents of the operand scan location is not equal to zero go to the Return Exit.
2. Increment the scan locator by
   a. +1 for a forward scan.
   b. -1 for a backward scan.
3. Go to (1).

BSCANR: Load Driver Routine.

1. If this is not a Boolean statement go to (3).
2. Preset a LOAD (in contrast to a LOAD NEGATIVE) operator.
3. Determine the operand location (JPR BSCANQ).
4. Do an operand conversion check (JPR BSCANK).
5. Do an index register test (JPR BSCAND).
6. Check for operand type (JPR BSCANE).
7. Set operator switch (BOPRTR) for LOAD command.
8. If the operand type is
   a. Pseudo Accumulator or Erase operand do a return jump to BSCANM.
   b. Normal operand do a return jump to BSCANG.
   c. Function Erase operand do a return jump to BSCANF.
9. Zero the operand indicator in the scan list.

   Zero the load command switch (BLODSW).
10. Go to the Return Exit.

BSCANS: Store Erase Routine.

1. Preset relative erase count to one.
2. Store an erase operand indicator in the scan list.
3. Starting with the low order bit of the erase bit counter (ERSBIT), check to determine if there are any more erase locations available. If there are, go to (6).
4. Load "No More Erase" indicator.
5. Go to TILT.
6. Update the operand indicator in the scan list.
7. Generate a STORE ERASE command (JPR BSCANM).
8. Go to the Return Exit.

BSCANT: Parentheses Group Store Check.

1. Zero the left parentheses indicator.
2. If this is a Boolean statement go to (4).
3. If there is a ** operator adjacent to the parentheses group go to (4).
4. Go to BPGG.
5. Do a subgroup scan (JPR BSCANA).
6. If the contents of the subgroup are not completely encoded go to (8).
7. Go to BPGDB.
8. Do a mode comparison check (JPR BSCANU).
9. Save previous mode (BMDSVI --> BMODE).
10. Do a subgroup scan (JPR BSCANA).
11. If, in the subgroup scan, one of the following conditions existed go to (4), if not go to the Return Exit.
    a. If there exists a second parentheses group in the subgroup.
    b. If the next executeable operator in the subgroup is not adjacent to the currently evaluated parentheses group.

BSCANU: Mode Comparison Check.

1. Save the current mode (BMODE --- BMDSV1).
2. If the current mode is the same as the statement go to the Return Exit.
3. Scan for the mode of the encompassing group. (JPR BSCANC).
4. If the current mode is identical to the mode of the encompassing group go to the Return Exit.
5. Generate a CONVERT ACC. command.
6. Go to the Return Exit.

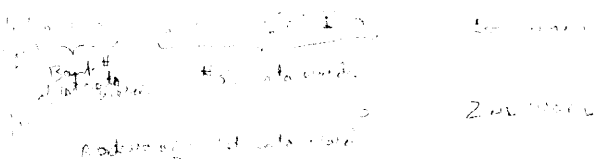
160 - A FORTRAN - COMPILER PASS 2

At the completion of the first pass of a 160-A Fortran compilation all source statements have been read and an identifier list - IDLIST, a table of information pertinent to the compilation, prepared and retained in memory. If Pass 1 has not detected any errors in the source program tape 2 contains a preliminary version of the object code. Otherwise tape 2 contains the appropriate diagnostic messages. The second pass creates some additional object code, if this is necessary, and produces a list of diagnostics and a memory map. It loads the preliminary object code from tape 2 and makes the appropriate corrections using information from the identifier list. Finally, the second pass determines which library functions and interpreter modules are required for the execution of the source program and loads them. Control is then turned over to the interpreter and the execution of the object code is begun. The following paragraphs provide a detailed description of these processes.

The Preliminary Object Code

During the first pass, object code, in the form of 160-A Fortran intermediate language instructions, tailored to the intent of the source program, is produced and written onto tape 2, and this process is continued until either all source statements have been processed or until a recognizable source program error is encountered. If there are no such errors, tape 2 will contain, at the end of pass 1, the preliminary object code. The object code, exclusive of the object code for format statements, is on tape 2 as binary records, each 80 decimal words in length, with the first word giving the number of words of object code in the record. During the first pass object code is accumulated in an 80 word buffer area. When the buffer is full it is written onto tape 2. However, when a FORMAT statement is encountered, the format statement is then written on tape 2, and the process continues. Since format statements are handled in the same fashion as constants and variables as regards memory allocation, it is possible to assign them memory locations during the first pass. They are then written onto tape 2 in a format which facilitates loading them into memory during pass 2. This format is a sequence of binary records each 80d in length. The first word of the record has the number of data words in the record in bit positions 0-6, the bank number of the address of the first data word of the record in bit positions 8-10, and a 1 in bit position 11. The second word contains the address of the first data word, which is the third word of the record. The further restriction is made that no record contains data for two different banks.

The preliminary object code as produced by pass 1 is not in a form that can be executed correctly by the interpreter. First, if any pseudo-b-boxes



and UP-instructions are required they have not yet been produced; second, there are a number of intermediate language instructions which refer to locations within the object code, and these locations cannot be known until the completion of pass 1.

More specifically the following intermediate language instructions are produced by pass 1 in the following preliminary form (b stands for bank of, A stands for address of):

1. Load Index: LXB (b-box)
   A (b-box) is produced in
   the form LXb (Id)
   A (Id) (where Id is the address of the corresponding pseudo b-box idlist entry.

2. UP instruction: GO4 b (UpI)
   A (UpI), where UpI, the address of the UP-subroutine, is produced in the form
   GO4 b (IdI)
   A (IdI), where IdI is the address of the idlist entry for the integer variable I.

3. GO TO instructions: GØO b (ØS)
   A (ØS), where ØS
   is the object code location of a statement number, an address in variable storage corresponding to a variable n in a GO TO n statement,
   is produced as GØO b (IdI)
   A (IdØ), where IdØ is
   the idlist address of a statement number entry, or a label (n) entry.

4. The store instructions which initialize the incrementation counters for a DØ loop:

   ```
   CA    M1
   TS1   I
   IS    M2
   ID    M3 - (may be absent)
   GØ1   M4
   CA    M3
   GØ    M4 + 2, where
   ```

   M1, M2, M3 are the parameters from the statement DØ n I = M1, M2, M3, M4, = M1-M2/M3 and is located following the corresponding TRM INCR instruction. The preliminary object code is produced as

   ```
   CA    M1
   TS1   I
   IS    M2
   ID    M3
   771   b (IdS)
   ```

```
                    A (IdS)
              CA    M3
              771   b (IdS)
                    A (IdS), where IdS is the
```
idlist address of a TRM INCR entry.

5. Transfers to library functions TRAn where n is the transfer vector number of the function in question is produced as TRAn' where n' is assigned during Pass -1. The first encountered library function is assigned n' = 1, the second n' = 2, etc.

6. For the Fortran statement IF (algebraic Statement) S1, S2, S3, for which the object code is:

```
(Algebra)
TRM  IF
b (S1), b(S2), b(S3), 0
A (S1)
A (S2)
A (S3), where
```

S1, S2, S3, are statement numbers, Pass 1 produces

```
(Algebra)
TRM  IF
b (IdS1), b(IdS2), b(IdS3), 0
A (IdS1)
A (IdS2)
A (IdS3), where
```

IdS1,.. are the addresses of the idlist entries for the corresponding statement numbers.

7. For the Fortran statements

```
IF  OVERFLOW S1, S2
IF DIVIDE CHECK S1, S2
IF SENSE SWITCH S1, S2
```

for which the object code is

```
TRM IFOV
b (S1), B (S2), 0, 0
A (S1)
nA (S2), where S1, S2
```

are statement numbers. PASS 1 produces

```
TRM IFOV
b (IdS1), b(IdS2), 0, 0
A (IdS1)
A (IdS2), where A (IdS1) is the
          address
```

of the idlistmentry for statement number S1, etc.

8. For the computed go to statement GO TO (S1,...SM), i for which the object code is

```
TRM CMPTD
M, b (I), A (I), b (S1)
```

9 bits .

.

b (SM)

A (SM), where the B1 are statement numbers, Pass 1 produces

TRM CMPUTD

M, b (IdS1)

A (IdS1)

.

.

.

b (IdSM)

A (IdSM), where the IDS1 are the addresses of idlist entries for the corresponding statement numbers.

9. For input-outout: TRM I/O-IN, or I/O-OUT

m n x b (F)

3 bits 5 bits 1bit 3 bits

A (F) or binary tape function,

where m is the unit number, n the vector number of the required I/O driver routine, x a binary flag, and F a format statement number, is produced as

TRM I/O-IN, or I/O-OUT

m n' x b (IdF)

3 bits 5 bits 1 bit 3 bits

A (IdF) or binary tape function

where n' has the same meaning as in 5, and IdF is the idlist entry for the format statement F.

LOW CORE FROM PASS 1

In addition to the identifier list and the object code on pate 2, Pass 1 also leaves information in the following low core cells.

IDBANK Address of last-made idlist entry
IDLAST

BNKCON Address of last assignment in constant-variable storage
CONLST

BANKS - SIC (number of banks in computer)

OBBANK Last relative object code location, i.e., the last location
OBLAST + 1 that the object code would occupy if it began at location 1 in bank o.

TABLC1 - If bit 0 = 1, A**B is used
If bit 1 = 1, A**1 is used
If bit 2 =1, I**J is used.

INTERP - If bit 11 = 1, floating point is used
If bit 10 = 1, format control is used
If bit 9 = 1, boolean arithmetic is used

and bits 0-5 give the number on the systems tape of the desired memory map routine. (0 = suppress memory map).

DIAGNS = 0 means no pass 1 diagnostics.

≠ 0 means pass 1 diagnostics.

ERASEL number of function erasable locations required for execution of the source problem (1/3 the number of storage locations required.)

SEQUENTIAL DESCRIPTION OF THE OPERATION OF PASS 2.

1. Preliminaries

At the beginning of Pass 2, if diagnostics have occurred, then DIAGNS ≠ 0. If this is the case the output routine is "blocked" so that any subsequent operation calling for writing on tape 2 are not performed.

Since the output buffer may be partially full, the object code is written onto Tape 2 followed by an end of file. Pass 2 then reads the following three records from the systems tape.

a. CALTBL, a table of information determing which library routines call other library routines.

b. INTBL, a table, 64_d words in length, whose nth entry is the length of the library function with transfer vector number n.

c. LIBTBL, a table, of length 180_d, containing 60 3 word entries, 6 BCD characters/entry, where the nth entry is proper FORTRAN name of the library function with transfer vector number n-4. (Vector numbers 1-4 refer to the interpreter modules.)

2. Determination of required library functions, storage allocation library functions and interpreter modules, and computation of the first object code locations.

Pass2 assigns storage to library functions and interpreter modules in the order in which the routines appear on the tape, subject to the following conditions:

a. The first interpreter module or library function following the interpreter control section (CONTRL) (which is always present) begins at the last word address + 2 of CONTRL. Each succeding interpreter module begons at the last word address + 1 of the previous one. The reason for this anomaly is that CONTROL is read as one binary record from the systems tape, after all other parts of the object program have been loaded, and this type of read zeroes the last cell loaded + 1.

b. Each library function begins at either the last word address + 1 or + 2 of the preceding routined, the additive 1 or 2 being chosen to make the resulting address even if the function goes in bank 0, and odd in bank 1. (odd in the sense that the binary number has a 1 in bit position 0, i.e. -1 is even)

In no case will a library function be assigned storage in both bank 0 and 1. If a function will ovdrlap, it is instead assigned storage beginning at location 0001 in bank 1, the excess memory in bank 0 being "wasted".

The information as to which interpreter modules and library functions are required by a particular compilation is left in the following form by Pass 1.

a. Interpreter modules: the use of the interpreter modules is determined by the upper 3 bits of low core location INTERP.

A module is required if its corresponding bit is 1.

bit 11 - floating point arithmetic

bit 10 - format control

bit 9 - boolean arithmetic

b. Power routines, A**B, A**I, I**J. The use of these routines is determined by the lower 3 bits of low core location TABLC1, a routine being needed if its corresponding bit is 1.

bit 2 - I**J

bit 1 - A**I

bit 0 - A**B

c. Library functions explicitly refered to by the source program. The use of these library functions is determined by their appearance in an IDLIST entry. They are identified by their BCD FORTRAN names.

Pass2 then prepares from this information a condensed table--5 low core words beginning with SUBRTN--which can be retained throughtout compilation and referred to when necessary. The bits in this table refer in order to the transfer vector numbers of the library functions with bit 0 of SUBRTN referring to transfer vector number 5. This table is intially cleared. Then, if format control (FORMAT) is required the bit corresponding to input control (INPUT, transfer vector number 45) is set to 1. Next the bits in TABLBI corresponding to the power routines are moved to their proper place in the table, and if, A**B is called, the bits for LOGF and EXPF are set. Now Pass2 begins to search the IDLIST for library function entries. When one is found, it is moved to low core, and its BCD name compared to those in LIBTBL. If a match is found, the proper bit is set to 1 in the table, and a search of CALTBL is made to see if this function calls and other library functions, and, if so, also sets the proper bits for these functions.

Pass2 now uses the condensed table beginning at SUBRTN, and INTERF to allocate storage for these routines.

This is done sequentially, subject to the restrictions stated above. The lengths of the routines are obtained from INTBL.

The final length of the interpreter and library functions is then decremented by 1 and stored in locations BNKINT, INTLTH as a 15-bit quantity. The decrement of 1 is to compensate for the fact that the first word of object code is assigned relative location 00001 instead of 0 0000. Since the object code begins immediatly following the interpreter and library functions, adding the increment now in BNKINT, INTLTH to a relative object code location gives an actual object code location. Pass2 now uses this information to compute the current actual last object code location + 1, leaving this 15-bit number in OBLANK OBLAST. At this point, and at each subsequent point where OBBANK, OBLAST are increased, a test is made to see that the last object code entry is not so close to the last data / constant storage entry as to not leave room for a sufficient number of function erasable locations for the excution of the program. If this is the case location LONGSW is set non-zero so that the message OBJECT CODE EXCEEDS MEMORY appears on the memory map.

3. Generation of Pseudo B-Boxes

At the time of excution of the source program, the addresses of elements in arrays are determined as a fixed base address and 15-bit variable quantity which is added to the base address. The base address is in the instruction being excuted. It is computed by the compiler, and is determined by the constant information in the source program subscript expression to which it refers. The variable quantity is saved in the pseudo b-box for this subscript expression. It depends upon the values of the integer variables involved in the subscript expression and is kept current as these values change by means of the interpretive UP instruction, discussed in the next section.

The information necessary for the generation of a pseudo b-box by Pass2 is left by Pass1 as a pseudo b-box IDLIST entry. Pass2 ganerates the pseudo b-box, assigning as its first object code location the current value of OBBANK, OBLAST, and puts it out with the object code. The object code location is then saved in the IDLIST entry so that all LX (load index) instructions can be co - rrected at a later time. Finally OBBANK, OBBLAST are increased by the length of the pseudo b-box just generated. When all pseudo b-box IDLIST entries have been processed, PASS2 begins the generation of UP-subroutines.

The pseudo b-box IDLIST entry has the form

```
2; level; length of entry-3 (1,4 or 6)
16      ; b (F1) ; b (i1)
          A (F1)
          A (i1)
 b (F2); b (i2) ; b (F3); b (i3)    (missing if dim = 1)
           A (F2)                   (missing if dim = 1)
           A (i2)                   (missing if dim = 1)
           A (F3)                   (missing if dim = 2)
           A (i3)                   (missing if dim = 2)
```

Where F1, F2, F3 are multiplicative 15-bit quantities, and i1,i2, i3 are the idlist locations of entries for the integer variables which appear in the subscript expression. These integer variables do not affect the form of the pseudo b-box; they affect only the associated UP subroutine. The pseudo b-box; generated by Pass2 has the form

```
                0
                0
b (F1), b (F2), b (F3), dimension (1,2, or 3)
                A (F1)
                A (F2)
                A (F3)
```

Where of course, only those Fi up to the dimemsion of the b-box appear. The idlist entry is then altered by replacing the 15-bit quantity b(F1), A (F1) by OBBANK, OBLAST.

4. Generation of UP-subroutines

The pseudo b-boxés are kept current during excution in the following way: in a subprogram in which dimensioned variables are present, interpretation instructions which might change the value of an intefer variable, I, are followed by the interpretive instruction UP (A(UI) where the address A(UI) refers to the UP-subroutine for I. The UP instruction causes each pseudo b-box involving I to be increased or decreased by the proper amount.

The UP subroutine for I has the following format:

X;0; 2; b (I)		Object code location of I. X = 0, unless
	A (I)	A (I) is in erasable storage in which case
		X = 4
	m	value of I at last
	n	UP (A(UI) instruction

D1; 0 ; 0 ; b (BBX1)
A (BBX1)
D2; 0 ; 0 ; b (BBX2)
A (BBX2)
.
.
.
Dk; 0 ; 7, b (BBXk)
A (BBXk)

Where BBXj is a b-box in which I occurs in dimension Dj. A 7 in bits 3-5 of an entry , e. g. Dk;0;7;b(BBXk)

A (BBXK) signifies that this is the last entry in this UP-subroutine. If the object code location of I, b(I), A(I), is in function erasable storage, then the first word of the UP-subroutine is 4; 0; 2; b(I) (=0)

Pass2 begins this procedure by searching the idlist for integer variables. When an integer variable is found the idlist is searched for pseudo b-boxes in which it is involved. If the integer variable is not involved in any pseudo b-box, no UP-subroutine is generated and an idlist entry of the form

```
0;0;0; length - 3
22   ; b(I) ;   7
       A(I)
       7777
```

is made at the beginnig of the IDLIST.

If this integer variable does occur in some pseudo b-boxes the information from all such b-boxes is collected and an UP-subroutine generated and written on the end of the object code.

An IDLIST entry of the form

```
0; 0; 0; length - 3
2 1 ; b(I) ; OBBANK
      A(I)
```

OBLAST is made at the beginning of the idlist. Here OBBANK, OBLAST refer to the 15-bit address in those locations, which will now be the starting address of this UP-subroutine. OBBANK, OBLAST are then increased by the length of the UP-subroutine.

When all integer vafiables have been processed and their UP-subroutines generated, generation of the preliminary object code is complete. PASS2 now adjusts OBBANK, OBLAST by 1 or 2, if necessary, to insure that there are an itegral number of function erasable locations in the bank of OBBANK, i. e. that 4096D - OBLAST is divisible by 3. The object code buffer is also written onto tape 2, followed by an end-of-file.

5. Diagnostic Messages

Pass2 now reads the next record on the systems tape, the diagnostic messages. This record overlays those parts of the code which have been discussed so far. A search of the idlist is then made for unassigned labels.

An unassigned label is either a statement number, FORMAT statement, or SUBROUTINE name with the relative address 0,0000, or a library function with bit 10 = 0 in the first word of the corresponding IDLIST entry. (This bit is set to 0 earlier in Pass2 if the library function is not one of those on the systems tape.)

If no unassigned labels are found Pass2 proceeds to test for diagnostics, otherwise Pass2 sets location MAPSWC, which is initialized to -1, to 0, and brings in the memory map I/O driver from the systems tape by means of a return jump to subroutine MAPIN. This subroutine brings in the memory map output driver and positions the systems tape at the beginning of file 3. If the option of supressing the memory map has been made, an error stop will occur at 0,5772. Compilation can either be stopped at this point, or the number of a desired output driver can be entered into the A-register and compilation continued. The heading UNASSIGNED LABELS is then put out followed by a list of the BCD identifiers of the unassigned labels.

After all unassigned labels have been treated a test is made to see if low core locations DIAGNS is 0. If so Pass 2 proceeds to the normal memory map procedure. Otherwise Pass 1 has detected errors in the source program, and there must be output in a usable format, MAPSWC is now increased by 1. If the result is not zero the memory map output driver is already present in memory. Otherwise it is brought in by a return jump to MAPIN. The diagnostics headings are then output, and tape 2 is rewound to get the diagnostic messages left there by Pass 1. These diagnostic messages are in the form of binary records of length 80d in which only words 2-3 are significant. These four words contain the following information:

word 2 b(Id)
word 3 A(Id)
word 4 increment
word 5 error

where b(Id), A(Id) is the IDLIST address of the first label preceding the statement in which the error occurred and the increment is number of statements from the last label statement to the offending statement. The error is designated by a number _ 60d. Pass 2 then gets the statement label from the IDLIST, and a descriptive phrase describing the error from the table of diagnostic messages and outputs this information, one line for each error.

There is one further diagnostic test which is made after all diagnostics are out and before the memory map is prepared. This is a test (LONGSW = 0) to determine whether or not the object code has exceeded the available memory. If this condition is present the appropriate message will appear before the memory map.

If this condition is present, or if there have been diagnostics or unassigned labels, DIAGNS is set non-zero at this point. Compilation will then stop (an error stop at 5432) after the memory map is out.

6. Memory Map

The memory map is essentially a listing, by type, of the contents of the IDLIST. It is produced in a completely straight forward fashion, i. e. a search of the IDLIST is made for a particular type of entry. If one entry of a desired type is present, then the appropriate heading is output, followed by a list of entries of that type. The last line of the memory map is headed ERASABLE STORAGE. The numbers given are respectively the first word of function erasable storage and the first word of constant / data storage.

7. Condensation of the Idlist

Pass2 is now ready to begin the correction of the preliminary object code. Since this object code may occur and and all of the locations after its known first location, it is necessary to save that information in the IDLIST which is required in these corrections in the lower part of memory. What is done is that the Idlist entries for statement numbers, subroutines, pseudo b-boxes, format statements, both types of UP-subroutines, plus entries for Do loop initialization are reduced to the following condensed form: x ; x ; b(Ob) ; b(Id)

A(Id)

A(Ob), where Ob is the Object code location, and Id the Idlist location of the statement in question. It is not necessary to retain the type of statement. These condensed entries are then moved to the lowest currently available part of memory, beginning immediately after the coding which updates the object code. Condensed idlist information is accumulated until it exhausts the idlist, or until it reaches the smaller of

Constants and the coding for assigned GO TO'S are written onto tape 2 during this process in the following formats:

1. Floating point constants:

```
1 1 ; 0 ; b (OB)  (Ob is object code location)
        A (Ob)
                  (3 words)
```

2. Integer Constants:

```
12; 0 ; b (Ob)
        A (Ob)
        value      (2 words)
```

3. Assigned GO TO'S (ASSIGN I TO ...)

```
12 ; 0 ; b (Ti)     (Ti) is storage
          A (Ti)    location for the GO instruction
GO (=76)  b (I)     (I) is object code location
          A (I)     of statement number I.
```

When the IDLIST is exhausted, the remaining constants in the output buffer are written onto tape 2, followed by an end-of-file. The completed library function transfer vector is now written onto tape 2, followed by an end-of-file.

8. Banks 1 through the last available are cleared to 0000.

9. Correction of Preliminary Object Code

Tape 2 is now rewound, and the first file, containing the preliminary object code and format statements from Pass1, is loaded. This code now occupies its final machine locations. Now knowing the beginning of the object code, the length of each instruction, and which types of instructions require corrective action (the types described earlier always need correction) it is a simple matter, using the condensed IDLIST and the table of true library function transfer vector numbers, to correct the object code.

10. Loading of Library Functions and Interpreter Modules

Pass 2 now loads Pass 2, Part 2 from the systems tape, overlaying Pass 2 Part 1 beginning at location 0 0402, and transfers control to this location. Then the pseudo-b-boxes, UP-subroutines, constants, and assigned GO TO's are loaded into their proper memory locations, and the completed library function transfer vector brought into a temporary location following the coding of Pass2 Part 2, but not exceeding the memory which will be later occupied by the control section of the interpreter. A test is now made on selective jump switches 1 and 2, and if these are set the appropriate object code lister is read from tape 1, and a listing of the object code produced. Execution is not possible, otherwise tape 1 is positioned at the beginning of file 4 and tape 2 is rewound.

At this time, the library function table in the five words beginning at SUBRTN, and the interpreter module information in low core location INTERP is still correct. Pass 2, Part 2 now goes sequentially through the transfer vector numbers, beginning with 2 (= floating point arithmetic) checking each one to see if it is required by this compilation. If the routine is required Pass 2, Part 2 searches forward on tape 1 until a routine with this transfer vector is detected. If no such routine is present there will be an error stop at 0 0722. If the proper routine is found it is then loaded relocatably beginning at the address for this routine given in the transfer vector, using this address also as a relocation incrememt. The routines in the fourth file are OSAP binary format, and this relocation is accomplished in the usual fashion when the routine in question goes in bank 0. For those routines which go in bank 1 there is the complication that those words which need to be increased by the relocation increment are usually part of the 15-bit address in a two-word interpretation instruction.

Hence for routines which go into bank 1 the loader not only adds the relocation increment to indicated words, but also increments the preceding word by one.

The transfer vector is now moved to the locations beginning at 0 0100. The interpreter control section is then read in as one binary record (file 5), and compilation is complete. If no selective stop switch had been set execution will begin.

Generation of UP subroutines

Symbols: Same as for pseudo B-boxes

1. Set TABLC1 = TABLC2 = IDLAST.
2. Set flag for 1st time through IDTYPE.
3. Return jump to IDTYPE.
4. -0 means IDLIST exhausted exit. If not 02 back to 3.
5. Make IDLIST entry for this UP subroutine.
6. Save TABLC2. Set TABLC1 = TABLC2 = IDLAST.
7. Set flag for 1st time through IDTYPE.
8. Return jump to IDTYPE.
9. If -0 go to 13, if not 17 back to 8.
10. Return jump to IDMOVE.
11. If I not mentioned in this B-box go to 8.
12. Move information about I into UP subroutine area. Go to 8.
13. Finish UP subroutine.
14. Return jump to OUTBUF.
15. Set TABLC1 = TABLC2 = Value of TABLC2 saved in 6.
16. Back to 2.

Output constants and condense IDLIST

1. Initialize IDTYPE

2. Return jump to IDTYPE

3. If - 0 goto 4., otherwise a transfer vector as follows:

   11,12 (constants) write constant on output tape, go to 2.

   1,2,3,4,5,6,7,10 (variables) and 15 (library functions) go to 2.

   21,20,13,14,17 (assign, up subroutine labels, subprogram names, and pseudo b-boxe
   create a condensed IDLIST entry of the form

   ```
   0,0,b (OBJECT CODE LCN) b (IDLIST LCN)
       A (IDLIST LCN)
       A (OBJECT CODE LCN)
   ```

   go to 2.

   22 (unused UP-SUBROUTINE) created condensed idlist entry of the form

   ```
   0,0,0, b(IDLIST LCN)
       A (IDLIST LCN)
          7777
   ```

   go to 2.

4. Output last block of constants.

5. Exit.

Load and update object code.

LOADER: Subroutine to load object code.

FIX : Subroutine which replaces IDLIST location with correct object code location.

OBJECT: Current object code location.

1. Load object code beginning at end of interpreter + library.
2. Initialize OBJECT to first object code location.
3. Get instruction in OBJECT, then go to a transfer vector as follows:

   0, 2-43: OBJECT = OBJECT + 1, go to 3.

   44 (LIX) return jump to FIX, OBJECT = OBJECT + 2, go to 3.

   45-75 OBJECT = OBJECT + 2, go to 3.

   76-77 (UP, GO) return jump to FIX, OBJECT = OBJECT + 2, go to 3.

   01 (TRM) test lower 6 bits of instruction.

      if RETURN; OBJECT = OBJECT + 1, go to 3.

      CMPUTD; OBJECT = OBJECT + 3,

         1a. Return jump to FIX.

         1b. OBJECT = OBJECT + 2, m = m - 1.

         1c. If m = 0 go to 3 otherwise back to 1a.

      IF; OBJECT = OBJECT + 1, C(OBJECT) = 8*C (OBJECT)

         2a. Return jump to FIX.

         2b. C(OBJECT) = 8*C(OBJECT, if not 3rd back to 2a.

         2c. OBJECT = OBJECT + 4, go to 3.

IFOV, IFDVCK: OBJECT = OBJECT + 1, C(OBJECT) = 8* C(OBJECT)

3a. Return jump to FIX.

3b. C(OBJECT) = 8* C(OBJECT) if not 2nd, go to 3a.

3c. C(OBJECT = 8* C(OBJECT),

3d. OBJECT = OBJECT + 2, go to 3.

CALL : OBJECT = OBJECT + 1, return jump to FIX, OBJECT = OBJECT + 2n, go to 3.

4. Exit.

Compiler Pass II

160-A FORTRAN Listing

1. Test low core location LISTRN to see which output medium, if any, has been chosen for the listing. If no listing has been opted, exit. Otherwise bring the appropriate output routine in from the systems tape.
2. Set TABLCi = TABLC2 = IDLAST.
3. Set HEDSWC = -1.
4. Return jump to IDPE.
5. If -0 go to 18, if not 2 (integer variable) back to 4.
6. HEDSWC = HEDSWC + 1.
7. If HEDSWC $\neq$ 0 go to 10.
8. Print INTEGER VARIABLE headings.
9. Enter information for integer variable.
10. Save TABLC2, set TABLC1 = TABLC2 = IDLAST.
11. Return jump to IDTYPE.
12. If not 22 or 23 (up subroutine) back to 11, if -0; error.
13. Compare to integer variable if no match, back to 11.
14. Enter the up subroutine information in the line.
15. Return jump to output subroutine.
16. Set TABLC1 = TABLC2 = value saved in step 1.
17. Return to 4.
18. Set TABLC1 = TABLC2 = IDLAST, HEDSWC = -1.
19. Return jump to IDTYPE.
20. If -0 go to 24 if not 01 (floating point variables) back to 19.
21. HEDSWC = HEDSWC + 1, if HEDSWC $\neq$ 0 go to 23.
22. Output FLOATING VARIABLES heading.

23. Output IDLIST information, back to 19.

24. The rest of the entries are generated in the same manner as the floating variables, above. For constants conversion will be required.

After all pertinent IDLIST information has been written the following will be done.

1a. Compute the length of erasable and output.

LISTER is a closed subroutine, entered by a JPR to the first location, which will write one 80 character record from location 100 in bank zero. If A is equal to zero upon entry, an end of file record is written. Characters are converted for output if necessary. All bank settings are assumed to be zero upon entry.

There are six routines available on the system tape which may be used interchangeably. A parameter is supplied to the system during operation which governs the choice of the routine to be called. All available routines exist in one file on the systems tape in binary format. The names of the routines and the equipment required are as follows:

CDLIST - 523 card punch (1610)

PCHFLX - paper tape punch

WRBCDC - 163, 164, or 606 magnetic tapes (unit 3)

LPRC - 1612 printer

W1607C - 1607 magnetic tapes (unit 3)

LP166C - 166 printer

Procedure: CDLIST

1. Enter by JPR. Is A = 0? Yes, go to 5.
2. Initialize counters. Zero card image area.
3. Convert BCD codes to Hollerith card image (80 characters).
4. Wait for card punch ready. Initiate buffer action to punch.
5. Set exit address and return. (A = return address).

Procedure: PCHFLX

1. Enter by JPR. Is A = 0? Yes, go to 8.
2. Initialize to punch an 80 character record from 100 to 220. Select paper tape punch and punch an upper case code. Is the first character a page eject code (BCD 1)? Yes, go to 9.
3. Add one to the line counter. Is a page eject required? Yes, go to 9.
4. Adjust record length to delete final blank characters. Is the entire record blank? Yes, go to 7.
5. Convert one character to flex codes. Punch character and necessary case codes. Is this the last non-blank character? Yes, go to 7.
6. Go to 5.
7. Punch a carriage return code (end of record).
8. Set exit address and return. (A = return address).
9. Punch six carriage return codes. Reset the line counter. Go to 3.

Procedure: WRBCDC

1. Enter by a JPR with A = entry parameter.
2. A = 0? Go to 11.
3. Initialize error counters; select even parity (BCD). Store (BCD) blanks in first 4 locations of buffer.
4. Write tape 4 (100 to 220). Status response = 0? Yes, go to 12.
5. End of tape indication? Yes, go to 9.
6. Is this the fourth try? Yes, go to 7. No, backspace go to 4.
7. Is this the fourth EOF? No, write EOF, backspace and go to 4.
8. Stop. Run go to 12.
9. Backspace, write 2 end of files, and rewind unload tape. Halt. Run go to 4.
10. Select even parity (BCD). Write end of file record.
11. Return.

Procedure: LPRC

1. Enter by JPR. Is A = 0? Yes, go to 5.
2. Select 1612 printer. Add one to the line counter. Is a page eject required? Yes, go to 6.
3. Does the first character request a page eject? (BCD1) Yes, go to 6.

3a Wait ready. Store blanke in first three locations of print buffer.

4. Print from 101 to 220 and advance paper one line.
5. Set exit address and return.
6. Eject the page. Reset the line counter. Go to 4.

Procedure: W1607C

1. Enter by JPR. Save the contents of the A register. Select write tape 4 in coded mode and wait for tape ready. Was an end of file record requested? Yes, go to 7.
2. Initialize counters. Add one to the line counter. Is a page eject required? Yes, go to 8.
3. Pack the 80 character record, two characters per word. Store packed BCD blanks in first two locations of buffer.
4. Write the record (from 100 to 150) and wait for tape ready. Is the end of tape indicator set? Yes, go to 9.
5. Is there a parity error? Yes, go to 10.
6. Set exit address and return. (A = exit address)
7. Write an end of file record. Wait for tape ready. Go to 6.
8. Set the first character to BCD 1 and reset the line counter. Go to 3.
9. Backspace, write 2 end of files, rewind tape, and halt. If restarted, go to 4.
10. Backspace the tape and wait ready. Is this the fourth consecutive error? No, go to 4.
11. Write an end of file record and wait ready. Is this the fourth consecutive error procedure? Yes, halt. If restarted, go to 6.
12. No, reset error counters and go to 10.

Procedure: LP166C

1. Is A = 0 upon entry? Yes, go to 5.
2. Initialize and set the first three characters of the print record to blanks. Pack the remaining record two characters per word. Wait for printer ready. Print from 100 to 150 and space paper.
3. Set exit address and return. (A = 0).

SECTION 3.1

160-A FORTRAN-Variable Length Interpreter

A given Fortran source program may well not require all the capabilities of the interpreter for its execution. For this reason, and since lack of space is always a problem, the 160-A Fortran interpreter will be written as several separate subprograms, all but one of which need not be present for the execution of a given source program. The compiler then prepares a suitable interpreter as a part of the compilation process, and this is done as follows:

First, during pass I of the compiler, notice is taken of the use of various portions of the interpreter, and this use is recorded by setting to 1 the appropriate bit of the upper 4-bits of a given computer word. No one knows how many modules there will be, but floating point arithmetic will be one of them and it will be assigned the left-most bit. The bit assigned to a given part of the interpreter also determines the position of this part relative to the rest of the interpreter. I think it will also be possible to use the rest of this word to signify the output medium for the memory map, and the information as to whether we compile and go, or dump the object code, etc.

In pass II, a transfer vector is prepared to provide entry into those parts of the interpreter which are used. This transfer vector is of fixed length and a given position in it always denotes the same routine. Right now this transfer vector is contiguous with the library function transfer vector although this is of course not necessary.

The transfer vector is prepared by pass II using a table of lengths which is read from the systems tape as a separate record.

The entry in the transfer vector is the first location used by this part of the interpreter in this particular case.

Since all the variable parts of the interpreter are loaded relocatably it is not necessary to provide for connections from them to the fixed part of the interpreter

SECTION 3.2

160-A FORTRAN INTERPRETIVE CONTROL SECTION

The interpretive control section is designed to execute the intention of the various interpretive commands (object code) generated by the 160-A Fortran compiler or used in library subroutines. It employs the following logic to identify a command, transfer control via a switchboard, encode the command, and return transfer to identify a subsequent command.

1. The bank and location counters (a 15-bit quantity) are restored to reference the next instruction address minus one.
2. The bank and location counters are incremented to reference the next instruction.
3. The upper six bits of the instruction are used to preset a 64 way switchboard.
4. Transfer is directed to one of the various command encoder routines via the switchboard described in (3).
5. The command encoder routines interprets the intention of the individual commands with, in general, reference to various necessary subroutines.
6. Return transfer is in general directed back to (1), (ARITH) or in some special cases as a GO TO command to transfer is directed to (2), (ARITHA).

The interpretive control section is divided into three classes of routines.

1. Routines designed to execute the intent of the various operation codes.
2. Special purpose routines needed as aids in implementing the first class of routines.
3. Routines (macros) designed to execute the intent of various Fortran Statements.

A complete description of the first class of routines will not be written. Instead, reference should be made to the attached write-up entitled "160-A Fortran Intermediate Language". A list of these routines with the operation codes implemented by the individual routines, will be listed.

A description of the second and third class of routines will follow in subsequent paragraphs.

Class 1 Routines

BOC00 :	Operation code	00	Halt and Proceed
BOC01 :	Operation code	01	Drop Out, Transfer to Macro
BOC02 :	Operation code	02	Transfer, Return Transfer
BOC50 :	Operation code	50	Transfer to Power
BOC03 :	Operation codes	03,04	Relative Transfer
BOC05 :	Operation codes	05,06	Positive Jump
BOC07 :	Operation codes	07,10	Negative Jump
BOC11 :	Operation codes	11,12	Zero Jump
BOC13 :	Operation codes	13,14	Non-Zero Jump
BOC44 :	Operation code	44	Modify Function Erase Counter
BOC45 :	Operation code	45	Store/Restore Pseudo Acca.
BOC46 :	Operation code	46	One Word Option
BOC47 :	Operation code	47	Transfer On Index
BOC74 :	Operation code	74	Load Index Register
BOC76 :	Operation code	76	Go To, UP B-box
BOC15 :	Operation code	15	Store Pseudo Accs.
BOC16 :	Operation codes	16,17,20,21	Pseudo Acc. Arithmetic
BOC22 :	Operation code	22	Inverse Divide Pseudo Accs.
BOC23 :	Operation code	23	Load Pseudo Accs.
BOC24A:	Operation codes	24,35	Store Erase/Function Erase
BOC25A:	Operation codes	25,26,27,30, 36,37,40,41	Erase/Function Erase Arithmetic
BOC31A:	Operation codes	31,42	Inverse Divide Erase/Function Erase

BOC32A:	Operation codes	32,33,43	Load Erase/Function Erase
BOC34A:	Operation code	34	Load and Float Function Erase
BOC52 :	Operation codes	52,53,54,55, 57,60,61,62	2-Word Arithmetic
BOC56 :	Operation codes	56,63	2-Word Inverse Divide
BOC64 :	Operation codes	64,65,66	2-Word Load
BOC67 :	Operation code	67	2-Word Load and Float
BOC70 :	Operation code	70	2-Word Load Negative and Float
BOC51 :	Operation code	51	2-Word Store
BOC71 :	Operation code	71	2-Word Boolean
BOC72 :	Operation code	72	Function Erase Boolean
BOC73 :	Operation code	73	Boolean Shift
BOC75 :	Operation code	75	Three Word Command

Class 2 Routines

1. ARITH:

   a. Reset the 160-A indirect bank setting to correspond to the value of the location counter bank setting BANK.

   b. Increment by one the location counter absolute value setting LOCC.

   c. Go to ARITHA.

2. ARITHA:

   a. Transfer control to one of the various command encoder (class 1) routines. This transfer is directed thru a switchboard of 64 locations based on the six bit operation code.

3. ARTSBA:

   a. Increment LOCC by one.

   b. If a bank change occurs go to ARTHSB.

   c. Exit.

4. ARTHSB:

   a. Increment the bank setting BANK by one.

   b. Reset the indirect bank to correspond to the setting in cell BANK.

5. BSUBA:

   a. Store the contents of the Fortran accumulator ACC into one of the pseudo accumulators ACC1, ACC2, ACC3 if the 160-A machine accumulator equals 10, 20 or 30 respectively.

   b. Go to ARITH.

6. BOC35:

   a. Calculate an absolute Erase location given the relative 4 bit location contained in the low order bits of the current command.

   b. Transfer, via a switchboard, to one of four routines depending on one of seven operation codes.

      35: BOC24A

      36: BOC25A

      37: BOC25A

      40: BOC25A

      41: BOC25A

      42: BOC31A

      43: BOC32A

7. BSUBCY:

   a. Preset a return address in an exit switchboard for routine BOC24.

   b. Go to BOC24.

8. BOC24:

   a. Calculate an absolute Function Erase location given the relative 5 bit location contained in the low order bits of the current command.

   b. Transfer, via a switchboard, to either a preset location or one of five locations depending on one of nine operation codes.

      General : Preset

      24: BOC24A

      25: BOC25A

      26: BOC25A

      27: BOC25A

      30: BOC31A

31: BOC31A

32: BOC32A

33: BOC32A

34: BOC34A

9. 52SBA: This routine is used to reference normal 2-word operands.

   a. Set the LOAD - FETCH - STORE option switch (BOPSW).

   b. Bank setting of operand to BWD11.

   c. Absolute location of operand to BWD12.

   d. If the command is not Index Register modified go to (g).

   e. Contents of Index Register to BWD21, BWD22.

   f. Return jump to BINTAD (15-bit add routine).

   g. Go to 52SBB.

10. 52SBB: This routine is used to reference normal 2-word operands.

   a. Set the indirect bank to the setting given in BWD11.

   b. Interrogate the LOAD - FETCH - STORE option switch (BOPSW).

      i. Negative indicates a STORE. The negative number indicates the number of words to be stored. Store indirectly the contents of the Fortran accumulator (ACC) into the operand indicated by the location contained in BWD12.

      ii. Zero indicates a FETCH. Load indirectly from the operand indicated by the location contained in BWD12 into the operand register (OPER).

      iii. Positive indicates a LOAD. Load indirectly from the operand indicated by the location contained in BWD12 into the Fortran accumulator ACC.

   c. Exit.

11. BOC75: This routine is used to reference 3-word operands.

    a. Upper 3 bits of modifier to BWD21.

    b. Lower 12 bits of modifier to BWD22.

    c. If the command is not index register modified go to (g).

    d. Index register to BWD11, BWD12.

    e. Return jump to BINTAD.

    f. BWD11 to BWD21, BWD12 to BWD22.

    g. Set 3-word switch.

    h. Go to ARITH.

12. 75SUB: This routine is used to reference Erasable and Function Erasable operands.

    a. Save option indicator in BOPSW.

    b. If the 3-word switch is set go to (e).

    c. Return jump to BINTAD.

    d. Clear 3-word switch.

    e. Set return address.

    f. Go to 52SBB.

13. ACCOPR:

    a. Store the contents of the Fortran accumulator ACC in the operand register OPER.

14. BOPACC:

    a. Store the contents of the operand register OPER in the Fortran accumulator ACC.

15. BINTAD:

a. Add the two 15-bit quantities A and B and store the results in A, where A is the low order 3 bits of BWD11 plus BWD12, and B is the low order 3 bits of BWD21 plus BWD22.

Class 3 Routines

This class of routines are referenced via a Transfer to Macro (00) command. When one of these routines is referenced, the location counter is set at the current TRM command. In general a list of information will immediately follow the TRM command. The various lists and the meaning of the macros will be discussed. In the following discussions b( ) means bank setting of, and A( ) means absolute location of.

1. IFSNSE:

   a. Information List

      i. b(n1); b(n2); 0,S

      ii. A(n1)

      iii. A(n2)

      S is a selective jump switch setting 1-7; where 1, 2, 4 are single switch settings; 3, 5, 6 are double switch settings and 7 is a triple switch setting.

   b. Execute the intent of the IF Sense Switch Fortran statement where n1 is the transfer location if the desired switch is set and n2 is the transfer location if the desired switch is not set.

IFOV:

   a. Information List

      i. b(n1); b(n2); 0; 0

      ii. A(n1)

      iii. A(n2)

b. Execute the intent of the IF Accumulator Overflow Fortran statement where n1 is the transfer location if the overflow switch is set and n2 is the transfer location if the overflow switch is not set.

3. IFDVCK:

a. Information List

i. b(n1); b(n2); 0; 0

ii. A(n1)

iii. A(n2)

b. Execute the intent of the IF Divide Check Fortran statement where n1 is the transfer location if the divide check switch is set and n2 is the transfer location if the divide check switch is not set.

4. IF:

a. Information List

i. b(n1); b(n2); b(n3); 0

ii. A(n1)

iii. A(n2)

iv. A(n3)

b. Execute the intent of the IF Fortran statement where n1 is the transfer location if the arithmetic statement is negative, n2 for statement zero, n3 for statement positive.

5. CMPUTD:

   a. Information List

      i. M; 0; 0; b(i)

      ii. A(i)

      iii. b(n1)

      iv. A(n1)

      .

      .

      .

      .

      .

      S b(nm)

      S+1 A(nm)

   b. Execute the intent of the Computed Go To Fortran statement. i is the integer control variable. Ni...Nm are the transfer locations.

6. RETURN:

   a. Information List

      None

   b. Return control from the "Call" subroutine to the main program.

7. INCR:

   a. Information List

      i. (M1 - M2)/M3

      ii. M3

      iii. 2, b(Z), 2, b(I)

      iv. A(I)

      v. A(Z)

Where M1, M2, M3 and I refer to the quantities in the DO statement.

DO N I = M1, M2, M3.

Z refers to the address where control is transfered to for continuation of the DO loop sequencing.

b. Execute the intent of the DO statement iteration test. The quantity (M1 - M2)/M3 is incremented by a count of one. If the result of this reduction gives a negative quantity control is transfered to the location Z. If the result is positive control is transfered to the location subsequent to the INCR macro (the statement following statement N referenced in the DO statement.

Floating Point Operations

The section designated 3 WORD PRELIM in the ASSEMBLY LISTING contains the entrance to all floating point operations. It tests for special cases, disassembles the Fortran accumulator (ACC) and operand (OP), puts constants into low core and exits to the proper subroutine.

Procedure: The header S is left out in the following:

FSUB	Entrance to subtract
FADD	Entrance to Add
FMLT	Entrance to multiply
FDIV	Entrance to Divide
ASK	1, 0 or -1 into FLAG for Add, Multiply or Divide. ACC = 0? No, go to TSTOV. Yes, if not multiply go to KAT. For multiply give 0 or overflow as OP is finite or infinite and exit.
KAT	Divide? No, go to MVOP. Yes, if OP $\neq$ 0, exit. Otherwise, go to DVY.
MVOP	Put OP into ACC and exit.
TSTOV	Overflow? Exit.
LOKOP	OP $\neq$ 0? No, go to TSTD. Divide? Yes, go to DVY. Exit, if add.
CLA	0 into ACC and exit.
DVY	Signal divide check.
MOV	Put overflow into ACC and leave.
TSTD	Overflow? No, go to DISAS. Divide, go to CLA. Otherwise go to MVOP.
DISAS	Disassemble ACC and store into EXP, A2, A3, and A4. (the floating accumulator). Disassemble the OP and store into EXPB (EXP + 1) and C1, C2, and C3.
FORN	Determine SIGN according to whether signs agree (+) or disagree (-). Put constants 1000,100,10 and 1 into OP.
JFLX	Entrance for the fixed to floating conversion. Exit to appropriate subroutine (add, multiply, divide).

Floating Point Addition

The floating accumulator (FA) occupies cells A5 to A1, with the most significant part last. Initially the addend D is put in the middle (MA) of FA, i,e., cells A2 to A4 and the augend E into cells (C) C1, C2 and C3. The difference in exponents d = expD -expE = FAC + 3TMOVE is tested for being less than 9 (absolutely) and the greater of the two numbers D and E is put into MA. The other is put into C (or the auxiliary storage B). MA is decimal shifted FAC times and b (or C) is moved TMOVE to the right as it is added into FA.

Procedure:

SADDR	Zero out flas, set up exit frommultiply by 10, determine which number is larger and fix exponent of answer, EXP.
SC	Exponents differ more than 8? No, go to ADJST if differ by exactly 8. Otherwise go to SASSEM.
CHOOS	Set exit from PLCE and go there.
EQX	For equal exponents and A2 = C1, choose OP to be the larger and put 1 into HIGH.

LODE	Signal exit from PLCE to SARITH, i.e., no decimal shift.
ADJST	Determine number of decimal shifts and moves necessary.
PLCE	Put the higher number into MA (HIGH $\neq$ 0 means augend is larger and put addend into B).
SGNF	Determine sign of answer
MLT	Special multiply by 10. Multiplies A1, MA by powers (FAC+1) of 10 and leaves answer there. Used by MULT and DIV routines as well. The arithmetic is first done mod 2000 in each cell of FA and then reduced to mod 1000 for the answer. The algorithm is; 10A1 into A1, 2MA into R (Cells SR1, SR2, SR3), 8MA into MA and R + MA into MA, with the answer in A1, A2, A3, and A4.
REP	Double MA
SUCH	Switch for carries to A1.
X2MLT	Add R to MA and reduce mod 1000. Repeat multiply by 10 if FAC $\neq$ 0.
LEV	Exit from routine.
SARITH	Add-subtract section proper. The smaller number (B or C) is added into cells A2, A3, and A4, or A3, A4, and A5, or A4, A5 according to TMOVE. Go to SUB if signs differ.
ADDER	Beginning of actual add.
SUB	Subtract routine.
INC	A 1 negative? Yes, subtracted wrong way, go to PLCE.
WHAT	Propagate carries (ADDER routine may terminate before reaching leading word).
RON	Find first significant word.
SHOW	If not A1, are there 8 digits in MA? If not, move number to left so that in MLT, the 8 digits are preserved.
SHOW2	Is leading word normalized? Greater than 99? Go to FAC2. Less than 10? Go to FAC1.
FAC2	2 decimal shifts are necessary. Go to MLT.
FAC1	1 decimal shift is necessary. Go to MLT.
DIFER	Moves result left so that there are 8 digits in MA.
SASSEM	Assembly of result. This section is used by MLT, DIV, ADD and SUB. Also fix to float conversion and READ use it. Tests are made for overflow and underflow. The results are left in MA and exit through CNFINI, which ordinarily then moves answer to the ACC, except in fix to float.

Floating Point Division

$10^{9}QU = 10^{a}A/10^{c}C$

Length: 127_8 cells from NFIXEX to NDIVEX (2517-2645 as of 4/4/62)

Description: On entry to floating divide (at ENTRYD), 32+a is in NEXPA; 32+c is in NEXP; A is in NA2, NA3, NA4; and C is in NC1,

NC2, NC3. Upon exit (through SASSEM), 32+a is in NEXP and QU is in NA2, NA3, NA4. The leading word is an intege between 10 and 99.

Procedure:

1) Initialize: I = 1, 32+q = (32 + c) + 32 + (32 + a), j = 2, Q = 0
2) If a C, then q = q + 1j go to 4)
3) Set A = 10 · A
4) If A C, go to 6)
5) If A C, set A = A-C, Q = Q + $10^{(j-1)}$, go to 4)
6) j = j-1
7) If j 0, back to 3)
8) Set j = 3, Q(I) = Q
9) For next time through, set Q = 0, I = I + 1
10) If I 3, go back to 3)
11) Otherwise process is finished. If 2• A C, round by increasing Q(3)
12) Transfer Q to A; exit.

Symbols:

I; Counts words in the quotient
j: Counts decimal digits wihtin a quotient word
Q: Accumulates one word of quotient
A: 3-word divident (or remainder)
C: 3-word quotient
a: Exponent of dividend
c: Exponent of divisor
q: Exponent of quotient

Floating Point Multiplication

$10^{p}PR = 10^{a}A \times 10^{c}C$

Length: 143_8, from MCNG1 to MSHOW (2646-3010 as of 4/4/62)

Description: Entry is made at MPX with 32 + a in EXP, 32 + c in EXPB, A in MA2, MA3, MA4, and C in MC1, MC2, MC3. On exit from the routine (through MSHOW), EXP contains 32 + p and PR is in MA2 through MA7 in BCK format and with 1 MA2 99. SHOW must noralize.

Symbols:

A: Multiplier (3 words in 5 word area) - also hold accumulated product
C: Multiplicand (3 words)
P: Partial product register (4 words)
PR: Product - overlays A
LIER: One word of A used as a multiplier.
a: Exponent of multiplier
c: Exponent of multiplicand
p: Exponent of product (same storage as a)
j: Counter for words of multiplier

Procedure:

1) Exponent work: 32 + p = (32 + c) - 32 + (32 + a), PR(4) = PR(5) = PR(6) = 0
2) A = 10 A (original shift for scaling purposes)
3) j = 3, COUNT = -10 for 10-bit BCK multipliers
4) Go to 10) to perform P = LIER xC where LIER = A(j)
5) PR(j) = 0
6) PR(j+k) = PR(j+k) + P(k) in BCK for k = 0(1)3
7) j = j-1
8) If j 0, go to 4)
9) Otherwise, (A = PR): EXIT to SHOW with address of leading word of product in the accumulator.
10) Set P = 0
11) If bit (11-COUNT) of LIER is "0", go to 13)
12) Otherwise, P(k) - P(k) + C(k) in BCK for k = 1(1)3 and where carries can extend into P(0)
13) COUNT = COUNT +1
14) If COUNT = 0, return to 5)
15) Otherwise P(k) = 2P(k) in BCK for c = 0(1)3
16) Go to 11)

Floating point operations using the 168-2.

1. Format of numbers

a. Floating point

Floating point numbers of the form $\pm 2^{E}.F$ where E is a binary exponent, /E/S 177_8, and F is a normalized 27- bit binary fraction, are represented by Sign bit exp+200_8 (I) Fraction 27 bits

Examples: As 3 160-A words (octal) with the word containing the exponent given first:

1 = 2014, 0---, 0---

2 = 2024, 0---, 0---

3 = 2026, 0---, 0---

-5 = 6035, 0---, 0---

9 = 2044, 4000, 0---

b. Integers

Integers are in the "standard" 22-bit format.

Examples: I = 0000 000I

4000_8 = 000I 0000

—I = 7777 7776

2. Use of routines:

The floating point package is approximately 700_8 in length. It is relocatable within bank 0 using the OSAP loader. The entries for the various function are as follows:

FLOAT	Ist address	
FIX	"	+I
MULTIPLY	"	+2
DIVIDE	"	+3
SUBTRACT	"	$+10_8$
ADD	"	$+13_8$

All operations return to symbolic location ARITH.

Overflow sets symbolic location OVFLW non-zero.

Dividing by 0 sets symbolic location DIVCHK negative.

All operations require five transient low core locations, plus a three word Operand register, also in low core.

With the exception of the conversion routines FIX and FLDAT, the operations also require a three word accumulator in low core,

The last word of the operand register is symbolic location OP, the last word of the accumulator symbolic location ACC.

Numbers are stored "backwards" in the operand register and accumulator. v.e. a floating point I in the accumulator looks like

ACC - 2	0----
ACC - 3	0----
ACC	2014

While the integer - I in the operand register looks like

OP -2	xxx
OP -I	7776
OP	7777

Under these assumptions the operations give the following results:

FLOAT : Replace the integer in the operand. register by its floating point equivalent

FIX : replace the floating point number in the operand register by its integer equiv.

MULTIPLY : replace the contents of the accumulator by the product of the contents of the accumulator and the operand register.

DIVIDE : Replace the contents of the accumulator by the quotient of the contents of the accumulator by the contents of the operand register (ACC/OP)

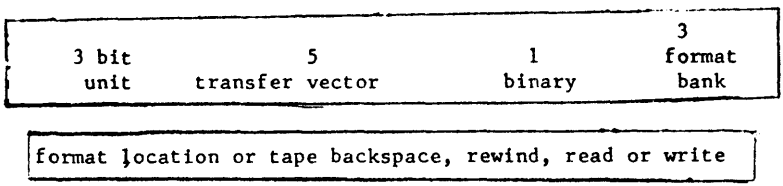
SUBTRACT : Replace the contents of the accumulator by the contents of the accumulator minus the contents of the operand register. (ACC-OP)

ADD : Replace the contents of the accumulator by the sum of the contents of the accumulator and the contents of the operand register.

Overflow occurs when the result of an operation has an exponent $\geq 200_8$.

DOCUMENTATION OF I/O IN 160-A FORTRAN INTERPRETER

All words in capital letters refer to actual symbols used in coding. Thus ACC refers to the first cell of the FORTRAN accumulator, which for short will be called the accumulator. OP is the first cell of the operand. The machine accumulator will be referred to as A (the A-register). The I/O part of the interpreter is divided into several sections. The fixed interpreter contains the Magnetic Tape and I/O List Control. Format Control together with WOUT (the output conversion) occur as a separate package and are present only when a format statement appears in the source code. Another subroutine called WIN (input conversion) is also called whenever a format statement appears. Although it appears as a library I/O subroutine it is really one of the several interpreter modules. All input-output data is transmitted through a 171 word buffer in the fixed interpreter (cells 200 to 371). Whenever a record needs to be transmitted a return jump is done to the necessary library I/O subroutine through the transfer vector switchboard in cells 100 to 177. Only WIN and Flextype need to have fixed transfer vector numbers, (44 and 7 relative to 100) the other routines can be arbitrarily ordered. They may be in bank 0 or bank 1. There are three entrances to the I/O from the macro switchboard; IOI, which is for input, IOO, for output, and IOT, end of I/O. The next two succeeding words of the object code after the transfer macro have bits set which give the format location, if any, the library I/O transfer vector number, the tape unit number, if any, and the type of magnetic tape operation if binary. The form is as follows:



The tape unit numbers are from 0 to 7 to correspond with the actual numbers 1 to 8. The I/O list consists of a series of load instructions (for output command numbers are 32 & 64) store instructions (for input command numbers are 24 & 51) interlarded with various other commands for dolooping and upping B-boxes. Every command is interrupted and the unpaired load (or store) commands are interpreted as data transmission commands.

Low Cores Flags Used in Format

Symbol		
;EOF	≠ 0 or 7777	either first entry to format or re-entry after end of format
;HFLAG	≠ 0	in the midst of Hollerith characters in format string
;BRECT		counter for number of physical records in a logical record
;BNK ∉ ZLOCC		current format string location
;BUF		buffer counter (from 200 to 371)
;WAY	- or +	input or output
;TIP		contains BCD characters, E,F,I,0 or A according to field specs
;WID		field width
;PEC		decimal specification in E or F
;REP1	-	number of repeats in a single field specification
;IF	0 or +	first entry to format or the relative location in format string
;FORMF	-1	that format has been processed to a data conversion specificati
;PARCT	0 or +	upper for every left parens and downed for right. 0 is end of format.
;QUIK	2525 or 5252	current character in upper part of word or lower part of word
;LOCLP		relative position in format string of character following last unquantified left parenthesis
;BNK & FLOCC		location of start of format string
;CHAR		current character in format
;HCNT		hollerith counter
;OFFLG		END OF FILE FLAG

Relative Flags in Format Control

Symbol	Set to	Signals
SDATRY	-1	data transmissive instruction has been encountered
SPECT	-1	a field delimitor (i.e. /, or)) has not been encounte:
SLLP		relative position of last quantified left parenthesis
SREPAR		number of repeats in a parens repeat
SNUM		storage for integers in string, e.g., width or decimal specifications

I/O List Control Initialization

IOI — Entry for input, put -1 into SWAP and go to STWAY

IOO — Entry for output, put 1 into SWAY

STSWAY — Decode object code and put switchboard location of I/O routine into TROUT determine whether binary and set SBINSW accordingly, set FBNK and FLOCC

SPLUG — Short circuit switchboard to cause interrupt on every command. Zero out flags in format, initialize buffer counter, set SEOF $\neq$ 0 to flag first entry to I/O. Set FLOCC. If binary called for, go to SBINAR. If input, read a record. Otherwise exit to ARITH.

SBINAR — Signal to I/O routine binary operation by negative tape number in UNIT. Set SFORMF negative and zero out SEOF. If not data transmission, go to SMVTAP. If read, go to SBREAD, otherwise write called for. Zero out first cell of buffer and set SBRECT. Exit to ARITH.

SBREAD — Enter 370 into SBUF to cause a record to be read on first store command and exit to ARITH.

Connection With I/O Library Subroutines

STRANS — Enter by JPR. Negative number in A is stored in SLENG (parameter to tape routine) to backspace n records and go to STLNG. Go to SBUFF on positive entry.

Magnetic Tape Control (Con't)

SBERR — Stop on error return from tape subroutine (only parity error encountered, i.e. - or + in A). If read, go to SBRED. Zero out first word of buffer to signal not final record of logical record and up SBRECT.

SBWRT — Write binary. Transfer accumulator to buffer and go to TSTLDD.

STWY — Branch to read (SBREAD) or write (SBWRT).

SBRED — Store first word of record into counter.

SBREED — Transfer three words from buffer to accumulator and exit.

SBINTR — Terminating I/O entry. If read, read to last record of logical record and exit. Otherwise go to SBWRIT.

SBWRIT — If write end of file, exit. Put in record count and write record. May be empty except for record count.

SBTX — Transmit record. Nonzero return indicates error. Exit to SCOMEX to restore switchboard and return to ARITH. Rewind, backspace and write end of file.

SMVTAP — If A $\neq$ 0 go to RWDEOF. Otherwise backspace one physical record. Read the record in binary to find the mode. If parity error return, go to BCDBK. Halt on other error (i.e., nonzero). Examine bits 4 to 10 in first word of buffer to find number of records in the logical record and backspace accordingly.

SBCDBK — Examine first word of buffer to see whether bits 0 to 3 are set to 1. If so, go to error halt, since this is an illegal BCD code. Otherwise flip UNIT positive and backspace one record.

RWDEOF — Rewind or write end of file according to whether the read or write library subroutine is called for.

Format Control

SFORMT — If SFORMT or SREP1 = -1, go to TGOSEN. If SEOF $\neq$ 0, go to SETFOR. Otherwise go to SELBNK.

TGOSEN — If SDATRY = -1, go to WIN or WOUT. Otherwise reset SFORMF and go to TSTLDD.

SAFT — Entry from WIN. If SREP1 set, go to TSTLDD. Otherwise go to SELBNK.

SBUFF — Set up length of record. If blank record, put two blanks at start of buffer. 0 length means rewind in case of read tape. Reset SBUF to 200, start of buffer.

SETSMP — Jump to I/O in bank 0 or bank L according to whether vector is even or odd.

SWITCHBOARD INTERRUPT

SWINT — Enter here for every command in I/O list. If op code is 75, exit to SWTF+1. If a store command (24 or 51) go to STINT. Ohterwiee go to STLDW.

STINT — Zero out SLDSWT (cancels any call for a load command). If output go to SWTBR. Otherwise test and clear store disable STSWT (= -1 diables store, otherwise not set). If SEOF not set go to SDATC. If SIF = 0 go to format processing first. Otherwise read record before executing store.

SDATC — Flag call for data (a real store or load encountered) and go to SCOFOR.

STLDW — Clear store disable and test load switch. If not set go to TSTLDD.

SLOAD — A load instruction encountered. If SEOF set, go to SGOFOR. Otherwise clear load switch and flag data call by going to SDATC.

TSTLDD — Test for load instruction (32 or 64). If yes, put -1 into SLDSWT and STSWT.

SWTBR — Exit on either SFORMF or SREP1 set.

SCOFOR — Branch to binary or format control.

SLDEXT — Return to macro switchboard.

IOT — Terminating instruction in the I/O list. Branch to binary or reset instruction inmacro switchboard. If input, exit. If output, send out record if not empty.

Magnetic Tape Control

SBIN — If buffer full (or set on entry read I/O) transmit record, unless on read binary, the last record has been read, then do not read any more records.

Format Control (Con't)

SHCON — Hollerith is packed in format string. A check is made of the field width so as not to exceed buffer. Error halt occurs if too large.

SENF — End of format indicated. Reset SPARCT and SIF. Put 7777 into SFORMF and SEOF and transmit record if output. Go to TSTLDD.

SFEXT — Check to see if this was first entry to format processing. If so and output is called for, exit to STLDW+1 rather than TSTLDD; if input, go to SFORMT. Otherwise go to TSTLDD.

SETFOR — Reset ZBNK and ZLOCC unless SIF = 0.

SFORA — If SHFLAG ≠ 0, go to SHCON (Hollerith).

SDELIM — The format characters are tested in the following order: blank,), (, /, digit, X, H, and if none of these, the character is stored in STIP. Blanks are ignored. In the following if a delimitor, i.e.,) /, is encountered test SPECT and go to SENSP first before performing required operation. If right parenthesis, up SREPAR. If +, go to SDONPR. Otherwise set SIF to SLLP and go to SETFOR.

SPROC — Put -1 into SFORMF and go to SFEXT.

SDONPR — Reduce SPARCT. If 0, go to SENF.

SFETUP — Fetch character from format string.

SELBNK — Set indirect bank to ZBNK to fetch. Restore to 0.

SFETF — Put upper or lower character into SCHAR according to SQUIK.

TSLFT — If "(", up SPARCT. If SNUM ≠ 0, go to SREGP. Otherwise set up infinite repeat by putting SIF into SLOCLP. EXIT*.

SLAS — If slash, transmit record.

SENSP — End of field indicated. If STIP has H or X in it, go to SDELIM. If E or F, put SNUM into SDEC, else into SWID and go to SPROC.

SFLSPC — Put -1 into SPECT. If character is a digit go to SCINT. If X or H flag, go to SDELIM. Otherwise put -SNUM into SREP1 and 0 into SNUM. EXIT.

---

* EXIT means return to SFETUP, the normal return after processing a character.

Format Control (Con't)

SREGP	Put - SNUM into SREPAR, SIF into SLLP, 0 into SNUM and EXIT.
TSX	For X, add SNUM to SBUF for output, and for input, put blanks into buffer.
SCINT	Put 10*SNUM + SCHAR into SNUM. EXIT.
SHCON1	Put -SNUM into SHCNT and SHFLAG and 0 into SNUM. EXIT.

Integer Routines in the Fixed Interpreter

Add

VISUB	Entry for Subtraction. Flip operand to complement.
VADD	Entry for addition. Partial results are accumulated in the accumulator and the necessary adjustments for differing signs, carries, etc., are made later by testing the appropriate bits. If signs differ go to VSUBT.
VADDRT	Like signs. Exit if no carry from lower part. Otherwise correct lower word and carry to upper and exit.
VSUBT	Signs differ. Combine lower parts of operand and accumulator and exit. Otherwise test for negative zero. If not, go to VNEGRE for negative result or VNZHI + 1 for positive result.

Multiply

VXT	Exit. Check for minus zero or plus zero and exit.
VINTML	Entry for multiply. Put sign of answer in VISIGN. Make operand and accumulator positive.
VFINMR	Determine multiplier, i. e., the number whose upper word is zero. Not both of the numbers can be more than 11 bits since then overflow would result. The larger number will be called the multiplican It goes into cells VM1 and VM2.
VXTQ	Flag overflow.
VLDC	If accumulator is multiplicand, put it into VM2.
VZERO	Zero answer.
VSAVR	Operand in multiplicand.
VSMD	Store into multiplier VMLTR and zero out accumulator.

Integer Routines in the Fixed Interpreter (Con't)

Multiply

VSTAR — MLTR is shifted right once each entry until 0, whereupon exit.

VFIN — After each shift test the lower bit. If set, add in multiplicand to accumulator.

VDOM — Double the multiplicand and return to VSTAR unless overflow, then go to VXTQ.

Divide

VINTDV — Entry for divide. Set up exit in VDIVEX so that switchboard can be used to go to the add routine. Put sign of answer in SDIGN. There is a 4-word temporary accumulator VJ1, VJ2, VJ3 and VJ4, the last 2 cells of which contain the positive accumulator. Check for zero operand, set divide check if so and exit.

VTSNEG — Set operand negative (for future subtracts) and zero out accumulator and quotient XSACCJ (2 cells), and set VQUO, the indirect addresser for the quotient.

VENDIV — Exit from routine. The lower part of the remainder is saved. Fix sign of answer and exit.

SHIFTY — 4 word left shift of dividend.

VCOMP — Compare new dividend and the divisor if smaller, go to VMOVQ. If upper parts are equal, go to VEQHI.

VDIFER — Otherwise do a subtraction and enter a 1 bit into the quotient.

VMOVQ — Right shift quotient bit VQUOB and if not zero, go to SHIFTY.

VNXBIT — Up VQUO and go to exit if end.

VFIRS — Set up VQUOB and go to VSTOB.

VDQHI — Compare lower parts of dividend and divisor and go to VMOVQ if dividend is smaller. Otherwise go to VDIFER.

Flags Used in WOUT (Output Conversion)

Low Core

WSIGN	Sign of Result.
BCNT	End of field indicator.
TSAV	Temporary for indirect addressing in integer conversion.
CANSWT	0 for E, -1 for F.
SGNEXP	Signed exponent.
WEXP	Absolute value of exponent.
KEEP1(2&3)	Fractional part of floating point number.
SDECLC	Absolute location of decimal point in buffer (for a field).
WLET	Start of location of significant digits to be stored in buffer.
NUM	Indirect addressing for KEEP.
KTDATA	Indirect addressing for accumulator.
FUNCD	Same as STIP, the BCD code for the type of conversion.

Relative

SDECTP	Temporary storage for SDEC (specification may be changed in case of F conversion).

Output Conversion, Last Half of FORMAT Package

WOUT	Branch to A conversion or INITO (initialization).
ACON	As many characters (beginning with the upper 6 bits of ACC) are outputed as called for in SWID, but garbage after first six.
*	See POW32 for 168-2 version.
FINISH	If positive answer, got to TOOUT2. Otherwise search through buffer to first nonblank character (not beyond end of field) and insert minus sign before this character.
TOOUT2	Set SBUF to next field. If answer is blank, replace last blank by zero. Exit to SAFT.
FETCH	Set CANSWT. Split up floating point number into WSIGN, WEXP, KEEP1, KEEP2, and KEEP3. Branch to ECON for CANSWT = 0.
*	See FETCH below.
(FCON)	F conversion. Set up SDECLC and WLET. Check for special case, SDEC = 0. Put in 0 in buffer.

Output Conversion, Last Half of FORMAT Package (Con't)

SNOZ	If room for digits left, go to CHKFLD.
CONZER	Put in decimal point and zeros after it in place of blanks. Go to FINISH.
INITO	Initialize SDECTP, NUM and KTDATA. Blank out portion of buffer for the field. Check for too large a field width and go to FTSMLL if not enough room. Otherwise branch to proper conversion (ICON, OCON, or FETCH).
CHKFLD	Check WLET against start of field. If not enough width, try E conversion. If still not enough room by reducing decimal specification, go to FTSMLL.
FTSMLL	Field is too small (or too large) for the number. Put out X's and exit.
FLDOK	Check WLET against SDECLC and up if necessary.
*	See FLDOK below.
NUMCON	Converts fractional part of floating point number to digits and stores them in the buffer. Exit to CONZER.
ECON	E conversion. Put the exponent field in the buffer. Set BCNT to location of "E" in buffer. Go to SETDEC.
ICON	I conversion. Put out integers right justified, truncating on right if too long for the field width.
TNOW	Set up entry to table and make accumulator positive.
TSTAR	Convert integers, double precision subtract used in fixed interpreter.
TPUTIN	Store digit in buffer, use blanks instead of insignificant zeros.
TUPSAV	If end of field, go to FINISH.
TSING	Single precision.
TLATE	Table of powers of 10.
OCON	O conversion. Right justify field and drop insignificant zeros. Truncate on right. Exit.

---

* Refer to revelant sections in the 168-2 version. See below.

Additions for 168-2 Version

Changes in WOUT were made for the E and F type conversion, so as to use the 168-2. Also entry from WIN (input conversion) was provided for. Except where noted below, the coding is the same as for WOUT in the regular system (as referenced by *).

POW32	Table of powers of 10 with exponents 32, 16, 8, 4, 2 and 1 in that order.
QO	Approximating $\log_{10}2$.
POWQ	The binary exponent of the number to be outputed.
*	Continue with FINISH above.
FETCH	The binary exponent is reduced by one and multiplied by log 2. The greatest integer in the answer is stored in SGNEXP as a trial power of 10. Branch to MOVTO for output.
QCONIN	Entry from WIN. Test exponent. If smaller than -30, flag UNSWT and up SGNEXP by 10. Otherwise an inadvertent underflow might occur.
MOVTO	Skip to QITZ if SGNEXP = 0. Put absolute value of SGNEXP in QEX. Initialize PL to power of 10 and set QFIRS.
QLD	Algorithm for computing QEXth power of 10. Exit to QIT when finished, i.e., QEX reduced to zero. Multiply in higher powers of 10 first.
QFUL	Put power of 10 in operand.
QIT	Move accumulator to operand.
QITZ	Move original number to accumulator and for output, divide by created power of 10 for positive exponent, multiply for negative exponent. Do the reverse in case of input.
QNORM	Go to FINORM for output. Otherwise test UNSWT and if necessary reduce result by 10**8 before exiting.
FINORM	Reduce accumulator further by dividing by ten if necessary so that the binary exponent is less than 4.
QNORD	If binary exponent not yet 0, up SGNEXP. Put absolute value of SGNEXP into EXP and branch to (FCON) or ECON.
QDIVT	Multiply accumulator by 10 to obtain next digit in answer.
QDIV10	Enter by JPR for the above multiply.
*	Continue with (FCON) above.
FLDOK	Left shift accumulator N times where N is in XXNX in the leading word ACC Since this is end around, pick off 1, 2 or 3 bits as necessary for the first digit of the answer.

Additions for 168-2 Version (Con't)

NUMCON	Jump to QDIV10 for multiply by 10. Continue with NUMCON above.

Low Core Flags in WIN

WSIGN	Sign of the answer
(W)EXPF	Exponent flag. -1 for positive exponent, +1 for negative exponent and 0 if no exponent flag encountered.
(W)DECCT	The relative location of the decimal point in the field.. A negative number means that a period has been encountered in the input data.
PLACCT	The number of digits after the decimal point.
(W)DIGCT	Significant digits counter. = 0 means no significant digits as yet and that any + or - sign belongs to the fractional part, not exponent.
WID	The field counter, which is initially set to -SWID.
(W)STORD	Indirect addressing cell for placing digits in the accumulator or exponent.
WIDF	Equivalent to SWID
DECF	" " SDEC
FUNCD	" " STIP
WHI	" " ACC
WLO	" " ACC + 1
SAC	Indirect addressing cell for A type.
CONSWT	Switch for I, E and F. Set 0 for I and $\neq$ 0 for E and F.

(W) means the W appears as a header sometimes.

WIN (Input Conversion)

WIN	Entry to input conversion. Branch to NONNUM for 0 and A type conversion. Otherwise go to DECODE.
ENDFL	If F or E type, go to DOFLT. Otherwise determine sign of integer result and exit.

DOFLT	Add bias (40) to signed exponent. Adjust exponent for decimal point and significant digits. If overflow, put 3740 in ACC. Combine fraction and exponent together, affix sign and exit to SAFT in format control.
*	See DOFLT below for 168-2 version.
WEXT	Exit to format control through transfer vector.
INUP	Up SBUF and WID and fetch next character from buffer.
WINIT	Character is tested thus: / - _ + E and the digits 0 to 9. A slash terminates the field. If no exponent flag had been set, adjust DIGCT accordingly and exit.
WMIN	Test for -. If DIGCT = 0, set WSIGN. Otherwise put 1 into WEXPF.
WTPLUS.	Test for +. Ignore if DIGCT = 0. Otherwise go to WESIG.
WESIG	Flag positive exponent and change CONSWT to integer conversion. Set PLACCT and initialize STORD.
WTSPER	Test for period. Set WDECCT.
DECODE	Set CONSWT, branch to FCON for E and F type.
INIT	Initialize WID, STORD, set PLACCT = 1, and zero out the accumulator, DIGCT, EXPF, WEXP and WSIGN. Go to WINIT.
FCON	E or F type conversion called for. Initialize DECCT to SDEC. Set up switches in the floating conversion (FLTCON). Go to INIT.
DIGCON	Digit conversion. Change BCD zero to 0. Go to TSCON for integer conversion. Check for insignificant zeros otherwise. Up DIGCT for significant digits and ignore digits after the 8th.
TSCONS	If process on fraction, go to FLTCON. Otherwise branch to WTRUE for I type or put 10*WEXP + VALUE into wexp and go to INUP.
WTRUE	Go to WM10LT for double precision integer multiply by 10. Otherwise check ACC + 1 for number less than 200 and use machine MUT instruction.
WCOMM	Add in VALUE to accumulator and go to INUP.
*	See FLTCON below for 168-2 version.
FLTCON	The current digit is multiplied by 1, 10, or 100 according to the switch BRANX and added into the proper cell of the accumulator. Go to INUP.

WM10LT Double precision decimal shift of the accumulator for integer conversion. Go to WCOMM.

OCTCON Octal conversion. Accumulator is shifted left and the current character is added in until the end of the field. BCD characters greater than 7 are ignored.

ACONV Pack characters from buffer into the accumulator until the end of the field (or until end of the accumulator) and fill the rest of the accumulator with blanks if necessary.

Modifications for 168-2 Version

The only changes made to the above are as follows:

DOFLT Adjust the exponent as in DOFLT above but without the bias and put it into SGNEXP. Normalize and float the fractional part which is in the accumulator. Exit to SAFT + 4 in format control to compute the necessary power of 10 and combining together of the exponent and fraction part.

FLTCON Multiplies the accumulator by 10, using the 168-2. Correct indirect ban bank is selected from SBNK, since the execution loader will up SBNK for bank 1 because SBNK occurs just before a relocatable constant. Return to INUP.

Error Stops in I/O

Magnetic tape operations

SMVTAP - 1 A parity error has occured on trying to backspace a binary record. Rerun from stop with 0 in A.

SBERR Either an end of file has been encountered or there is a parity error on the tape (+ or -). Can't continue.

Format control

SHUPBF + 4 Field width too large in Hollerith.

SLDUP + 3 A zero has been encountered in the format string, which is illegal. Probably unmatched parenthesis so that end of format has been overshot.

There is no error stop in case the field width is too great for E, F or I on output but only X's are outputed. No check is made on input (other than H) but garbage will be picked up for conversion after the end of the buffer.

FLOATING POINT PACKAGE

The floating point arithmetic in 160 A FORTRAN is the same as that in 160 FORTRAN except that all reference to SNOPSW has been dropped and the arithmetic is unrounded.

FIX TO FLOATING CONVERSION

METHOD: To convert an integer N to its equivalent floating point form a b c d e f y * 10 **M perform the following arithmetic operations from left to right.

$$(N - a*10^6) /10^3 = bcd + efg / 10^3$$

(The lower part of the remainder after an integer division is kept in cell OP + 3) The parts a, b c d and e f g are than put together with the correct exponent by jumping to a special entry SJFLX in the floating point package.

FLCONV — Preserve exit thru macro switchboard. Save the accumulator and set up flags and switches to the floating point package. Determine sign of answer and set the operand positive. Put -1,000,000 in the operand. Go to SECPAR if subtraction is not necessary.

SGOSUB — Keep subtracting 1,000,000 until accumulator is negative. Then add 1,000,000 back in. Answer in A2.

SECPAR — Divide reduced number by 1,000. Put quotient in A3 and remainder in A4. Set up return to SBACK and jump to SJFLX for assembly of parts.

SBACK — Transfer lower part of result to operand, restore accumulator and exit.

FLOATING TO FIX CONVERSION

METHOD: To convert a floating point number N= a b c d e f g *10**M, add to it .64* 10**8. Then a is in A2. bcd in A3 and efg in A4. Perform the following from left to right.

(a * 1000 + bcd) * 1000 + efg

XCONV Preserve exit and save the accumulator. Determine sign of result and set the operand positive. If number too big, i.e. greater than .9*10**7, go to XBIG. If less than one, answer is Zero.

XAD16 Add in . 64*10**8.

XM1000 Multiply A2 by 1000. Add in A3 and again multiply by 1000. Add in A4 and sign and go to XSTOP1.

XBIG Set overflow switch.

XSTOP1 Restore accumulator and exit.

ANALYSIS OF CONVERSION OF FLOATING POINT NUMBER ON OUTPUT

Let the number

$N = 2^{p}f = 10^{q}g$, where $.5 \leq f \leq 1$ $.1 \leq g < 1$.

$2^{p}f$ being the internal representation. To obtain a first estimate to q, consider the following, where log is to the base 10.

$\log 2^{p}f = \log 10^{q}g$

$pc + \log f = q + \log g$, where $c = \log 2$

$q = pc + \log f/g$

Now $1/2 < f/g < 10$ and hence

$-c < \log f/g < 1$

Therefore

$$c\,(p-1) < q < pc + 1$$

Let q' = entier $\lceil c(p-1) \rceil$ i.e. greatest integer function

Now for $p \geq 1$

$$10^{\log 2^p} > 10^{\log 2^{p-1}} \geq 10^{q'}$$

and hence

$$(1) \qquad f \leq \frac{2^p f}{10^{q'}} = 2^r h$$

where h is the normal fraction in the internal representation and $0 \leq r < 4$. (If $10^{q'}$ does not reduce r below 4 another division by 10 is done.)

If $r \neq 0$, the fraction h in the accumulator is left shifted r times to obtain the first digit of g. The succeeding digits are obtained by multiplying h by 10.

In the case $P \leq 0$, the above inequalities are reversed and $10^{q'} > 10^{q}$. Since then the number N is multiplied by $10^{q'}$ and the relation (1) still holds.

Powers of 10 are computed from a table containing 10^{32}, 10^{16}, 10^{8}, 10^{4}, 10^{2} and 10, from which any power S, $|S| \leq 63$, can be obtained from at most one multiplication of each factor, e.g.

$$10^{32} = 10^{16} * 10^{4} * 10^{2} * 10$$

Specifications for Format Decoding in the Interpreter

All format cracking will be done in the interpreter, the compiler will merely take all characters between the enclosing parenthesis (including these also) and pack them in data storage. Thus it will be possible at execution time to read in a format statement to be used during the run. There will be no error stops executing the format statements but illegal characters will be ignored as far as possible, though they may occassionally mislead the routine. A format statement will be processed until data execution is called for (i.e., an I,O,A,E or F code) whereupon control is returned to the list until an unpaired Load (Store) is encountered, which causes return to the format routine, switches to Out (in) conversion routine, back to format to see whether a simple repeat is being performed, whence to list control, otherwise resume format processing. End of format is signalled by encountering the left parenthesis that matches the beginning parenthesis. A flag is set, the format counter is set so that re-entry to the routine resumes processing at the field following the rightmost left parenthesis not preceded by a numeral and a record is sent if Out conversion is called for. Upon encountering the second TR IO if an out conversion has been called for and the buffer is not empty a record is sent. If there is a further call for data transmission (by an unmatched Load-Store pair) and an In conversion is flagged than another record will be read before format processing is resumed.

Note that when a format statement is used in conjunction with a PRINT statement then the first field of every format group producing a line of print should be either 1HO for double spacing, 1H1 for page eject or 1H for single spacing. If no control character is specified and the first character of the record is not a 0 or a 1 then single spacing will be performed and the first character will be lost.

Specifications for Integer Add and Multiply

These routines may be entered either by a JFI 1 to VADD and VINTML respectively (with also an entry to subtract via VISUB) or by a JPR to VADXT and VMLTIN with a positive number in the accumulator for add and negative for subtract in the former entrance. For an entry via JFI the integers should be in ACC, ACC 1, OP and OP 1 with the result in ACC and ACC 1. With a JPR entry the numbers should be in ACCJ, ACCJ 1, OP and OP 1 with result in ACCJ and ACCJ 1. With a JFI entry the exit is to ARITH. The number should have a 22 bit magnitude, using the left 11 bit magnitude, using the left 11 bits of two words. A negative number is indicated by complementing its magnitude (both upper and lower parts). Thus the octal number 26047 would have the form 000 000 000 101 010 000 100 111 in binary or in octal digits 0005 2047. -26047 would then be 7772 5730. No overflow or fault indicator is set but garbage will result if the numbers are out of range.

. Specifications for the Object Code for the I/O in the Interpreter

There will be two transfer to I/O macros: TR IOI (INPUT) and TR 100 (OUTPUT), TR IO stands for either in the following. The locations of the I/O subroutines in the transfer vector will be grouped together and there will be less than 32 of them, so bits are needed for the tape unit (or which of the four types in the flex-type routine) and 15 bits for the location of the format statement (which will be somewhere in data storage). One bit must be set to signal binary. If binary is flagged the second word carrying the location of format is not needed and instead the codes 0 for Read, 1 for Write, 2 for backspace, 3 for rewind and 4_1 for write end of file will be used. Thus the I/O coding will look like this:

TR IO

11-9	8-4	3	2-0
Unit	VecLoc	Binary flag = 1	Format Bank

Location of format or
Code for tape movement

List

TR IO

The second TR IO can be of either type and need not agree with the first. Thus the I/O control will operate independently of what equipment is used and will only know what type of equipment is being used when binary is called for, since in that case only a tape routine is possible. Hence I/O executions calling for backspace, rewind or write end of file should be signalled by flagging binary.

SECTION 3.5

Specifications for I/O Transmission

The I/O Transmission routines are absolutely relocatable and are available in OSAS-A binary format on the library tape. The function of these routines is to read or write one physical record, check for errors, and indicate special results. All bank settings, except relative, must be zero upon entry to the routines. The routines are entered by a SRJ to the first location of the routines with Location 77 containing the address of the parameter list (in bank zero). Location 75 contains the return address. The parameter list is explained for each individual routine, but is always three contiguous words. If a normal record was transmitted, the return should be with A = 0. The routines for I/O transmission are as follows:

Name	Input/Output	Equipment
RWTF	I/O	161 Typewriter-paper tape punch and reader
LPRINT	0	1612 printer
LP166	0	166 printer
CDPNCH	0	523 card punch
RDC088	I	088 card reader
RDC167	I	167 card reader
RD1634	I	163, 164, or 606 magnetic tape units
WR1634	0	163, 164, or 606 magnetic tape units
RD1607	I	1607 magnetic tape l units
WR1607	0	1607 magnetic tape units
RDC405	I	405 card reader

Parameter list

1. location of buffer
2. punch flex = 1

   type out = 2

   read flex = 3

   type in = 4
3. length of record (necessary for output only)

The return is always with A = 0. If an operation is requested when the unit is not ready, or if reading is requested when no information is available, a delay will be executed which will not allow any further action to be taken. Recovery is not possible except by satisfying the condition which caused the delay.

There are no page control functions available. All lines for output are single spaced. Blank lines are described for each function. All illegal codes are converted to blanks with no error indications made.

Punch flex -

Each record is preceded by a lower case code. Additional case codes are supplied as necessary to provide the correct characters. These case codes have no effect on the character count to determine the record length. The record length may be 1z x 120 decimal characters, but should be governed by the length of the line of the felxowriter being used for listing. Each record is terminated with a carriage return. A blank record may be created by supplying one or more blank codes as a complete record. All letters are punched upper case and numbers all lower case as a result of the BCD to Flex conversion.

Type Out-

Each record is preceded by a carriage return with case codes supplied as needed. All letters are typed upper case. A record may be 1z x z120 decimal characters, but should be governed by the practical length of the line provided on the typewriter. No additional blanks are supplied, and the carriage return is not provided at the end of the record. Since only legal BCD characters may be typed, this eliminates the use of the tab for output. Blank lines may be typed by providing a record containing one or more blank codes.

Read flex -

Each record is assumed to begin in lower case. Characters are converted until a carriage return is read or until 120 decimal BCD characters are stored. If 120 characters are stored before a carriage return has been read, reading continues until a carriage return is found and any intervening information is skipped. If a carriage return is found before 120 characters have been stored, the record is terminated and remains in this shortened form. Blanks are supplied after the carriage return has been read. The special flex characters are as follows:

Apostrophe is converted to asterisk

Colon is converted to dollar sign.

Blank records may be created by successive carriage returns, with each carriage return terminating one record.

Type in-

When a typewriter input is requested by the program, a carriage return and a question mark, will be typed, and lower case code selected. The computer is ready to accept input at this time. Characters will be accepted, converted, and stored until a carriage return is received, or until 120 decimal characters have been stored. Blank records may consist of a carriage return as the only character of the record. There are several special codes for typewriter input:

Tab - this will delete the record which has just been typed and reposition the typewriter as for a new record. This allows immediate error correction but must be used before the carriage return has been typed.

Apostrophe or Quotation Marks - this is a psuedo change of case code. The use of either of the case shift keys on the typewriter does not affect the record in any way. This change of case code must be used in order to make all legal codes available for input. Since this code does not affect the character count, there is no limit to the number of times it may be used in each record.

Procedure: RWTF

1. Enter by SRJ with TBLADR containing the location of the parameter list and RETURN containing return address.
2. Is type or Flex input called for? Yes, clear buffer area.
3. Typewriter operation called? Yes, go to 5.
4. Set to do flex code conversions. Parameter call for punching? Yes, go to 7.
5. Go to 12.
6. Set to do typewriter code conversions. Parameter call for output? Yes, go to 20.
7. Go to 31.
8. Preset addresses; select punch; punch lower case code.
9. Pick up character, go to 37.
10. Is A negative? Yes, punch case code. Restore converted code.
11. Punch converted code. Was this the last character? Yes, punch carriage return code. Go to 30 (A = zero).
12. Go to 8.
13. Preset addresses and flags. Select paper tape reader.
14. One character to A (omit leader).
15. Is character a carriage return? Yes, go to 30.
16. Is character a delete code? Yes, go to 13.
17. Restore character. Go to 38.
18. Is A negative? Yes, go to 13.

19. Store code. Have 120 characters been stored? Yes, advance input record to the next carriage return. Go to 30 (A = zero).

20. Go to 13.

21. Preset addresses and flags. Wait for typewriter ready. Select typewriter output. Type carriage return and a lower case shift.

22. Pick up one character.

23. Is character a left paren, dollar sign, asterisk, or right paren? Yes, go to 28.

24. Restore original code. Go to 37.

25. Is A negative? Yes, type case shift code and pick up converted code.

26. Type converted code.

27. Has last character been typed? Yes, go to 30 (A = zero).

28. Go to 21.

29. Appropriate code to A. Save code. Is a case shift required? Yes, type upper case shift.

30. Go to 25 (code in A).

31. Return.

32. Preset addresses and flags. Wait for typewriter ready. Select typewriter output. Type carriage return, upper case code, question mark, lower case code. Select typewriter input.

33. One character to A.

34. Is character a carriage return? Yes, go to 30.

35. Is character a Tab? Yes, go to 31.

36. Is character an apostrophe? Yes, change case flag setting. Go to 32.

37. Restore code. Go to 38.

38. Preset to convert BCD codes to Flex or Typewriter codes. Go to 39.

39. Preset the convert Flex or Typewriter codes to BCD codes.

40. Convert code. Return to 9, 17, 24, or 40.

41. Store converted code. Is this the 120th character? Yes, go to 30 (A = zero).

42. Go to 32.

RD1634 or RD1607

Parameter list:

1. location of buffer
2. logical tape number - 1

   positive - BCD mode (even parity)

   complemented (negative) - binary mode (odd parity)
3. zero = rewind tape

   positive non zero - read one record

   negative - backspace n records. N is the complement of the number of records to be backspaced.

The normal return will be with A = 0. An end of tape error response will cause the tape to be rewound and the routine will halt with A (non zero). If restarted, a normal exit will be made. If A is cleared the present record will be reread, assuming a new tape. If a parity error is sensed the tape will be backspaced and will be reread. If errors persist after trying three times the parity selection is checked. If BCD mode was selected a stop is made. Restarting will cause an exit to be made. If binary mode was selected, a return is made with a negative number in A. If EOF is encountered, the buffer will be cleared, a flag set for EOF function, and a register exit made.

Procedure: RD1634

1. Enter by SRJ with PARAM containing the location of the parameter list and RETURN containing return address.
2. Initialize counters, create EXF codes, wait for ready on the requested unit, select parity, pick up third parameter. Was backspacing requested? Yes, go to 12.
3. Was reading requested? Yes, go to 6.
4. Rewind tape. Set A = zero.
5. Return.
6. Read one record. Wait ready. Was this record an end of file record? Yes, set EOF flag. Clear buffer to blanks and go to 5.
7. Was there a parity error? Yes, go to 10.
8. Was the end of tape indicator set? Yes, rewind tape, and stop. If restarted, go to 9. If restarted with A cleared, go to 6.
9. Set A = 0 and go to 5.
10. Is this the third try? Yes, go to 14
11. Backspace; wait ready, go to 6.
12. Backspace; wait ready. Is this the n^{th} backspace? Yes, go to 5 with A = zero.
13. Go to 12.
14. Is this a binary operation? Yes, set A negative; go to 5.
15. Is this third time? Yes, halt. If restarted, go to 5. No, backspace three records, reposition tape. Go to 6.

Procedure: RD1607

1. Enter by SRJ with PARAM containing the location of the parameter list and EXITAD containing the return address. Save this location, and initialize counters. Create FWA and LWA + 1 and pick up tape number. Is binary operation operation requested? Yes, go to 11.
2. Initialize for BCD operation.
3. Create the select code, select the tape, and wait ready. Is rewinding requested? Yes, go to 12.
4. Is backspacing requested? Yes, go to 13.
5. Read one record, save last address, and wait for tape ready. Is the end of tape indicator set? Yes, go to 14.
6. Was an end of file record read? Yes, go to 15 (A non zero).
7. Was there a parity error? No, go to 17 (A = zero).
8. Was this the third error? No, go to 16.
9. Is this a binary operation? Yes, go to 19 (A is negative).
10. Stop. If restarted, set A = 0 and return.
11. Initialize for binary operation. Go to 3.
12. Rewind the selected tape. Go to 19 (A = zero).
13. Backspace n records. Go to 19 (A = zero).
14. Rewind the selected tape and stop. If restarted, return (A is set to zero). If restarted with A cleared, go to 5.
15. Set EOF flag and return. (A = zero). Clear buffer to blanks.

16. Backspace tape and wait ready. Go to 5.

17. Was this binary read operation? Yes, go to 19 (A = 0).

18. Unpack the record and store in buffer area. Go to 19 (A = 0).

19. Return.

WR1634 or WR1607

Parameter list

1. location of buffer
2. logical tape number

   positive = BCD mode (even parity)

   complemented (negative) = binary mode (odd parity)
3. length of record (number of characters)

   zero requests an end of file record.

The return will always be with A = 0. If the end of tape indicator is sensed, a stop will be made with A containing the rewind tape code. If restarted, the record will be written again, assuming a new tape. If parity errors persist after three efforts to rewrite the record, some blank tape will be skipped and another effort made to write the record. If this fails after three times a stop is made with A = 0. Restarting will cause an exit to be made.

Procedure: WR163

1. Enter by SRJ with PARAM - location of the parameter list and RETURN containing return address.
2. Initialize to create EXF codes for the proper tape unit, and the required parity. Wait for ready response from requested unit. Select parity.
3. Is the record length = zero? Yes, go to 11.
4. Write record. Request status. Response indicate parity error? Yes, go to 8.
6. Response indicate end of tape? Yes, backspace, write 2 EOF's. If restarted, go to 4.
7. Return with A = zero.
8. Is this the third try? Yes, backspace write EOF. If this third EOF? Yes, go to 10.
9. Backspace, wait ready, go to 4.
10. Stop. If restarted, go to 7.
11. Write end of file record. Go to 7.

Procedure: WR1607

1. Enter by SRJ with PARAM containing the location of the parameter list and EXITAD containing return address. Save this location. Initialize. Is binary operation requested? Yes, go to 8.
2. Set for BCD mode. Is an end of file record requested? Yes, go to 4.
3. Pack the characters two per word and adjust counters.
4. Select the requested tape and wait ready. Is an end of file record requested? Yes, go to 9.
5. Write one record and wait ready. Was there a parity error? Yes, go to 10.
6. Was the end of tape indicator set? Yes, go to 13.
7. Return. (A = 0).
8. Set for binary mode. Go to 4.
9. Write and end of file record and wait ready. Go to 7.
10. Backspace tape and wait ready. Is this the third error? No, go to 5.
11. Write end of file record and wait ready. Reset counters. Is this the third error sequence? No, go to 10.
12. Stop. If restarted, go to 7.
13. Backspace, write 2 EOF's, rewind selected tape and stop. If restarted go to 5.

RDC088

Parameter list

1. location of buffer
2. ignored
3. ignored

Eighty (80) characters are converted to BCD codes to be stored in the specified buffer area. There is no error checking done. Blank cards may be read without special indications made. The only return is with A = 0. This routine uses only the primary feed of the 088. If the card reader does not become ready when control has been transferred to this routine, a delay will be executed which will not allow any further action to be taken. There is no recovery from this condition except a ready signal from the reader.

Procedure: RDC088

1. Enter by SRJ with PARAM containing the address of the parameter list and BUFCHK containing return address.
2. Wait for card reader ready. Initiate buffer action to read one card. Initialize counter.
3. Wait for row to be read.
4. Convert 36 columns. Is this the last row? Yes, go to 6.
5. Set new column value and adjust counters. Go to 3.
6. Is this the last group of columns? Yes, go to 8.
7. Preset to convert columns 37-80. Go to 3.
8. Return.

CDPNCH

Parameter list

1. location of buffer
2. ignored
3. length of record

The return from this routine is always with A = 0. The length of record may be from 1 to 80 characters. A blank card may be punched by supplying a one word record consisting of a BCD blank code. There is no error checking done.

Procedure: CDPNCH

1. Enter by SRJ with PARAM containing the address of the parameter list and CDP98 containing return address.
2. Pack BCD characters. Zero punch image. Initialize.
3. Convert BCD codes to punch image (1 to 80 codes)
4. Wait for card punch ready.
5. Initiate buffer action.
6. Return.

Parameter list

1. location of buffer
2. ignored
3. length of record

The first character of the record is interrogated as a page control character and is not printed.

BCD 1 = page eject

BCD 0 = double space

BCD blank = single space

There is no error checking done for illegal codes. The return is always with A = 0. The record length may be from 2 to 121 decimal characters. This length must include the page control character which is never printed.

Procedure: LPRINT (1612)

1. Enter by SRJ with PARAM containing location of parameter list, and RETURN containing RETURN address.
2. Save parameters.
3. Select printer (without interrupt), wait ready.
4. BEGIN = location of second character in buffer.
5. PRINT1 = last word address + 1 of buffer.
6. First character of record = BCD1? No, go to 10.
7. Eject page.
8. Print line and advance paper.
9. Return with A = zero.
10. First character of record = BCD zero? No, go to 8.
11. Space paper on line. Go to 8.

Procedure: LPR166

1. Initialize to print, set exit address, and wait for printer ready. Does the first character request a page eject? Yes, go to 4.

2. Does the first character request a double space? Yes, go to 5.

3. Pack characters two per word and adjust the record length. Wait for printer ready. Print and move paper. Set A = 0 and return.

4. Eject the page. Go to 3.

5. Move paper one space. Go to 3.

RDC167 or RDC405

Parameter list

1. locating buffer
2. ignored
3. ignored

Eighty (80) characters are converted to BCD code to be stored in the specified buffer area. Error checking is done in the 167 routine for feed and amplifier failure. Recovery from the 167 routine consists of running from next location after halt with last card replaced in hopper for the case of amplifier failure. In the case of reader failure in the 405 routine, the card is sent to the secondary hopper. Recovery consists of replacing the card in hopper, and running from next location after halt. Blank cards may be read without special indications made. The only return is with A = 0. If the card reader does not become ready after one card read cycle, a halt occurs with 3535 in A-register. Run from next location wait ready again.

Procedure: RDC167 or RDC405

1. Save parameter list location. Wait one card read cycle for card reader ready, then halt with 3535 in A-register. Run to continue waiting ready.
2. Initialize to convert 80 column card image.
3. Convert to BCD codes and store in buffer.'
4. Return.

The Natural Logarithmic Function

Length: 151_8 locations

Method: The fractional approximation was used as follows: choose n and m such that $X = 10^n 2^m$

Where: .48 y 1 and 0 m 3

Then: log X = n log 10 + m log 2 + log y

Procedure:

1) Is X 0? If no error return to main program.
2) Store exponent of 10 in N. Let denote the fractional part.
3) Is .48? If so M = 0 and go to 8.
4) Is .24? If so M = 1 and go to 7.
5) Is .12? If so M = -2 and go to 7.
6) Is .12? If so M = -3.
7) Compute Y = (2^{-m}).
8) T = (Y-1)/(Y+1)
9) log y = $T(A + T^2 (B + C/(D + T^2)$

   where

   A = 2.

   B = .10907889

   C = .777314

   D = -1.3940651

10) log X = N log 10 + M log 2 + log y
11) Return to main program.

Accuracy:

8 digits for f .6 or f .48

7 digits for .48 X .6

Timing:

190

The Exponential Function

Length: 211_8 location

Method: The fractional approximation was used on e^r as follows; choose integers n and m such that $e^x = 10^n 2^m e^r$ where m 3 and R $\frac{\log 2}{2}$.

Procedure:

1) Set sign X positive and denote it by Y.
2) Fix Y/log 10 and store in N
3) Is N = 0? If so go to 6.
4) Float N
5) N. log 10
6) Z = Y - N. log 10
7) Find integer part of Z/log2 + .5 and store in M.
8) Pick up 2^M (=2, 4 or 8) and store in EX2.
9) R = (Z/log2 - M) log 2.
10) $-S(R) = 600/(60+R^2)-12.$
11) -S(R)+R = 0? If so $e^R = 4$ jump to 15.
12) $e^R = (-S(R)-R)/(-S(R) + R)$
13) $e^y = e^R \cdot EX2 \cdot 10^n$
14) Check over/under flow. If o/u set overflow switch, and give overflow no as answer.
15) Is X 0. If not go to 17.
16) $e^x = 1/e^y$.
17) Return to main program.

Accuracy:

7 digits or better.

Timing:

220

AR TAN FUNCTION (NORMAL)

ATANF(X)

1. If X is positive, set FLAG to 0. If negative, set FLAG to negative and X to -X.

2. If X 0.4, set T = X and Y = 0.

3. If 0.4 _ X 2.4, set T = (X-1) and Y = PI/4.

4. If X _ 2.4, set T = -1./X and Y = PI/2,

5. Then ATANF(X) =

$$Y + T * DO + T*T* D1 + \cfrac{E1}{T*T + D2 + \cfrac{E2}{T*T + D3}}$$

where: DO = 1.0
D1 = -0.015585371
D2 = 2.1005541
D3 = 1.6210238
E1 = -0.58531514
E2 = -0.419003

6. If FLAG is negative set ATANF(X) = -ATANF(X)

7. Exit

SQUARE ROOT FUNCTION (NORMAL - SQRTF(X)

1. If X = 0 go to 8.

2. Set X = ABSF(X).

3. Divide exponent (biased 32_{10}) by 2 discarding any remainder. Store in ROOTN. Note that bias is 16_{10}.

4. If exponent is odd, set X = mantissa * 10.
   If exponent is even, set X = mantissa.

5. First approximation: $R = \frac{X + P}{S}$. (See below for values of P and S).

6. $$ROOT = \frac{X}{\frac{X}{R} + R} + \frac{\frac{X}{R} + R}{4}$$

7. Add exponent ROOTN to ROOT adjusting for additional bias of 16_{10}.

8. Exit.

For mantissa in range .1 to .32, P = .18666408, S = .88191319
.32 to 1. , P = .58926406, S = 1.5656854
.1 to 3.2, P = 1.8666408, S = 2.7888544
3.2 to 10., P = 5.8926406, S = 4.951132

COSINE FUNCTION (NORMAL) COSF(X)

SILE FUNCTION (NORMAL - SINF(X) = COSF (X - PI/2)

1. X = ABSF(X).

2. N = Integer part of X/(PI/2). This is the quadrant number -1.

3. R = PI/2 * N - PI/2 * X. This will be nagative.

4.

For N =	FLAG =	RTEST =	SIGN =
0 (mod 4)	0	.96	0
1	1	.64	1
2	0	.96	1
3	1	.64	0

5. If R _ RTEST, reverse flag. Set R = R + PI/2.

6. Flag = 1 use following SILE approximation, then go to 8.

$$COSF(X) = -R * S1 + \frac{S2}{R*R + S3 + \frac{S4}{R*R + S5}}$$

This will be positive.

7. Flag = 0 use following cosile approximation:

$$COSF(X) = 1 + R * R * C2 + \frac{C3}{R*R + C4 + \frac{C5}{R*R + C6}}$$

8. If SIGN = 1, set sign of result to minus.

9. Exit.

COS - SIN, Continued.

S1 = 7.2308469
S2 = -814.80759
S3 = 55.409623
S4 = 1262.6242
S5 = 16.754492

C2 = -1.6771458
C3 = 271.20668
C4 = 80.855187
C5 = 2442.5426
C6 = 16.333897

EXITF Function

This function is used to transfer control from the execution of one program back to an initialization routine within the system for compilation of the next program. The form of the statement is

X = EXITF (N)

X is any floating point variable and N specifies the magnetic tape unit as follows:

N	Magnetic tape unit
0	163 or 162
1	1607

The EXITF function statement should replace all STOP statements in a program.

PDUMPF, MDUMPF, CDUMPF Functions

These functions are used to obtain reloadable core dumps. PDUMPF outputs to paper tape, MDUMPF outputs to 163 or 162 magnetic tape (unit 2) and CDUMPF outputs to cards, using the 523 card punch. These statements have the form

X = CDUMPF (N)

X = PDUMPF (N)

X = MDUMPF (N)

X is any floating point variable and N is the number of the last bank to be dumped. N can be an integer variable or constant while X is a dummy variable whose value will be changed during the execution of the function. A halt occurs after the dump is completed. Running from this halt continues execution at the statement following the dump function.

Separate loaders are available for each of the dumps. Operating instructions for these loaders are contained in the 160-A FORTRAN OPERATIONS MANUAL.

When the dump is reloaded, execution begins at the statement following the dump function.

Procedure: X 163F and X 1607F

1. Initialization
   a. Set Indirect and Direct Bank settings to zero.
   b. Store Relative bank setting in direct cell 1.
   c. Set up counter containign the compliment of the number of banks to be dumped plus one.
2. Rewind to loadpoint tape 2/
3. Store 101 as last word address, and 2 as first word address to be output.
4. Write tape 2.
5. Output 0 - first word of record 1 contains

   0
6. Output record via normal output.
7. Check for parity error - if zero go to 8 - backspace 1 record, write end of file, backspace 1 record, go to step 4.
8. Determine location of last record written.
   a. Subtract 101 from last word address, if zero go to 9.
   b. Subtract 117 from last word address, if zero go to 10.
   c. Subtract 400 from first word address, if zero go to 11.
   d. Subtract 1 from last word address, if zero go to 12.
   e. Increase last word address and first word address by 1000, go to 6
9. Store 7777 in first word address and 220 in last word address. Set indirect bank setting to 9, go to 6.
10. Store 400 in first word address, 1000 in last word address, go to 6.
11. Store 1000 in first word address, replace add 1000 to last word address; go to 6.
12. Add one to number of bank counter, if zero go to 14.

13. Reinitialize for banks 1-7.

    a. Store 7777 in first word address

    b. Store 1000 in last word address

    c. Increase by 1 Indirect bank setting

    d. Go to 6

14. Write end of file mark.

15. Halt.

16. Normal sequence of return.

Procedure: XCDMPF

Assume knowledge conversion of card column to card row output for 523 punch.

1. Initialize card image by zeros in card buffer.
2. Set Indirect and Direct bank setting to zero.
3. Store Relative Bank setting.
4. Set up Exit address.
5. Initialize first word address and Indirect bank setting.
6. Initialize number of locations to be stored on card in bits 11 - 6 of col 1.
7. Store the number of bank from which card is dumped in bits 1, 3, 4 of column 1.
8. Binary code placed in bits 0, 2, of column 1.
9. Initialize counter for X words to be stored. This is normally 105.
10. Store first word address in column 3 of card.
11. Load output area, store in columns 4-72 of card image.

    Add 1 to First word address.

    Add 1 to storage address.

    Add 1 to word counter if zero, go to 20, if non-zero, go to 11.
12. Sequence cards in columns 73-79.
13. Punch card.
14. Test last word address of card.

    a. Is it zero, no go to 14b.

       Store 7777 in first word address

       Output card containing only specific cell, go to 6.

    b. Is this the card following specific cell, yes to to 15.

    c. Is this the card prior to location 220 in bank 0, yes go to 16.

d. Is this location 220 in bank 0, yes go to 17.

e. Is this the last card in bank 0, yes go to 18.

f. Is this the last card in bank 1-7, yes go to 19.

15. Add 1 to bank counter, if zero go to 21.

Add 1 to Indirect bank setting, go to 6.

16. Store 6 in number of words on card, go to 6.

17. Store 400 in first word address, go to 6.

18. Store 54 in number of words on card, go to 6.

19. Store 30 in number of words on card, go to 6.

20. Store checksum in column 2 of card image, go to 12.

21. Halt.

22. Return to normal sequence.

Procedure: XPDMPF

1. Initialization
    a. Set Direct Controls to zero.
    b. Store Relative bank setting in direct cell 1.
    c. Set up counter containing the number of banks to be dumped plus one.
2. Punch out leader containing 300 frames.
3. Set up frame counter containing complement of 100.
4. Punch out first frame containing number of last bank, add the contents of this frame to checksum.
5. Set Indirect bank setting to zero.
6. Load output cell.
    a. Store in temporary cell.
    b. Replace add to checksum.
    c. Output left half of word.
    d. Increase frame count by 1.
    e. Output right half of word.
7. Test location of output word:
    a. Is it loc 0, yes go to 7d.
    b. Is it loc 7777, yes go to 9.
8. Add one to output address
    a. If zero, go to 12.
    b. Is it loc 220, yes go to 10.
    c. Increase frame count by one, not zero go to 6, if zero go to 11.
9. Add one to number of banks counter, if zero go to 13.
    a. Reinitialize output address.
    b. Add one to Indirect bank setting.
    c. Add one to number of banks dumped.
    d. Go to 5.

10. Check for bank 0, not zero go to 8c; load 400 store in output address.
11. Load checksum.

    a. Output left half of word + 100.

    b. Output right half of word + 100.

    c. Reinitialize frame counter with complement of 100.

    d. Reinitialize checksum by storing 0; go to 6.

12. Load 7777 in output address go to 6.

13. Halt.

14. Return to normal sequence.

160-A FORTRAN PLOTF FUNCTION for 165 PLOTTER

The PLOTF function in its current form drives the plotter (165) at plotter speed on the normal channel. The PLOTF function is referenced in a manner similar to the other library functions with the exception that the arithmetic output of the function is meaningless. The plot function has three arguments and is called as follows:

```
C = PLOTF (X,Y,J)
```

The interpretation of X and Y is determined by the value of J.

J = 1 X and Y are interpreted as the scale factors.

J = 2 X and Y are interpreted as the origin or current pen location

J=3 or 4 This will result in pen motion

If J is 3 the pen will be raised, moved to X,Y and lowered.

If J is 4 or greater a line will be from the current pen location to X,Y. For J equal to or greater than 4 the plotter function will assume that the pen is down, this assumption results in a time savings of 70 milliseconds per pen excursion

The pen motion in the X or Y direction resulting from one reference of the PLOTF function may not exceed 20.

160A FORTRAN END OF FILE FUNCTION, EOF

The EOF function provides a means of testing for the reading of an end of file during the magnetic tape reads at execution time. The form of the statement used to reference this function is as follows:

IF (EOF(X)) N_1,N_2,N_1

where X is a dummy variable. Control will transfer to N_1 if an end of file mark was read or N_2 if one was not read. This test must be made before the next magnetic tape READ statement is executed.

A**I for 168-2 Unit

Length: 66_8 locations.

Method: A is multiplied to itself I times if I is positive. 1/A is multiplied -I times if I is negative.

Procedure:

1) Is A = 0? If so go to 10.
2) Is I = 0? If not go to 4.
3) A**I = 1 and go to 10.
4) Is I negative? If not go to 6.
5) (ACC) = 1/A
6) Store (ACC) in ACC2 and ACC3.
7) I = I-1.
8) Is it zero? If so go to 10.
9) (ACC3) = (ACC3) * (ACC2) and go to 7.
10) Return to main program.

Accuracy: 8 digits.

Timing: 5 (I) + 1 ms

A**B for 168-2 Unit

Length: 40_8 locations.

Method: A**B = EXPF (B*LOGF(A))

Procedure:

1) Is A = 0? If so go to 8.
2) Is B = 0? If not go to 4.
3) A**B = 1, and go to 8.
4) Store B in location B.
5) Transfer to LOGF to obtain log (A).
6) Multiply by B.
7) Transfer to EXPF to obtain Exp (B x log (A))
8) Return to main program.

Accuracy: 7 digits or better

Timing: 65 ms

I**J for 168-2 Unit

Length: 51_8 locations.

Method: I is multiplied to itself J times to obtain I**J.

Procedure:

1) Is I = 0? If so go to 10.
2) Is J = 0? If not go to 4.
3) I**J = 1 and go to 10.
4) Is J 0? If not go to 6.
5) I**J = 0 and go to 10.
6) Store I in ACC2 and ACC3.
7) (ACC) = I*(ACC2).
8) (ACC1) = J-1.
9) Is (ACC1) = 0? If not go to 7.
10) Return to main program.

Accuracy: 8 digits

Timing: 5 (J) + 1 ms

EXPF for 168-2 Unit

Length: 123_8 locations.

Method: The fractional approximation was used for e^r after portioning $e^x = 2^m \cdot e^r$ where $r \quad \frac{\log 2}{2}$.

Procedure:

1) Store given value in X.
2) Take absolute value of X.
3) Z = X/log 2.
4) Z + .5.
5) Take integer part (let N).
6) Float N.
7) Subtract Z.
8) Multiply by log 2 to obtain -
9) -S( ) = 600/(60 + r^2) - 12.
10) e^r = (-S(r) - r)/(-S(r)+r).
11) Add N to the exponent of e^r to obtain e^x.
12) Check 1st word of e^x.
13) If it is positive go to 15.
14) Set overflow flag.
15) Is X 0? If so go to 17.
16) $e^x = 1/e^x$.
17) Return to main program

Accuracy: 7 digits or better.

Timing: 31 ms.

LOGF for 168-2 Unit

Length: 132_8 locations.

Method: $X = 2^m f$ where .5 f 1

Then log (X) = m log (2)+log(f).

Procedure:

1) Is X = 0? If so error return to main program.
2) Take absolute value of X.
3) Store exponent of 2 in m.
4) t = (f+1)/(f-1)
5) log f = $t(a+t^2(b+c/d+t^2)))$

   a = 2.

   b = .10907889

   c = .777314

   d = 1.3940651

6) log(X) = m log (2) + log (f)
7) Return to main program.

Accuracy: 8 digits

Timing: 34 ms.

SINF for 168-2 Unit

Length: 146_8 locations.

Method: For 0 t $\frac{II}{2}$

$$\sin t = t\left(1+\cfrac{t^2}{a_1+t^2-\cfrac{}{a_2+t^2-\cfrac{}{a_3+t^2-\cfrac{}{a_4}}}}\right)$$

Where: A1 = -6.0000029
A2 = -3.3334504
A3 = 11.4363332
A4 = 8.9696229

Procedure:

1) F = X/(/2).
2) Fix F, ie I = F.
3) If In even SIGN = 0, if odd SIGN 0.
4) If I/2 is even T = 1., if odd T= -1.
5) If X 0 change sign of T.
6) Calculate t = /2 (F-I)
7) If SIGN = 0 then go to 9.
8) t = /2-t.
9) Compute SINF(t) by above formula.
10) Return with SINF(X) = T.SINF(t)

Accuracy: 7 digits or better

Timing: 40 ms

COSF for 168-2 Unit

Length: 6_8 locations

Method: Cos (X) = Sin(X - /2)

Accuracy: 7 digits or better

Timing: 41 ms

ATANF for 168-2 Unit

Length: 155_8 locations.

Method:

$$ATANF(t) = t\left(d_1 + \cfrac{e_1}{t^2 + d_2 + \cfrac{e_2}{t^2 + d_3 + \cfrac{e_3}{t^2 + d_4}}}\right)$$

for t 2 - 1

Procedure:

1) Store sign of X in SIGN.
2) Take absolute value of X.
3) Is X 2-1)? If so t = X, go to 6.
4) Is X 2+1)? If so t = $\frac{1+X}{1+X}$, SI = /4, go to 6.
5) t = -1/X, SI = /2.
6) Arct (t) = $t\left(d_1 + \cfrac{e_1}{t^2 + d_2 + \cfrac{e_2}{t^2 + d_3 + \cfrac{e_3}{t^2 + d_4}}}\right)$

   Where $d_1 = .20131206$
   $e_1 = 3.1138501$
   $d_2 = 5.4062285$
   $e_2 = -3.9283157$
   $d_3 = 2.718\ 2904$
   $e_3 = -.15058394$
   $d_4 = 1.3387596$

7) Arct (x) = SI + Arct (t)
8) If SIGN is negative Arct (X) = -Arct (x).
9) Return to main program.

Accuracy: 8 digits

Timing: 36 ms

SQRTF for 168-2 Unit

Length: 153_8 locations.

Method: Two Newton's iterations are used after a linear approximation.

Procedure:

1) Take absolute value of X.
2) Store (exponent)/2 in (EXP).
3) Is exponent of X even? If so go to 6.
4) (EXP) = 2(EXP)
5) Fractional part F = F/2 and go to 7.
6) A = .875 and B = .27863 and go to 8.
7) A = .578125 and B = .421875
8) $X_1 = AF + B.$
9) $X_2 = .5(X_1+F/X_1)$
10) $X_3 = .5(X_2+F/X_2)$
11) SQRTF(X) = (EXP)X_3
12) Return to main program

Accuracy: 8 digits

Timing: 23 ms

SECTION 4.1

THE OSAP-AF ASSEMBLY SYSTEM

OSAP-AF is a version of OSAP-A which will correctly assemble programs containing both OSAP and 160-A FORTRAN intermediate language mnemonic operation codes.

The FORTRAN intermediate language and the 160-A assembly system are described elsewhere. The mnemonics assigned to the FORTRAN intermediate language instruction and their proper use for assembly purposes are described below:

MNEMONIC OPERATION CODES FOR THE 160-A FORTRAN INTERMEDIATE LANGUAGE.

The operations fall into the following categories determined by the use the assembl program makes of the other information on the line with the op-code.

NO ADDRESS REQUIRED.

For these op-codes the assembly program ignores the remaining information on the line.

The following FORTRAN op-codes are of the no address required type:

NAME	OSAP-AF MNEMONIC	OCTAL
Drop Out	DRO	0100
Return Transfer	RTR	0200
Convert Accumulator (FIX)	IAC	4601
Convert Accumulator (FLOAT)	FAC	4600
Set sign positive (integer)	IPS	4641
Set sign positive (floating)	FPS	4640
Set sign negative (integer)	INS	4645
Set sign negative (floating)	FNS	4644
Change sign (integer)	ICS	4643

NAME	OSAP-AF MNEMONIC	OCTAL
Change sign (floating)	FCS	4642
Addition skip	ASK	4602
Store ACC in ACCn	STO a	15x a
Add ACC_a to ACC, Store in ACCn	ADn a	16n a
Subtract ACC from ACC, Store in ACCn	SBn a	17n a
Multiply ACC by ACC_a, Store in ACCn	MPn a	20n a
Divide ACC by ACC_a, Store in ACCn	DVn a	21n a
Inverse Divide ACC_a by ACC, Store in ACCn	DIn a	22n a
Load ACC_a, Store in ACCn	LDn a	23n a

It should be noted that in ACCn commands no checking for illegal characters is done in the n portion of the command while a is only checked to be 0 a 9. It should also be noted that ACCn commands are written as four letter op-codes.

NO ADDRESS MODE

For no address mode instructions the assembly program computes the sum of the address and additive fields, and, if the result is less than 100_8, adds the sum to the octal instruction.

The following instructions are of the no address mode type.

NAME	OSAP-AF MNEMONIC	OCTAL
Halt and Proceed	HPR	0000 + (AA)
Transfer	TRA	0200 + (AA)
Transfer to Macro	TRM	0100 + (AA)
Store function erasable, floating	FST	2400 + (AA)
Add function erasable, integer	IAD	2540 + (AA)
Add function erasable, floating	FAD	2500 + (AA)
Subtract function erasable, integer	ISB	2640 + (AA)

NAME	OSAP-AF MNEMONIC	OCTAL
Subtract function erasable, floating	FSB	2600 + (AA)
Multiply function erasable, integer	IMP	2740 + (AA)
Multiply function erasable, floating	FMP	2700 + (AA)
Divide function erasable, integer	IDV	3040 + (AA)
Divide function erasable, floating	FDV	3000 + (AA)
Inverse Divide function erasable, integer	IID	3140 + (AA)
Inverse Divide function erasable, floating	FID	3100 + (AA)
Load function erasable	FLD	3200 + (AA)
Load negative function erasable, integer	ILN	3340 + (AA)
Load negative function erasable, floating	FLN	3300 + (AA)
Load negative and floating convert function erasable	ILC	3400 + (AA)
Store erasable, integer	IES	3540 + (AA)
Store erasable, floating	FES	3500 + (AA)
Add erasable, integer	IAE	3640 + (AA)
Add erasable, floating	FAE	3600 + (AA)
Subtract erasable, integer	ISE	3740 + (AA)
Subtract erasable, floating	FSE	3700 + (AA)
Multiply erasable, integer	IME	4040 + (AA)
Multiply erasable, floating	FME	4000 + (AA)
Divide erasable, integer	IDE	4140 + (AA)
Divide erasable, floating	FDE	4100 + (AA)
Inverse divide erasable, integer	IIE	4240 + (AA)
Inverse divide erasable, floating	FIE	4200 + (AA)

NAME	OSAP-AF MNEMONIC	OCTAL
Load erasable, integer	ILE	4340 + (AA)
Load erasable, floating	FLE	4300 + (AA)
Increase function erasable counter	IEC	4400 + (AA)
Decrease function erasable counter	SEC	4440 + (AA)
Store accumulators	SAC	4500 + (AA)
Restore accumulators	LAC	4540 + (AA)
Transfer to Power	TRP	5000 + (AA)

(AA) = Address + Additive

Since the assembler only checks (AA)$\leq 100_8$ while for these instructions it is necessary to have (AA)$\leq 40_8$ ($\leq 14_8$ for erasable) it is advisable to use caution when (AA) is defined symbolically.

FORWARD AND BACKWARD RELATIVE INSTRUCTIONS

On these instructions the assembler computes the sum of address and additive fields and, if neither is symbolic, adds the result to the octal instruction. If either the address or additive field is symbolic, and the location field does not begin with a minus sign, then the current location counter is subtracted from the result, and this number (or its negative, depending on whether the mode is forward or backward), is added to the octal instruction. If either address or additive is symbolic, and the location field starts with a minus sign, the sum of address and additive is computed and the result, or its negative, added to the octal instruction.

The following instructions are of relative type:

NAME	OSAP-AF MNEMONIC	OCTAL
Transfer on index backward	TIX	4700 + (AA)
Relative transfer forward	TRF	0300 + (AA)
Relative transfer backward	TRB	0400 + (AA)

(AA) = Address + Additive

NAME	OSAP-AF MNEMONIC	OCTAL
Transfer on positive, forward	TPF	0500 + (AA)
Transfer on positive, backward	TPB	0600 + (AA)
Transfer on negative, forward	TNF	0700 + (AA)
Transfer on negative, backward	TNB	1000 + (AA)
Transfer on zero, forward	TZF	1100 + (AA)
Transfer on zero, backward	TZB	1200 + (AA)
Transfer on non-zero, forward	ZNF	1300 + (AA)
Transfer on non-zero, backward	ZNB	1400 + (AA)

(AA) = Address + Additive

TWO WORD OF CODES

These op codes cause two lines of coding to be generated, the second line being the sum of address and additive.

The following are the two word instructions.

NAME	OSAP-AF MNEMONIC	OCTAL
Store, two-word	TSnb	51nb
Floating add	FAnb	52nb
Floating subtract	FSnb	53nb
Floating multiply	FMnb	54nb
Floating divide	FDnb	55nb
Floating inverse divide	FVnb	56nb
Integer add	IAnb	57nb
Integer subtract	ISnb	60nb
Integer multiply	IMnb	61nb
Integer divide	IDnb	62nb
Integer inverse divide	IVnb	63nb

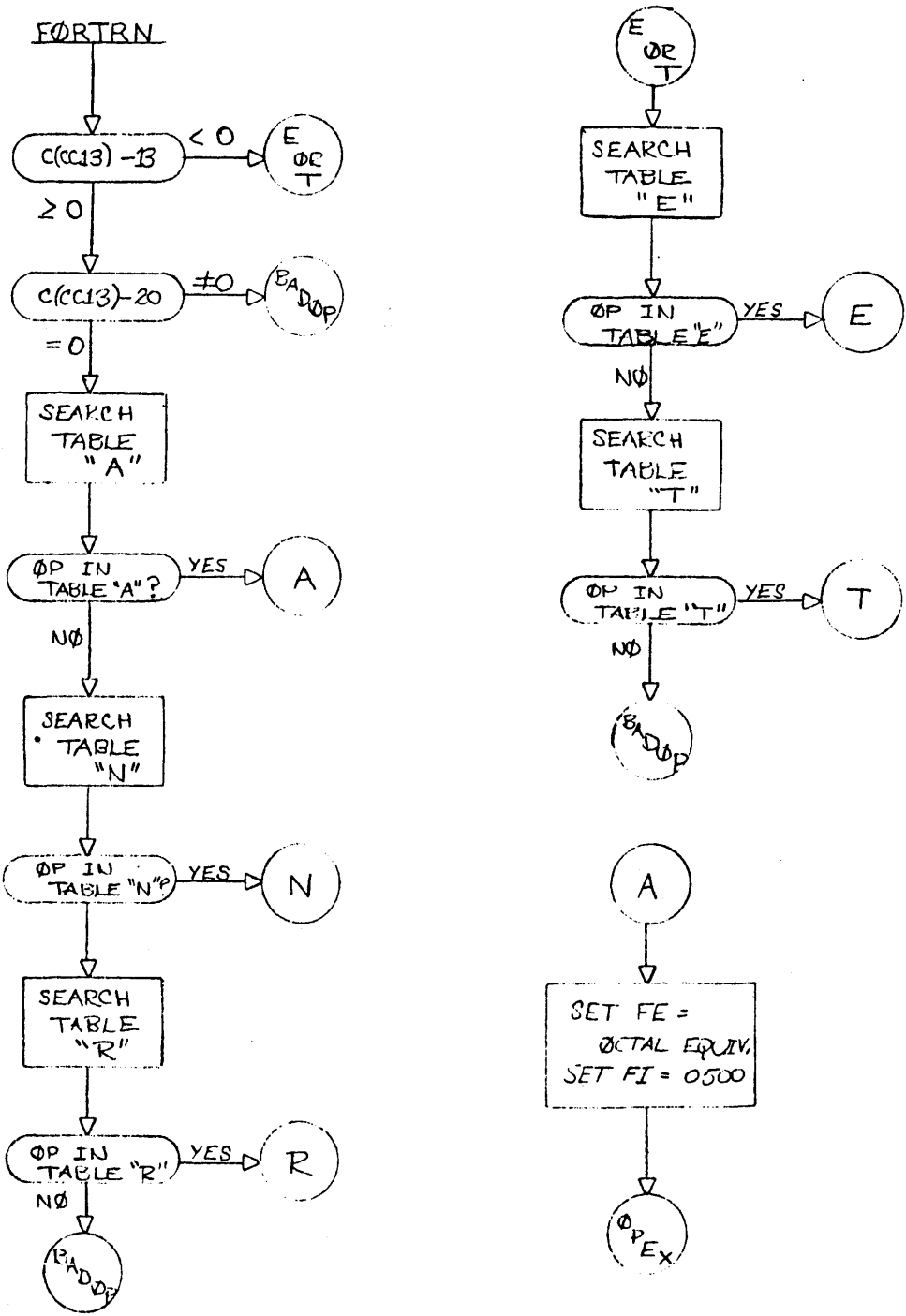
NAME	OSAP-AF MNEMONIC	OCTAL
Load	CAnb	64nb
Load negative, floating	NFnb	65nb
Load negative, integer	NInb	66nb
Load and floating convert	FCnb	67nb
Load negative and floating convert	NCnb	70nb
Normal Boolean	BNnb	71nb
Function erasable Boolean	BFnb	72nb
Boolean shift	BSnb	73nb
Load index	LIXb	74nb
Three word command	TWnb	75nb
Go to (also UP)	GOnb	76nb

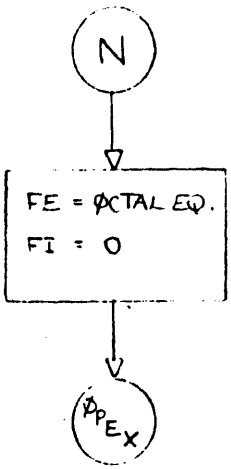
The octal digit "b" denotes the bank of (AA) it must be punched, even if b = 0.

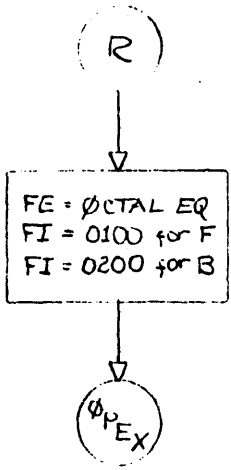
The octal digit n serves a dual purpose, both denoting a pseudo accumulator, and showing whether or not the array option is chosen. Thus n = 0, 1, 2, 3 denotes pseudo accumulators 0, 1, 2, 3 respectively, without the array option, while 4, 5, 6, 7 denote the same pseudo accumulators, with the array option being chosen.

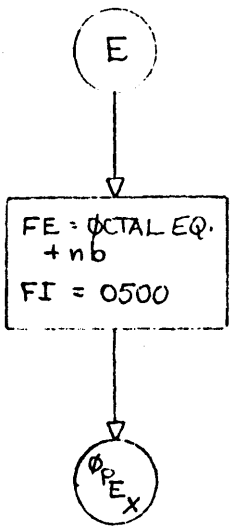
Again no check for illegal n is made although "b" is checked to be $0 \leq b \leq 9$, so that all illegal "b" except 8 and 9 are caught.

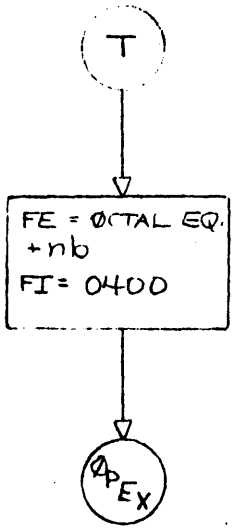
"FØRTRN" IS A RETURN JUMP SUBROUTINE ENTERED AT "BADØP"




N
FE = ØCTAL EQ.
FI = 0
ØPEX


R
FE = ØCTAL EQ
FI = 0100 for F
FI = 0200 for B
ØPEX


E
FE = ØCTAL EQ.
+nb
FI = 0500
ØPEX


T
FE = ØCTAL EQ.
+nb
FI = 0400
ØPEX

OSAP-A SYMBOLS

CC10, CC11, CC12, CC13 Contain the locations of the OP CODE portion of the RCD Card image

OPC1, OPC2 Contain OP Code packed two characters/word

RADOP is the entry to the region in which t e illegal OP flap is set

F_e Is where the octal equivalent of the OP CODE is stored

EI is a word of condensed information for this line of coding as follows:

BIT

ii	1 = Pseudo OP
io	1 = illegal OP CODE
9	1 = Duplicate location symbols
8	OP - Flags
7	OP - Flags
6	OP - Flags
5	1 = - Sign in location field
4	0 = Constant address, i = variable location
3	1 = Signed address
2	0 = + Sign, i = Sign
1	0 = Blank, 0 1 = Octal, 1 0 = Decimal 1 1 = Symbolic Address
0	0 = Blank, 0 1 = Octal, 1 0 = Decimal 1 1 = Symbolic Address

OP - FLAGS

BLR	0	N, D, OR I
END	1	F
EQV	2	B
ORG	3	R
PRG	4	TWO -WORD OP
CON	5	NO ADDRESS REQUIRED
REM, BNK, SUP	6	BLANK OP
BCD, FLX, TTY	7	NUMERIC OP

CARD VERSION OF 160-A FORTRAN

The two magnetic tapes normally used in 160A FORTRAN may be dispensed with by placing the system on cards and using paper tape as intermediate output. Room requirements restrict the card reader to a column reader.

Since all of the magnetic tape handling is done in a routine called BINARY, this routine along with the bootstrap are all that need to be changed to form a card-papertape system. The tape initialization routines have been removed, eliminating the use of an asterisk card; and halts have been substituted for rewinds of the magnetic tapes. The deck is divided into two sections with the source code placed between the two. This division is after file 1. (See insert for card format).

Specifications for BINARY

This version of BINARY is a closed subroutine which replaces the version designed to do all binary (odd parity) tape handling operations. It will be assembled separately from the compiler and will assume that all bank settings are zero upon execution of the routine. The routine will be entered by a JPR to the first location (i.e., 220) with A containing a parameter requesting these functions:

A = 0 - Punch object tape from locations 100 to 217, incl.
(Valid only in Section 1 and File 1 of Section 2).

A = 1 - Read object paper tape into locations 100 to 217, incl.
(Valid after File 1 of Section 2).

A = 2 - Punch end of file frame on paper tape.
(Valid only in Section 1 and File 1 of Section 2).

A = 3 - Halt to place object tape in reader. Paper tape read version of BINARY will be brought in at this time. (Replaces rewind so that any other rewind in Section I must be replaced by a HLT. This will be ignored in Section II).

A = 4 - Not valid.

A =7777 Search forward for an End of File in the system.

A =Address (Anything else)

Load one record of the system beginning at the address in A.

Abnormal Conditions

Parity error on read - stop - no recover.

End file - return with A negative.

Invalid Entry - assume no check is made.

Normal Return

A was		will be	
	0		= 0
	1		= 0
	2		≠ 0
	3		≠ 0
	7777		≠ 0
	Address		= 0

When the first error in a program is found, the request for rewind of the tape has been changed to a halt. At this time all previous intermediate object code is discarded. By placing a 2 in the A register and running, a single frame is punched as was at the beginning of the program.

There is not enough space set aside for BINARY to read and punch paper tape and read cards. Two versions of BINARY are provided. The first is on the bioctal paper tape used to initiate the system. It reads cards and punches paper tape. The second is imbedded in Section II after file 2. It reads cards and paper tape. When the request for rewind of the object code occurs, the first version of binary reads an initialization routine from cards into location 100. This reads in the second version of BINARY from cards and halts so that the paper tape intermediate object code may be placed in the photoreader. This procedure is the same whether diagnostics have occurred or not.

What has been described are, with one exception, the only differences between the cardpaper tape system and the magnetic tape system. This exception is the use of jump switch 1 to change the object computer. Because this change requires magnetic tape (system deck) to be read and rewound, this option has been removed. The bank change is possible, however. Any changes to the object computer must be made before the deck is punched.

PAPER TAPE FORMAT

Each record is 243_8 frames long. The first 240_8 frames (120 words) are the contents of locations 0100-0217 inclusive. Frame 241_8 is a record mark, 127. Frames 242_8 and 243_8 are the 12 bit check sum of the 120_8 words. An End of File frame is 177.

TAPE TO CARD OPERATION

1. Set load-clear switch to clear.
2. Set A to 0000
3. Set jump switch 1.
4. Place blank cards to be punched in the 523 card punch hopper. Turn punch on.
5. Place tape to be punched on unit 1.
6. Set run step switch to run.
7. Normal halt is at P = 0301
8. Error stops

P = 0150
No jump switch was set. Set jump switch 1 and run.

P = 0304
Parity error on tape read. No recovery is possible.

P = 0567
The punch is not ready. Correct condition and run.

CARD FORMAT

Col. 1:	7 bits (row 12 - row 4) for the word count, row 7 and 9 indicate binary cards. Row 8 will be punched if this is the last card in a record. Row 6 is punched on every other card. Note that Column 1 is 4505 or 4515 octal except for the last card in the record.
Col. 2:	Check sum of column 3-76 inclusive.
Col. 3-76	(less on last card in the record). The actual words.
Col. 77-80	Sequence number.

An end of file card has 0177 as Column 1. The rest of the card is ignored.

GENERATING A SYSTEM DECK

Since a tape-to-card and verify program is available (chapter 2 of the operating instructions), the program used to generate the system deck is a special tape copy program. This program does not copy the first two records (initialization routines for magnetic tape) on file six, inserts the second version of BINARY and its bootstrap and sets the diagnostic halt. In using the tape-to-card and verify program, the first octal digit of the A register should be set to 5. Once the deck has been produced the last card, an End of File card, should be removed. File 1 (about 100 cards) should be carefully marked or reproduced on a different color since the source deck (with a single blank card after the end card) follows file 1.

INITIAL BOOTSTRAP

The first version of BINARY is on the same bioctal paper tape so it is also in core.

1. Output 77_8 blank frames as leader.
2. Output an EØF using BINARY as a dummy frame so that leader may be skipped.
3. Read the first record using BINARY beginning at 0400. This is Pass 1 Part 1.
4. Jump to 0400.

BINARY, FIRST VERSION

This version does not read paper tape. It is used while intermediate object is being punched.

ENTER 1. Save A register parameter in TEMO

2. Set up exit

3. If (TEMO) = 0 go to 7

4. If (TEMO) ≠ 2 go to 15

WEOF 5. (TEMO) = 2. Write an EOF on tape, i.e., punch 177.

6. Go to 23

PTWR 7. (TEMO) = 0. Punch a record from 0100-0217 inclusive.

8. Set checksum (loc. 0220) to zero.

9. Initialize pickup address to 0100.

PTWR 10. Punch out word. No 7th level punch.

11. Bump address. If less than 0220 go to 10.

12. If greater than 220 go to 23.

13. Punch record mark (127).

	14.	Go to 10.
NEXT	15.	If (TEMO) ≠ -0 go to 24.
GOEOF	16.	Read cards until EOF card (0177 in col. 1) is found.
	17.	Lock out timing fault
GOEOF	18.	Set A to -1. Go to 23.
BTSTRP	19.	Bring in second bootstrap. Save return address in 0073.
	20.	JPR ENTER (BINARY) with A = 0100, i.e., load second bootstrap at 0100.
	21.	Jump to second bootstrap
	22.	Set A to 0.
	23.	Exit.
NEXTA	24.	If (TEMO) = 3 go to 19.
RDCRD	25.	Read a record from cards. A is initial address. Set up STM instruction.
RDCRD	26.	Start the read. Store col. 1 in TEMO
	27.	If col. 1 is 0177 (EOF) go to 17.
	28.	Set up SCN instruction for rows 6, 7, 8, 9, if present.
	29.	If no 7-9 punch, go to 41.
	30.	Zero the checksum (CHKSUM)
SC	31.	Set up COUNT (12 row through col. 4) as negative number.
	32.	Store col. 2 in TEMO (checksum from card).
RDCRD	33.	Read a col. and store it.
	34.	Add it to CHKSUM, bump address and COUNT
	35.	If COUNT is not zero, go to 33.
	36.	Lock out timing fault.

37. If calculated check sum is not equal to check sum from card, go to 40.

38. If col. 1 row 8 was punched go to 22.

39. Go to 26.

40. Check sum error. To try again replace card, adjust storage address by count and run from 26.

41. Reader failure. See 40.

SECOND BOOTSTRAP

This will be in core at 0100 and will be entered from the first bootstrap. The return address is in 0073. The second version of BINARY is read into 0220 and a halt made so that the intermediate object code paper tape can be placed in the photoreader.

BOOT4 1. Set up return.

2. Read in BINARY. Initialize address at 0200.

BOOT5 3. Read in col. 1 and save in YTEMP and COLIHD.

4. Set up SCN instruction at SCX to take card of rows 6, 7, 8, 9, if present.

5. If 7-9 punch is not present, halt. Run from 3.

6. Set YCHKSM to zero.

7. Set up. Y COUNT (negative).

8. Store col. 2 in YTEMP

BOOT 9. Read a col. and store it.

10. Add to checksum, bump address and count. If count is not zero, go to 9.

11. If YTEMP and YCHKSM do not agree, halt. After modifying address according to the count, rerun from 3.

12. If this is not the last card in the record, i.e., check 8 row punch using SCX, go to 3.

13. Punch trailer. Halt.

14. Skip leader

15. Set up jump in second BINARY which checks alternating punch in row 6 col. 1. If last card read had no punch, jump is okay as is, i.e., go to 17.

16. Change jump to ZJF.

17. Exit.

SECOND VERSION OF BINARY

This version reads both cards and paper tape. There is room to check the alternating punch is row 6 col. 1. Some low core is available now.

ENTER 1. Save A register parameter in TEMO

2. Set up exit.

3. If TEMO ≠ 1, go to 13.

RDPT 4. Read a record from paper tape into 0100-0217 inclusive. Zero the checksum.

5. Initialize address and EOF jump at step 6 jump. An EOF must be the first frame encountered.

RDPT1 6. Read a frame. If EOF go to 16.

7. NOP/EOF test in step 6

8. Store word. Add it to check sum and bump address.

9. If address is not 220, go to 6.

10. Check record mark. Halt is error.

11. Check check sum. Halt if error.

12. Go to 17.

NEXT 13. If TEMO = 3 go to 18.

14. If TEMO ≠ -0 go to 17.

GOEOF 15. Read cards until an EOF card is reached (177 in col. 1).

16. Set A to -1.

17. Exit.

RDCRD 18. Read a record from cards. Store address in TEM1

RDCRD1 19. Read col.1 and store it in TEMO.

20. If EOF go to 16.

21. Set up SCN instruction at SC to take care of row 6, 7, 8, 9, if present.

22. If 7-9 punch is not present, halt.

23. Look at row 6. This will alternate as 0 and 1. XJUMP is changed from ZJF to NZF accordingly. Halt if error.

24. Zero out CHKSM.

25. Set up COUNT as negative number.

26. Read col. 2 (checksum) from card. Store in TEMO.

27. Read and store a col. Add it to checksum. Bump COUNT.

28. If COUNT is not zero, go to 27.

29. Lock out timing fault. Check check sum. If error, halt.

30. If row 8 of col. 1 is not punched (check SC) go to 19.

31. Set A to 0

32. Go to 17.

COPY PROGRAM FOR GENERATING SPECIAL TAPE

As explained, once the new tape is generated 5 files are punched on cards. In copying, the magnetic tape system tape is placed on unit 1. The tape to be generated is placed on unit 2. The copy program contains the second bootstrap and the second version of BINARY. These are assembled at 0200 and 0320, respectively, i.e., all addresses are off by 100.

1. Rewind the tapes and set them to binary.
2. After initializing counters, skip the first two records. These are the magnetic tape initializating routines.
3. Copy records 3 (Pass 1 Part 1) and 4 (Pass 1 Part 2).
4. Read in record 5 (Pass 1 Part 2) and change location 4054 (now at(1) 3016). Output the record.
5. Copy rest of file 1 and all of file 2.
6. Output second bootstrap and second BINARY as two records.
7. Copy files 3, 4, and 5.