INTERNAL MAINTENANCE SPECIFICATIONS

6000 Series FORTRAN EXTENDED

VERSION 4.0

© COPYRIGHT CONTROL DATA CORP. 1971

Contained herein are Software Products
copyrighted by Control Data Corporation.
A reproduction of the copyright notice must
appear on all complete or partial copies.

TABLE OF CONTENTS

ction		
1.	Introduction	1
2.	FTNTEXT	6
3.	FTN	18
4.	LSTPRO	25
5.	OUTPTK	36
6.	PS1CTL	40
7.	STMTP	49
8.	ENDPRO	54
9.	SCANNER	65
10.	CONVERT	90
11.	DATA	94
12.	ERPRO and FORMAT	109
13.	LISTIO (PRINT)	129
14.	ARITH	153
15.	ASFPRO	190
16.	CALL	197
17.	GOTO	204
18.	DOPROC	214
19.	DPCLOSE	226
20.	DECPRO	242
21.	PH1CTL	257

22.	Code Generation Technique	268
23.	CLOSE2	273
24.	FAX (FTNXAS)	276
25.	REFMAP	316
26.	POST	321
27.	APLISTE	327
28.	PRE	333
29.	READRL	340
30.	DOPRE	346
31.	PROSEQ	379
32.	JAMMER	389
33.	SQUEEZE	411
34.	OPTB	417
35.	SQZVARD	426
36.	MACROE	429
37.	BUILDDT	433
38.	PASS15	438
39.	MACRS	441
40.	BLDSEQ	444
41.	GET	447
42.	CONNECT	449
43.	SYMFND	451
44.	SYMDEF	453
45.	DOOPT	455

46.	FNDLOOP	457
47.	FNDINVR	459
48.	MOVINVR	467
49.	USEDEF	469
50.	ADDROW	474
51.	SETBIT	477
52.	TESTBIT	478
53.	PUTMS	479
54.	GETMS	481
55.	UPDOWN	483
56.	READR	484
57.	CHECK	487
58.	PASS14 (DBGPHCT)	490
59.	PUT	502
60.	PUTUPDT	510
61.	BUGACT	513
62.	GETOUT	515
63.	TURNON	517
64	BUGSOUT	523
65.	BUGCON	526
66.	TURNOFF	534
67.	DEBUGER	537
68.	BUGPRO	541
69.	PUTIN	546

70. SETARR 548

71. BUGCLO 552

APPENDICES

R-list Language Description 556

FORTRAN Extended I/O Calling Sequences 568

Subroutine Linkage and Formal Referencing 572

IMS Introduction

FORTRAN Extended is a two pass compiler; the input is FORTRAN source card images and the output is an assembly language program. Assembly is by FTNXAS, a one pass assembler which recognizes a subset of the COMPASS language, (this assembler is embedded in Pass 2 of the compiler.)

PASS 1 is divided into two phases: the FTN control card, the header card, and declarative statements are processed in Phase 1, executable statements in Phase 2.

During Phase 1, the header card is processed; the COMMON, DIMENSION and EQUIVALENCE information is held in linked lists in working storage. These lists are processed at the end of Phase 1 and COMPASS instructions are issued for storage allocation.

Phase 2 converts the executable statements to an intermediate language, R-list, and when the END card is seen storage is issued for usage defined variables and program constants.

Thus Pass 1:

1) Converts all source statements to an intermediate language, E-list.

2) Forms the symbol table.

3) Issues COMPASS instructions for program identification, variable initialization and storage allocation, program constants, traceback and formal parameter initialization.

4) Produce the R-list intermediate file for executable statements.

Pass 1 Task Summaries

FTN is the main controlling routine. It loads the overlays, cracks the FTN control card, contains the I/O buffer area and the general purpose I/O routines.

SCANNER transforms all source statements into the intermediate language, E-list, and determines statement type. Basic syntax errors are diagnosed.

OUTPTK provides FORTRAN formatted I/O facilities for portions of the compiler coded in FORTRAN.

LSTPRO locates in or enters into the symbol table a given symbolic name or label.

CONVERT converts the display code representation of a constant to its internal binary form. Illegal constants such as those containing too many digits, non-octal digits in an octal constant or constants out of range are diagnosed here.

ERPRO saves diagnostic information accumulated during Pass 1 for processing in Pass 2.

FLY contains a transition diagram state table used in parsing format statements.

DATA processes DATA statements and produces appropriate COMPASS code for data initialization.

DOPROC examines DO statements, DO-implied lists, statement numbers, statement number references and integer variable definitions and references. Determines the characteristics of DO's and index functions, diagnoses nesting and the use of statement numbers and generates R-list defining the beginning and end of each DO loop and DO-implied list.

STMTP is the miscellaneous statement processor. Statements processed are STOP, PAUSE, NAMELIST, ENTRY.

CALL processes all FORTRAN CALL statements.

ARITH processes replacement statements and translates into R-list the arithmetic, logical, relational, or masking expressions that legally appear in any statement.

ENDPRO generates R-list for exit code, issues storage for variables and arrays, processes EXTERNAL names, and sends CONLIST, the program constants, to the assembly code file.

PS1CTL is the primary controlling routine for all pass one, phase two processing. Each of the statement processors is called from and returns to PS1CTL.

PRINT processes all I/O statements.

GOTO processes all GOTO type statements and the ASSIGN statement.

ASFPRO processes all statement function definitions by saving the text, and processes all statement function references by expanding the E-list and inserting the text.

DPCLOSE collapses the linked lists of declaratives generated during Phase 1 into static tables. Diagnostics which can only be produced when all declaratives are known are issued.

DECPRO processes declarative statements. Declarative information is held in linked lists until Phase 2. Header statements cause COMPASS instructions for program initialization to be issued.

PH1CTL handles routine header cards and serves as a controlling routine for all phase one processing.

Pass 2 may be divided by function into two principal areas, namely, the pre-processing of R-list and the actual code generation. The former phase basically entails accumulating R-list for optimization, usually one sequence, and provides for the expansion of all R-list macros. The various optimizing routines are then called for code generation. Control, in turn, reverts back to the first area and continues to fluctuate between the two functions until all R-list on the file has been decoded. The variable dimension and formal parameter code is sent to the COMPASS file. If the C option was selected, Pass 1 is loaded and receives control if more FORTRAN programs are present; otherwise COMPASS is loaded to assemble the contents of the assembly code (COMPS) file. If the C option was not selected, the generated code is assembled by FTNXAS and the binary sent to the binary file.

PRE is the main controlling routine. It calls other Pass 2 routines and also defines a sequence. It puts out inactive label names to COMPASS, passes control to PROSEQ for optimization, and detects the end of R-list.

READRL obtains R-list file input for Pass 2. It also receives as input macro expansions from MACROE. When called, it returns either a single entry plus descriptor or an entire macro reference.

DOPRE examines DO begin and DO end macro references, standard index function macro references and all R-list instructions generated within the innermost loop of a DO nest provided the loop is well behaved. R-list instructions are generated to count DO-loops, reference standard index functions and to materialize the control variable when necessary.

MACROE expands macro references into normal R-list form.

The following routines combine to perform the code optimization functions:

PROSEQ calls the optimizing routines and also handles the cutting down of a sequence should tree complexity or working storage limitations become a problem.

SQUEEZE marks redundant instructions for elimination.

PURGE physically eliminates the instructions marked by SQUEEZE.

BUILDDT forms a dependency tree from the squeezed sequence. The tree reflects precedence relationships within the sequence.

OPT is the code selector. Having considered timing aspects and register usages, it calls POST with the particular instruction to be issued.

POST transforms the R-list instruction into a COMPASS card image, and eventually issues the code for the sequence to the COMPASS file.

FTNXAS is the compiler's specialized one pass assembler.

REFMAP produces the cross reference map.

The following routines are involved in closing Pass 2.

SQZVARD eliminates redundant store operations from the VARDIM initialization sequence and transforms corresponding storage allocation to the COMPASS file.

APLISTE converts APLIST entries to COMPASS card image then puts them in the COMPASS file.

CLOSE2 performs the close out processing for Pass 2. Both FTNXAS and the reference map processor, REFMAP, are called from CLOSE2. A SUB macro reference is generated for any formal parameter not referenced in the program.

JAMMER restructures the tree in case PROSEQ reduces the sequence to a single statement and still cannot issue it. If necessary, JAMMER can subdivide a statement issuing intermediate stores in order to issue the statement.

Record Manager Usage

FORTRAN Extended Version 4.0 includes the 6000 Record Manager for input/output during compilation. Usage of the Record Manager facility will be a user option, selectable at installation time. Selection is accomplished by setting a single flag, CP=RM, contained in the common deck OPTIONS (called during FTNTEXT updating). When the Record Manager is employed, field length requirements will increase by approximately 5000B words for any compiler mode.

The general installation approach has been to convert the actual I/O interface suboutines, located primarily in FTN and LSTPRO, to use conventional Record Manager macro calls. In this way, FTN I/O macros have been disturbed as little as possible, although some changes have proven mandatory. Certain buffer allocations have been revised; new File Information Tables have been added to conform to Record Manager requirments.

FTNTEXT

1.0 General

FTNTEXT is a text file used in assembling FTN Version 4.0. It contains macros for performing commonly occuring tasks and symbol definitions used in accessing bits and fields. It is always used as a systems text for assembling the comiler.

2.0 Generating FTNTEXT

The systems text file may be created by assembling the text of FTNTEXT and EDITLIBing the resulting binary file, or by using the assembled binary as a local text file.

3.0 Macros

3.1 I/O and Overlay Control

3.1.1 LOVER OVLmn

This macro produces a call to the routine in FTN used to load compiler overlays. The parameter consists of the characters OVL followed by two digits specifying the overlay level. The name OVLmn must also be defined as entry point in FTN and contain the name of the overlay. Control is transferred directly to the entry point of the overlay after loading. Thus, to load and enter pass 2 of the compiler it is sufficient to write

```
LOVER   OVL12
```

where the entry point OVL12 contains CLOSE2$ in 7L format.

3.1.2 SYSREQ request

All PP calls should be made using this macro. The formatted request is the only parameter.

3.1.3 File names and unit numbers

Each file used by the compiler has a number associated with it. For unit n, the file name and FET address are in location RA+n+1 in the following format:

```
42/7L file, 18/FET address
```

All units should be addressed symbolically for cross reference purposes. The symbolic name is derived by prefixing the file name, with the characters U.. Thus, the input file should be referenced as U.INPUT.

In addition to the unit numbers, the first word address of a file FET is an entry point with the name F.file (except OUTPUT).

3.1.4 REWIND file, table

REWIND will reset either a file or a table to beginning of information. In the case of a file, the buffer is flushed with an end-of-file write and the file is then rewound. When the second optional parameter is present, the file will be rewound if it has spilled to disk. Otherwise, the status will be set to end of record read and control will be returned.

Examples:

```
Rewind LGO file
     REWIND    LGO
Rewind COMPS as a table
     REWIND    COMPS,TABLE
```

3.1.5 OPEN file, code

OPEN causes an open call to be issued on the file specified by the first parameter. The second parameter selects the type of open to perform. Register defaults are X1 for the file address and X2 for the code.

Example:

```
OPEN INPUT, 100B
```

3.1.6 CLOSE file, code

CLOSE performs a close call on the specified file. See the OPEN call for details.

3.1.7 DO.IO file, code, return

DO.IO is used to initiate an I/O operation on a file. The code should be specified symbolically and may be one of the following:

READ	buffer read
WRITE	buffer write
READS	read skip
EOR	end of record write
EOF	end of file write
BKSP	backspace PRU
REW	rewind
CLUNL	close unload

If the code is preceded by a minus sign, control is not returned until the operation completes.

If a return is selected, control is transferred to it. Otherwise, flow resumes at the next statement.

Examples:

a) Initiate a read on the input file

```
DO.IO       INPUT, READ
```

b) Endfile the LGO file and wait for completion

```
DO.IO       LGO, -EOF
```

c) Close unload the OPT file and transfer control to the label STOP

```
DO.IO       OPT, -CLUNL, STOP
```

3.1.8 READL file, FWA, wordcount

READ wordcount number of words from the designated file to the address specified by FWA.
Default registers are:

B6 = unit number
B7 = FWA
B1 = word count

On return, B1 contains word count minus the number of words read.

3.1.9 LISTL FWA, wordcount

LISTL places a line image on the OUTPUT file. The line starts at FWA and is wordcount number of words long. Default registers are:

B7 = FWA
B1 = wordcount

3.1.10 WRITE file, FWA, wordcount

WRITE places information in working storage on the selected file. The second parameter specifies the starting location of the information in working storage and the third parameter gives the number of words to transfer. Default registers are:

B6 = unit number
B7 = FWA
B1 = word count

Upon return, if B6 is equal to the unit number, a call to CIO was made because the buffer threshhold size was exceeded.

3.2 Compiler Functions

3.2.1 name ENTRY. data

The ENTRY. macro creates an entry point designated by name which contains the item specified by data.

Examples:

Define the entry word of a subroutine which is to be entered by external calls.

```
SUB       ENTRY.
```

Define an entry point OVL12 containing the name of the pass 2 overlay.

```
OVL12     ENTRY. 7LCLOSE2$
```

3.2.2 MOVE wordcount, FWA, destination

This macro should be used to move a block of data. Word count number of words are transferred, starting at FWA, to the destination. The move routine ensures that the operation is non overlapping. Therefore, MOVE may be used to move tables up or down in memory. Default registers are:

X1 = wordcount
X2 = FWA
X3 = destination

Example:

Move a table up ten words from its present location (specified in X2). The table length is specified in B4.

```
MOVE      B4,, X1+10
```

3.2.3 SYMBOL nameloc, return

SYMBOL causes a symbol table search for the name specified. Nameloc is the address of a word containing the symbol in 8R format. If the symbol is not found, control is returned to the word following the SYMBOL call and the symbol has been entered in the table. If the search finds the symbol, control is transferred to the location two words after the SYMBOL call. If return is specified, control is transferred to return for not found and return +1 for found. Defaults assume the name to be in X1.

Examples:

Enter TEMPA0. into the symbol table

```
SYMBOL =8RTEMPA0.
```

Determine if the symbol in X1 is in the table

```
SYMBOL
EQ   NO
EQ   YES
```

3.2.4 LABEL nameloc, return

LABEL performs the same function as SYMBOL except that the object of search is a label rather than a symbol. For details, see the description of SYMBOL.

3.2.5 ADEXTS nameloc

ADEXTS places an external symbol into the symbol table. Nameloc is the address of a word containing the name in 8R format.

3.2.6 ADDREF ordinal, type

ADDREF adds a reference to the refmap file (should only be called if RSELECT is non-zero indicating R=1 or 3). Type reflects the type of the reference. Possible character string values are

REF	reference
DEF	definition
FREF	reference as a file name

If type is omitted, REF is assumed.

Ordinal is either a memory location or register containing the symbol table ordinal to be entered as a reference.

Examples:

Add a reference to the variable upon return from SYMBOL (ordinal is in B1)

```
ADDREFF   B1, REF
```

Enter a definition for the symbol whose ordinal is in TEMP

```
ADDREF    TEMP, DEF
```

3.2.7 X BIT Y

This macro produces a set symbol named X with the value of 2**Y for Y less than 22.

3.2.8 RMHDR macnum, length

RMHDR creates a header word for an R-list macro. Macnum is the macro number and length in the word count for the rest of the macro excluding the header word.

Example:

Define the header word for a load macro

```
LOAD    RMHDR    107B,3
```

3.2.9 OUTUSE name

OUTUSE issues a USE name line to the COMPS file. It is used in switching from one local relocation base to another. If the name indicated is the current block no line is issued.

Example:

Switch to the CODE. block

```
OUTUSE CODE.
```

3.3 Table Manager Macros

3.3.1 ADDWD tnam, word

Adds a word to end of a managed table. The table is specified by tnam and the second parameter is the word to be added. It may be either a register expression or a memory location.

3.3.2 ALLOC tnam, nwds

Allocate nwds to the table specified by tnam. The number of words may be either positive or negative.

3.3.3 ALLAE tnam

Allocate almost all available core to table tnam.

3.3.4 TABLES A,B,C,D,E,F,G,H,I,J

Defines externals of O.A and L.A for up to 10 table names. O.A is the external word containing the table origin and L.A holds the length.

3.4 DEBUG mode macros

3.4.1 CFO context

CFO is used to compare options selected on the debug cards with actual usages. Any conflicts in use are diagnosed. Context is the character string VAR if the symbol occurred as a variable or array and EXT if it occurred as a subroutine or function reference.

3.4.2 DBGERR message

Issue an error message in debug processing.

Example:

```
DBGERR          BAD OPTION ON DEBUG CARD
```

3.4.3 CALLF name, RESET

CALLF links from a COMPASS routine to a FORTRAN routine with no parameters. If RESET is present, B5 is set to 1 upon return from the FORTRAN routine.

Example:

Call the debug processing routine BUGPRO

```
CALLF           BUGPRO
```

3.5 Test mode macros for compiler checkout

3.5.1 DEBUG

If this macro is called in a COMPASS routine, all subsequent macros in this section will become defined.

3.5.2 SNAP fwa,lwa,len,ll,ul,inc,name,no regs,no head

The SNAP macro can be used to dump a printout of the registers or core to the output file and continue program execution without destroying any registers.

The arguments are

fwa	- first word address of the area to be dumped
lwa	- last word address of the area to be dumped
len	- number of words to dump starting from fwa
ll	- begin snapping on the ll time through this snap
ul	- snap until ul times through this snap
inc	- snap every inc time through this snap
name	- up to ten display code characters identifying the snap
no regs	- non-zero indicates no register snap desired
no head	- suppress identification line

If an argument is preceded by a minus sign, it is assumed to be a word containing the value of the argument.

3.5.3 REGS name

Snap the registers on the first thousand times past this snap. Identification is given by name.

3.5.4 FCOPY file, prefix, index

FCOPY rewinds and copies a record of the specified file to the file SYMTAB. The character string given by prefix and the value of index are appended to the start of the record.

3.5.5 TABDMP prefix, index, fwa, lwa

TABDMP writes the table bounded by fwa and lwa to the SYMTAB file. The character string prefix and the value of index are appended to the beginning of the record.

3.5.6 ELIST name

Snap the E-list during pass 1 identifying the output with the specified name.

3.5.7 SNAPT tbl, name

Snap the table tbl and identify the snap with name. The table origin and length must be in locations named O.tbl and L.tbl.

3.5.8 ONSPY FWA, lwa, name, binw

Initiates the PP program SPY to perform P register sampling. Fwa and lwa give the bounds of the area to watch. Name can be up to eight characters to be displayed while spying. Binw specifies the width of the bins used in the register sampling. Legal values are 10B, 20B,40B, 100B. Default size is 40B.

3.5.9 OFFSPY

This call to SPY turns off P register sampling.

3.6 Utility Macros

3.6.1 Integer Multiply and Divide

These two opdefs are provided for coding convenience. The divide opdef will destroy the contents of B7. Both opdefs destroy the contents of the operand registers.

3.6.2 Left Shift Opdef

This redefinition of the constant left shift instruction ensures that negative shift counts are treated as right circular shifts. Whenever possible, bit positions should have a name assigned to them so that they are included in the cross reference listing and the bit position can be relocated easily if necessary.

Examples:

Test the bit named P.BIT for an on condition. Assume that the word containing the bit is in X2.

```
LX2     59-P.BIT
NG      X2,ON
```

Shift the bit named P.ONE to the position where the bit named P.TWO is currently.

```
LX2     P.TWO-P.ONE
```

3.6.3 length MICCNT character string

The MICCNT macro returns a value of the length of the character string passed as a parameter. The character

string should not contain the equivalence sign (0-6-8 punch) or be longer than one hundred characters.

3.6.4 name DECMIC value, digits

Form a micro called name, which is the decimal representation of the value parameter. If the parameter digits are present, then the micro will have at least that many characters. Otherwise, leading zeros will be suppressed.

3.6.5 name OCTMIC value, digits

OCTMIC performs the same function as DECMIC except that an octal representation is generated.

4.0 Symbol Definitions

Whenever possible bits and fields should be accessed symbolically. To facilitate this, FTNTEXT contains definitions for commonly used fields and their sizes. The format of the symbols is X.name where X is a one or two character prefix and name is the name associated with the symbol. Commonly used prefixes are:

P	-	the position of the base of a bit field in a word
O	-	table origins
L	-	the length of the bit field or table
S	-	table sizes
V	-	the values of bits where P.name is less than 17
C	-	CIO codes
T	-	values the type field in the symbol table may assume
EL	-	values of E-list codes
F	-	file names or first word address symbol

Examples:

Test word B of a symbol table entry to see if it is a DO generated label. Assume word B is in X2.

```
LX2     59-P.GEN  move the bit to the sign
NG      X2,DOGEN  if DO generated
```

Extract the type field from word B of the symbol table.

```
MX0       60-L.TYP
AX2       P.TYP          move to bottom of word
BX6       -X0*X2         extract the field
```

Test the referenced as statement number and referenced as format number bits. Word B is in X2.

```
LX2       59-P.RSN
NG        X2,YESRSN      if RSN bit is on
LX2       P.RSN-P.RFN
NG        X2,YESRFN      if RFN bit is on
```

FTN

1.0 General Information

FTN is the 0,0 overlay of the FORTRAN Extended compiler. The primary functions of this routine are system interface and I/O processing, control card cracking and compiler initialization.

2.0 Entry points

2.1 FTN

This entry is from the operating system loader. It is the entry to the code which will crack the control card and perform compiler initialization.

2.2 LOVER

This entry issues a call to the loader to load a compiler overlay. Control is transferred to the overlay loaded.

2.3 LDPH1

Reload the 1,0 overlay. Used upon return from COMPASS.

2.4 LDCOM

Load the COMPASS assembler. This is used for intermixed COMPASS programs as well as when the C option is selected.

2.5 FTNEND

This entry terminates compilation. The output and binary buffers are flushed, scratch files evicted and CPU time calculated. If the G option was selected, we call the loader to execute the binary. Otherwise, END is placed into RA+1.

2.6 CIO1.

This routine makes a call to CIO. Parameters are

X1 FET address
B6 Return address
X2 I/O function code

If X2 is less than zero, then the call is made with auto recall.

2.7 REWIND

This entry will rewind the file whose unit number is in B6. Before rewinding, an end of file is written to flush the buffer.

2.8 SETFET

This routine will initialize a compiler FET. Calling parameters are:

X1 FET address
X2 FWA of buffer
X3 size of buffer

2.9 WRWDS

WRWDS performs all file writing done in the compiler on buffered files. Calling parameters are:

B1 word count
B6 file number
B7 FWA of area to receive data

On exit, B1 = word count - words read

2.10 REWINDT

This routine rewinds a tabled file. If the file has spilled to disk, REWIND is called. Otherwise, the file is set to end of record status.

B6 = file number

2.11 MVWDS

A general purpose move routine. It can be used to move data from point A to point B when there is no chance of overlap. To shuttle tables up or down, the routine called MOVE in LSTPRO should be used.

2.12 Flag entry points

A number of locations in FTN are flag cells used in the parts of the compiler. These entry points are detailed below.

SAVLINE — Address of an area that can hold 20 line images. When L=0 is selected, this is used in listing only lines in error. This area should be used only if L=0 since it occurs in the middle of the ordinary output buffer.

F. name — First word of the FET for each of the compiler files. Name may be INPUT, LGO, COMPS, RLIST, RMAP, OPT, and DEBUG.

FL — Available field length.

BERRORF — Batch error flag.

MACFLAG — UFLAG or'ed with OLIST.

O.GCON — Origin of the global constant table (DEBUG mode).
L.GCON — Length of the global constant table.

GL.IND — Length of DEBUG random file index.
LASTREC — Last record cell for DEBUG mode.
GL.DRL — Length of the global debug routine list.
GL.DVL — Length of the global debug variable list.

DFLAG — Non-zero and holds the name of the debug file when D is selected.

DIRECT — Direct usage flag for LCM mode.

ZFLAG — Non-zero if zero word load desired for external calls without parameter lists.

QFLAG — Non-zero if quick mode compilation is selected.

CBNFLG — Call by name flag is set non-zero if T is specified.

PLIMIT	Compiler default print limit value.
OPTLVL	The value is zero, one or two depending on OPT = n.
ROPFLAG	Non-zero if rounded arithmetic is selected.
NOLSFLG	Non-zero if a listing is to be produced.
NASAFLG	Non-zero if ANSI diagnostics are selected.
R=FLAG	Reference map level. Values are 0,1,2,3.
RSELECT	Non-zero if R greater than or equal to 2.
OLIST	Non-zero if O was specified.
SUPIDFL	Non-zero if N was selected (suppress informative diagnostics).
UFLAG	Non-zero if E was selected.
CAFLAG	Non-zero if C was selected.
F.LFN	Word B bits for file names.
COMPMSG	Entry for COMPILING XXXXX message.
LIBRARY	Library name for compiler overlays.
OVLmn	Overlay names for compiler overlay.
L.TITLE	Length of primary title line.
O.TITLE	Start of title line.
TITLE1	Second word of title line.
CCOPT	Start of control card option area of title line.
DATE	Date in title line, DATE+1 contains time.
PAGE	Page number.
STITLE	Subtitle area (for REFMAP).
LMAX	Lines per page.

MACCLG

LCNT Lines remaining on this page.

3.0 Messages and Diagnostics

COMPILING XXXXXXX

Appears on the B display during compilation.

nnn.nn CP SECONDS COMPILATION TIME

Appears in the dayfile at end of compilation if the CTIME option is active.

CANT LOAD XXXXXXX

Issued if a compiler overlay cannot be loaded.

* POINTS TO FTN CONTROL CARD ERROR

Issued when a control card error is detected. The * approximately locates the field in error.

FTN NEEDS nnK TO EXECUTE, JOB ABORTED

Insufficient field length for the compiler.

DEBUG MODE IMPLIES OPT = 0

Issued if D and an OPT level other than zero are selected.

4.0 Environment

FTN 4.0 is set up for the standard interface to COMPASS. This includes the locations of the INPUT, OUTPUT and binary FETs. The 0,0 overlay will fit entirely below 3000B since this is the origin of the first COMPASS overlay.

5.0 Processing

5.1 Control Card Cracking and Processing

Upon entry from the system loader, the FTN control card is burst in 1R format into a working storage buffer.

Bursting continues until a legal terminator is found, and may involve processing continuation control cards. Blanks are squeezed out of the control card image and not placed in the working storage buffer, but a blank count is provided in packed exponent form for each character indicating the number of blanks preceding that character. Blanks may be freely embedded in the control card statement, and are ignored during option processing.

Option recognition is based on the first character of the option. A first-character jump table leads to the actual parameter processing. For simple one letter options, the appropriate compiler flag is set. For more complicated options, like LXRON, option recognition takes place within the first-character code for that option set. Checks are performed to ensure correct syntactic form and option separation. The routine AFN is called to pack up the filename following the equal sign for options of form option=lfn. Once an option has been selected and processed, an error jump is placed into the jump table for that option, thereby causing a control card error to occur if that option were again specified.

5.2 Compiler Initialization

5.2.1 Buffer Allocation

First, we allocate buffer space to each of the files that will be active. If insufficient core is available, the compilation is aborted. Scratch file buffer sizes are allocated as follows:

(MIN.FL = 40K)
201B word buffers for MIN.FL≤fl<MIN.FL+3K
Standard buffers for MIN.FL+3K≤fl<MIN.FL+6K
50 percent of excess core to buffers for MIN.FL+6K≤fl<MIN.FL+14K
75 percent of excess core to buffers for MIN.FL+14K≤MIN.FL+26K
25 percent of excess core to buffers for FL≥MIN.FL+26K

This algorithm was chosen because:

1. Space needed to compile is MIN.FL+3*number of symbols.

2. Since most programs will have less than 1000B symbols, they will compile in MIN.FL+6K.

3. For MIN.FL+6K to MIN.FL+26K, we are given most of the space to the buffers so that long subprograms will not overflow to disk.

4. For greater than MIN.FL+26K, most of the space is allocated to the working storage area assuming a very large program.

5. In general, of the available buffer space, the long reference map file will get 1/16th (if selected) and the size of R-list to COMPS will be a three to one ratio.

5.2.2 I/O Setup and Final Control Card Processing

Next, we open all the compiler files that will be used. Call TIME to get the time and date for the header line. Then scan the flags set during control card cracking to set up the options selected in the page header line. It is at this point that conflicting control card parameters are diagnosed (OPT = 1 or 2 and D). Next, we set up all interface cells used in communicating with COMPASS. Finally, we issue a read on the input file and proceed to load the 1,0 overlay.

LSTPRO

1.0 General

LSTPRO contains the routines which fetch from or enter into the symbol table a given symbol or label.

LSTPRO calls one external routine, ERPRO.

As an instrument for storing data, the symbol table is active during Pass 1 only. The two word symbol table is saved for the FTNXAS assembler during Pass 2, The assembler uses only the finding feature of LSTPRO. The rest of Pass 2 processing accesses the table directly via the ordinals.

Throughout Pass 1, symbol table entries are two words in length. Any necessary information which does not fit in the two word entry will be kept in auxiliary tables elsewhere in memory. The symbol table will begin below the buffers for R-list and COMPS and expand (as new entries are made) into lower addressed consecutive locations, while the auxiliary tables are built from the first available location in low core and expanded into higher addressed locations. These auxiliary tables contain the DIMENSION information as well as the information required to process COMMON and EQUIVALENCE statements.

2.0 Usage

2.1 Entry Point Names: SYMBOL, LABEL

SYMBOL searches for a given 7-character symbol in the symbol table. If the symbol is already in the table, the entry is loaded and SYMBOL returns to the caller. If the SYMBOL is not presently in the table, it is entered in the table, loaded, and SYMBOL returns to the caller.

LABEL searches for a given 6-character statement label in the symbol table in exactly the same manner as SYMBOL searches for symbols.

2.2 Calling Sequence and Returns

Entry is made to SYMBOL or LABEL via a direct jump (not a return jump) with the following register requirements:

X1: The symbol (or label) left justified in bits 0-47 with blank fill. The contents of bits 48-59 are insignificant.

B7: The address to which control is to be returned if the symbol was not already in the table.

B7+1: The address to which control is to be returned if the symbol was already in the table.

Control is returned to the caller with:

B1 = ordinal of word A of the symbol.
B2 = double the ordinal of word A of the entry.
B5 = 1
X1 = word A of the entry.
X2 = word B of the entry.
A0 = starting address of the symbol table.
A1 = address of word A of the entry.
A2 = address of word B of the entry.

In addition for the first occurrence of a name

X6 = natural type in the type field
X7 = 0

For DEBUG mode and first usage of a variable of type T.DBG

X6 = Saved natural type
X7 = DEBUG field bits (non-zero)

3.0 Diagnostics

3.1 One fatal to compilation condition may be detected: "SYMBOL TABLE OVERFLOW" (a maximum of 8192 words is used for the symbol table).

3.2 No fatal to execution errors are detected.

3.3 No information diagnostics are issued.

3.4 No non-ANSI errors are detected.

4.0 Environment

When LSTPRO is entered, it is expected to search for a given symbol or label, enter the symbol or label if it is not presently in the table, and return the entry to the caller. Hence, no conditions are expected to be set up by any other processors (with the exception of the common cells noted in section 7.0 of this document).

During Pass 2, and certain phases of I/O processing, the storing feature of LSTPRO is disabled.

5.0 Structure

5.1 SYMBOL

SYMBOL hashes the 7-character symbol (to be searched for) into a 7-bit pointer. This value is added to the base address of a table (SLIST) to load a word which contains an ordinal of a symbol table entry which is the head of this particular list. Routine SLCOMM is then entered.

5.2 LABEL

LABEL hashes the 6-character statement label (to be searched for) into a 5-bit pointer. This value is added to the base address of a table (LLIST) to load a word which contains an ordinal of a symbol table entry which is the head of this particular list. A jump is then taken to SLCOMM.

5.3 SLCOMM

SLCOMM transfers the symbol (or label) to the specific register (X0). Then, if this particular list is empty, the symbol is entered in the symbol table, its ordinal is set as the head of the list and the not-found exit is taken. If the list is not empty, SLCOMM sets the head of the list ordinal in B4 and jumps to TOP.

5.4 TOP

TOP is the main search loop of SYMBOL. After loading the symbol located at the head of the list, the loop is entered to compare the symbol searched for with each symbol already in the list. The comparison is an integer

subtraction. The list is searched until a match is found or the end of list is found (Px=0).

5.5 ENTER

Control transfers to ENTER when it is determined (at TOP) that the current symbol is a new symbol and consequently must be entered in the table. The new symbol is appended to the end of the symbol chain.

5.6 RETRN

RETRN is the not-found exit. B5 is set to 1, B2 is set to twice the ordinal of the current symbol (B1+B1), the first word of the entry is loaded into X1, and a jump is taken to the address specified in the B7 register.

5.7 FOUND

FOUND is the found exit. Its function is exactly the same as RETRN except that the jump is taken to the address specified in B7+1.

5.8 NTYPE

This routine determines the natural type of a variable according to the implicit type table. On entry, the name is in X1. When it has been typed, X0 holds the type (right adjusted) and X6 contains the type in the type field.

5.9 LDRPH1

A transfer to this entry point will reload pass one of the compiler after resetting FETS and clearing batch control cells. SYMORD (number of symbols), NAALN (next available APLIST number), NDOTEMP (number of DO temporaries), PASS2R (starting value of pass 2 R number), and P2NOGO (GO/NOGO flag for DEBUG mode) are reset. The symbol entering facility that was deactivated for FTNXAS is restored. Finally, we load pass one of the compiler (1,4 overlay if DEBUG mode, 1,1 overlay normally)

5.10 LIST

LIST places a line on the output file, decreases the line count, and forces a new page if necessary. On entry, B1 = word count and B7 = FWA of line.

5.11 OPENF/CLOSEF

These entries can be used to open and close files. X1 holds the FET address and X2 the open/close code. The call is made with auto recall, after waiting for file activity to cease.

5.12 MOVE

This routine will move the number of words specified by X1 from a starting address in X2 to a destination given by X3.

5.13 CONDEC

CONDEC converts the binary integer (less than 2 ** 17) in X1 to display code. On exit, X6 contains (in 10H form) the integer, right adjusted, and B2 holds six times the number of digits converted.

5.14 OUTUSE

On entry, X6 holds the address of a block name. A USE name is issued to COMPS and blocks are switched.

5.15 RSSW

RSSW is called to set switches so that SYMBOL and LABEL do not add an entry to the symbol table if the name is not found.

5.16 KSSW

KSSW is called to restore the switches set by RSSW to their original state.

6.0 Formats

6.1 Symbol Table Formats

```
VFD  42/NAME,1/FP,1/DEF,1/FUN,1/COM,1/DIM,1/EQU,12/P+
```

WORD A

NAME: (BITS 59-18) The name of the symbol or label, 7 (6 if a label) or less display code characters left justified with blank fill.

FP: (BIT 17) Set if the symbol is a formal parameter.

DEF: (BIT 16) Set when the symbol becomes defined.

FUN: (BIT 15) Set if the symbol has been used as a function (external, ASF, or inline).

COM: (BIT 14) Set if the symbol is in common.

DIM: (BIT 13) Set if the symbol is a dimensioned variable.

EQU: (BIT 12) Set if the symbol is a non-base member of an equivalence class.

P+: (BITS 0-11) The ordinal of the symbol table entry, in this list, which is the next greater entry than this entry, or if none exists, P+ = 0.

```
VFD  4/TYP,1/ASF,1/EXT,1/0,12/DIMP,1/VAR,1/0,2/RL,
     18/RA,7/RB,10/0,2/LVL
```

WORD B FOR VARIABLES AND ARRAYS

TYP: (BITS 59-56) Contains the type of the symbol.

0 - logical	1 - integer
2 - real	3 - double
4 - complex	5 - ECS
6 - label	7 - RETURNS parameter
10 - NAMELIST name	11 - unused
12 - entry point	13 - LFN
14 - CGS	15 - unused
16 - unused	17 - unused debug variable

ASF: (BIT 55) Set for arithmetic statement functions.

EXT: (BIT 54) External symbol if set.

DIMP: (BITS 52 - 41) The ordinal of the dimension or equivalence information.

VAR: (BIT 40) Set if a symbol is referenced as a variable.

RL: (BITS 38-37) Holds type of relocation.

0 - absolute	1 - local or program
2 - common	3 - external

RA: (BITS 36-19) The relative address of the symbol.

RB: (BITS 18-12) The relocation base that the symbol is in.

LVL: (BITS 1-0) The level number of the symbol when it is a variable, indicating the SCM/LCM residency of the variable.

```
VFD  4/TYP,1/ASF,1/EXT,1/0,12/DIMP,1/VAR,1/0,2/RL,
     2/0,1/AC,1/SF,1/IF,1/NOT,12/DTO,7/RB,10/0,2/LVL
```

WORD B FOR DEBUG MODE

AC: Selects array bounds checking or CALL tracing.

SF: Selects stores checking or function tracing.

IF: 0 for arrays and stores; 1 for calls and functions.

NOT: Indicates if debugging is selected for this symbol.

DTO: Ordinal into the debug table.

```
VFD  4/TYP,1/ASF,1/EXT,2/0,6/FARG,1/BEF,1/INF,1/0,
     1/NRET,1/NORF,1/VAR,1/0,2/RL,18/RA,7/R8,12/0
```

WORD B FOR FUNCTIONS AND SUBPROGRAMS

FARG: Number of arguments.

BEF: Set for basic external functions.

INF: Set for intrinsic functions.

NRET: Set for functions that do not return.

NORF: Set for functions which do not have side effects.

```
VFD  4/TYP,1/GEN,1/RZ,1/RSN,1/DSN,1/DFN,1/RFN,1/RAS,
     1/DLT,12/DLN,12/TRO,12/L,12/0
```

WORD B FOR PROGRAM LABELS

TYP: Type of the label (will always be 6).

GEN: Zero for program labels; set for compiler generated labels.

RZ: Set if label is referenced prior to current DO nest.

RSN: Set if the symbol is referenced as a statement number (i.e. active label).

DSN: Set if the symbol is defined as a statement number.

DFN: Set if the symbol is defined as a format number.

RFN: Set if the symbol is referenced as a format number.

RAS: Set if the label is referenced in context as a statement number.

DLT: Set if the label is used as a DO loop terminator.

DLN: Line number the label is defined on.

TRO: Label table ordinal (Trace option in debug mode).

L: Ordinal of the loop that the label is referenced in.

```
VFD  4/TYP,1/GEN,1/E,1/X,1/I,1/M,1/V,1/J,1/R,
     12/0,12/TLLN,12/0,12/0
```

WORD B FOR DO GENERATED LABELS

TYP: Type of the label (will always be 6).

GEN: Always set for DO generated labels.

E: Set if loop may be entered at a point other than the top.

X: Set if the loop may be exited at a point other than the terminating statement of the DO.

I: Set if the loop contains another loop.

M: Set if loop control variable must be materialized.

V: Set if control variable is equal to incremental variable (DO 10 K=1,N,K).

J: Set if the loop contains an external reference.

R: Set if all integer variables are considered to be redefined within the loop.

TLLN: The line number associated with the top of the loop.

7.0 When LSTPRO is entered, it is assumed that SYM1 (RA+12B) and SYMEND have been initialized. Thereafter, LSTPRO updates SYMEND each time a new symbol or label is entered in the symbol table.

8.0 In order to find a given symbol in the symbol table (or to determine that the symbol is not yet in the table) with the least number of comparisons, the symbol table is actually broken down into a number of short lists. Each symbol in a list is linked to the other symbols in the list. Each symbol contains a pointer (P+) to the next symbol in this list.

Although each list is linked only within itself, this does not mean that the elements of one particular list must be stored consecutively in memory. As each new symbol is encountered, it is simply stored in the next available location, and pointers are set up to reflect its location in the table.

Each one of the short lists must have a starting point, or head of the list. Further, we must have a way of determining what list a particular symbol belongs to. This is done by commutatively forming (by the use of shifts and the exclusive OR (logical difference operation) a 7-bit value or a 5-bit value for symbols or labels respectively. This value is an index into one of two local tables (SLIST for symbols, LLIST for labels) which contain symbol table ordinals which point to the head of that list. Initially, the SLIST and LLIST tables of list heads are set to zero. If a given cell is loaded that is zero, then we know this is the first symbol in this list and therefore no searching must be done. The symbol is merely entered, set as the head of this list, and a return is made to the caller.

OUTPTK

1.0 General Information

OUTPTK is a combined KODER-OUTPTC facility for use by portions of the compiler coded in FORTRAN. The format processing offered is a subset of that available in KODER.

2.0 Entry Points

2.1 OUTCI.

This entry corresponds to the initial call entry of the FORTRAN object time coded output routine. X1 holds the unit number and A1 points to the I/O parameter list.

2.2 OUTPUT

The file number of the output file.

3.0 Messages And Diagnostics

If an illegal format specification is used, OUTPTK will cause the compiler to mode out by jumping to -1.

4.0 Environment

4.1 External Routines

All output lines are written using LIST if the file is the output file or WRWDS if some other file is specified.

5.0 Processing

5.1 General

On entry at OUTCI., the file number and format location are saved. Initial pointers to the format are established and an entry is made in the parenthesis level stack for the zero level parenthesis.

For subsequent intermediate entries at OUTCR., a certain amount of initialization is performed to setup the format word, the output word, format shift and the number of bits filled in the output word.

At NEXTDESC, we extract the next format character, for a digit DECIMAL is called to compute the repeat count. Then if the format descriptor requires a field width value (Aw, Iw, etc.), we call DECIMAL once more. Then, we transfer to the appropriate processing section via a jump vector. After a specification is processed, control returns to NEXTDESC.

5.2 R Format

The R format data item is shifted to an A format data item and A format processing is used.

5.3 A Format

If the number of characters to be output will fit into the space remaining in the current output word, then the space is cleared and the data inserted. Otherwise, the data is split between the current word and the next word.

5.4 I Format

The constant is converted to display code and positioned to the top of the word, then A format processing is used to insert it. For widths larger than ten, spaces will be filled.

5.5 A Format

For A specification greater than twenty characters wide, the excess is space filled and the width treated as twenty characters. For widths greater than ten, the upper and lower parts of the data word are converted in parallel and added to the line via RFORM and AFORM.

Widths less than or equal to ten characters are converted in a separate loop. Leading zero digits will be replaced by the character replicated ten times in OFORMCON. Thus, to obtain octal output with leading zero suppression, this constant should be changed to blanks.

5.6 H Format

The Hollerith string is issued via AFORM in groups with a maximum size of ten characters.

5.7 Delimited Hollerith String

The initial delimiter (asterisk or quote) is obtained. Then characters are accumulated and issued via AFORM (in groups of ten) until a matching delimiter is found.

5.8 Left Parenthesis

The current restart information (beginning group address and repeat count) is saved in the parenthesis level stack. Then the new repeat counter is established.

5.9 Right Parenthesis

First, we decrement the group repeat count. If it is not exhausted, we reset to the group start and exit to NEXTDESC. If this is the zero level, a new line is forced and the format is restarted at the last left parenthesis encountered. For a non-zero level, we remove a member from the parenthesis level stack and reset to it.

5.10 T Format

For a tab exceeding the present maximum line length, we space fill to the tab position. In the case of a backward tab, the pointers are simply reset to the proper word and bit position.

5.11 X Format

The proper number of spaces are filled by SPACE.

5.12 Slash Processing

The line pointers are reset to the maximum length so far reached. Then the line is padded out until a zero byte line terminator can be appended. Then, the line is written to the unit specified in the initial call. Finally, the registers are reset to start a new line.

5.13 SPACE

This routine will append a specified number of blanks to the line under construction.

PS1CTL$

1.0 General Information

PS1CTL$ is the interface routine between SCANNER and the statement processors for all non-specification statements. The Pass1 table manager routines and the routine to collect references when R=2 or R=3 is selected, are located in PSICTL$.

2.0 Entry Points

2.1 PH2CTL

This is the entry to phase two statement processing. Control is passed to PH2CTL by DPCLOSE.

2.2 IPH2

This routine sets up phase two for executable statement processing. Space is allocated for the ARLIST buffer used by ARITH and the base address of the buffer is substituted into all references to it within ARITH. If debug mode is selected, we set up the pointer block used in the FORTRAN part of Pass 1,4 and call BUGPRO. Then the FORTRAN copies of the pointers are copied back to the appropriate COMPASS versions and BUGACT is called to turn on options if packet information is present.

2.3 PH2RETN

All statement processors return here upon completion of processing.

2.4 LDPS2

Terminate pass one processing and load pass two of the compiler. If in debug mode, call BUGSOUT to scan the AREA list for errors. The R-list file is rewound and the reference map file dumped if R=2 or 3. If there were fatal errors, we will load the 1,3 overlay. Otherwise, the 1,2 (OPT=0 and 1) or the 1,5 (OPT=2) overlay will be loaded.

2.5 ADDREF

ADDREF is called by the statement processors, when the R option is selected, to add a reference for a symbol. The references are collected into lines and the lines are dumped to the REFMAP table for processing at the end of pass two when all symbols have been assigned addresses. Each line consists of a number of fifteen bit parcels terminated by one or more zero parcels to fill out the last word. The first parcel holds the line number (in binary). Succeeding parcels have the form

1/0, 2/REF, DEF code, 12/SYMTAB ordinal

The low order parcel of the last word contains the number of parcels in the last word in the format 3/parcel count, 12/0.

2.6 ALLOC

Adjust table size for table n.

On entry:

A0 = table number

X5 < 0 then the length of the table (L.TBL) and size of the table (S.TBL) will be made equal to -X5.

X5 ≥ 0 then the size of the table will be adjusted such that S.TBL is greater than or equal to X5+ the length of the table.

On exit:

A0 = table number

X7 = non-zero if the space was allocated.

2.7 ADDWD

Adds a word to the end of a managed table.

On entry:

A0 = table number

X1 = word to be added

B5 = 1

On exit:

X6 = word that was added

X7 = new length or zero if no space available

2.8 ALLAE

Allocates almost all available core to table n. On entry, A0 contains the table number.

2.9 INITBL

This entry initializes tables for a phase. On entry, X6 holds the address to be used as the low core address of scratch storage. It is called from PH1CTL, the start of DPCLOSE and the end of DPCLOSE.

2.10 PTU

This routine is called to pack tables to high core before loading pass two. On entry, X1 holds (right adjusted) in successive 6 bit fields the table number plus one of tables to be saved in high core.

2.11 CFMTN

Set non-zero when it is necessary to check for a deleted jump to the next label and the next label is a format.

2.12 LSFLG

Set non-zero if the last statement was an unconditional transfer of control.

2.13 DOFLAG

DO loop nesting depth value.

2.14 CTBLOVL

Issues the diagnostic COMPILER TABLE OVERFLOW.

3.0 Diagnostic Messages

CONFLICTING USE OF LABEL	Fatal to execution
OUT OF SEQUENCE DECLARATIVE STATEMENT	Fatal to execution
NO PATH TO THIS STATEMENT	Informative
COMPILER TABLE OVERFLOW	Fatal to compilation
NO EXECUTABLES IN BLOCK DATA	Fatal to execution
HEADER CARD NOT FIRST STATEMENT	Fatal to execution

4.0 Environment

4.1 LOWCORE CELLS

12B	SYM1	FWA of symbol table
13B	SYMEND	LWA of symbol table
17B	DIMI	30/Length, 30/FWA of dim table
21B	LTYPE	Type of logical IF
23B	CLABEL	Label of current statement
24B	TYPE	Statement type code
32B	SELIST	FWA of E-LIST
34B	LELIST	E-LIST pointer for true side of IF
37B	DUKE	Binary line count
51B	ATYPE	Arithmetic statement type
52B	NGLN	Next generated label
56B	PROGRAM	12/200n,48/0 where n=0 for program n=1 for subroutine n=2 for function
54B	NRLN	Next available number for RI

4.2 COMMON BLOCKS

//	Blank common used by debug processing

/DOLVL/	Do nesting depth level
/STSORD/	Number of statement temporaries generated for a single statement
/MACBUF/	Temporary scratch area
/BUMBLEB/	Cells pertinent to debug processing
/BIGBUGS/	Flags for major debug specification types
/NONFTNX/	SCANNER to FORTRAN communication cells

5.0 Processing

5.1 PH2CTL

Entry to Pass 1, Phase 2

a. Set next generated statement label to 1.

b. Set next available R number to 2. (0,1 reserved for B0, A0 respectively)

5.2 IPH2

Initialize Phase 2 for Executable Statement Processing

a. If the system programmer package option is selected, set the length of the instrinsic function table to include these extra functions.

b. Set FSTEX to DUKE, first executable satement for DEBUG.

c. If processing block data, executables are illegal; issue diagnostic and continue through main loop.

d. If less than 400 words of working storage are available issue Fatal to Compilation error.

e. Allocate space for ARLIST buffer and set up base so that it's address can be substituted when referenced in Pass1.

f. If in non-DEBUG mode, return to caller. If in DEBUG mode:

1. Set up FORTRAN correspondents of COMPASS pointers via POINTRS and set up tables via BUGPRO.

2. Readjust COMPASS pointers to reflect changes made by FORTRAN routines.

3. If there is packet information, activate the options via BUGACT.

4. If variable dimensioned F.P.'s set up dim table for them.

5. Return to caller.

5.3 PH2RETN

Return from statement processors

a. If the statement was labeled, check for termination of DO loop via DOLAB.

b. Form RLIST for end of statement and send to RLIST file.

c. Terminate line of references for R > 0.

d. If not in DEBUG mode continue with main loop, 5.4. If in DEBUG mode:

e. If executables have begun and there is packet information, set up FORTRAN pointers via POINTRS and deactivate options which had been turned on at 5.2.f.3 via BUGACT.

f. If there were comment cards, activate and deactivate options via BUGACT which were supposed to be activated or deactivated at those line numbers. The process is repeated for the update idents for the comment cards.

g. If executable statements have not begun and there is no packet information, check DTYPE to see if the next statement is a debug statement.

h. If it is, call SCANNER to put the statement into E-list.

i. If there is any packet information, set up the FORTRAN pointers via POINTRS and activate options via BUGACT.

j. If SCANNER did not type the statement, call GETTYPE to do so.

k. If it is a bad or non action (DEBUG or AREA) statement, repeat loop starting at 5.3.e.

l. For an action statement set up FORTRAN pointers via POINTRS, set up area and options list locations and convert statement to table form via BUGCON.

m. For statements other than OFF call TURNON to activate the option, and continue through loop beginning at 5.3.e.

n. If the next statement is not debug (5.3.g) call SCANNER to get the statement. If the statement is a bad FORTRAN statement and the next statement is debug, it will be that debug statement that SCANNER returns.

o. So check for a valid debug card, and repeat loop 5.3.e - 5.3.m.

p. If it is a program card and executables have begun, or there is no packet information, go to 5.4.b. The RJ SCANNER can be skipped since there already is a statement in E-LIST ready to be processed.

q. If executables have begun and there is packet information, set up pointers via POINTRS and activate the options via BUGACT; go to 5.4.b.

5.4 PH2SCAN

Main loop for processing executable statements.

a. Call SCANNER to put the next statement into E-list.

b. If the statement is a declarative (type 2-9), issue "out-of-sequence" diagnostic. For a header card, (type 0-1) issue "header card not 1st) diagnostic.

c. For FORMAT:

1. Check for the following condition and issue diagnostic if it holds:

```
        IF (expr) N1,N2,N3
N1      FORMAT (...)
```

   where the IF processor deleted the jump to N1.

2. Call FORMAT to process the FORMAT statement and return to 5.3.c.

d. For any other statement: Check for unreachable statements.

1. If the previous statement was not an unconditional jump or if the statement has a label, it is not unreachable, so go to 5.4.e.

2. If the statement does not terminate the block, issue the "unreachable statement" diagnostic.

e. If there is a label, process it via DOLABCN.

f. Reset number of statement temporaries used to 0 and update STMAX, number of ST.'s needed, to the maximum of previous STMAX and STSORD.

g. Reset "next available R number" to 2 if necessary.

h. If executables have not begun and the current statement is executable, initialize the compiler for Phase 2.

i. Jump to the appropriate statement processor using the statement type and the table VTABL. The DATA and NAMELIST statement processors return to the return routine at 5.3.c. For END statement and end-of-file (END card assumed), return is to LDPS2 (5.5) to load Pass2. All other statements return to the start of the common return routine (5.3).

5.5 LDPS2

a. If in debug mode check for errors in the AREA list.

1. If FWA = LWA, there was no AREA list.

2. Call BUGSOUT to scan the AREA list for errors and issue diagnostic if necessary.

b. Rewind RLIST file.

c. If reference map level 0, terminate current lint of references.

1. Terminate the reference list.

2. Add dimension and common table information to the end of reference map information.

d. If there were fatal errors, load (1,3) overlay to print the errors.

e. If the Q option (quick mode - pass 1 compilation only) is not selected load either (1,2) overlay for OPT = 0,1 or the (1,5) overlay for OPT = 2.

f. If Q is specified and refmap is selected, load (1,2) overlay to process the reference map.

g. If Q is selected and input buffer is empty, terminate compilation. Otherwise reload Phase 1 for compilation of subsequent program units.

STMTP

1.0 General Information

STMTP is the miscellaneous statement processor. NAMELIST, ENTRY, STOP, PAUSE are processed here. In addition, there are three entries for forming macros to place on the COMPS file.

2.0 Entry Points

2.1 NAMELST

Processes the NAMELIST statement using E-list produced in SCANNER. Line images written to the COMPS file are produced.

2.2 ENTRY

Processes the ENTRY statement. Appropriate macros are written to the R-list and COMPS files.

2.3 STOPP

Processes the STOP statement.

2.4 PAUSEP

Processes the PAUSE statement.

2.5 SVARG

Saves an argument to a COMPS file macro under construction. On entry:

B7 = number of words in argument buffer (must be initialized to zero and maintained between calls)

B7 = argument number

X6 = 12/2000B+conversion code, 6/0, 42/arg

Successive calls to SVARG must have ascending argument numbers. Conversion codes are:

0 - arg is a symbol table ordinal

1 - octal conversion (arg is -377777 to 377777)

2 - integer conversion (arg is 0-9)

2.6 F1AMAC

Form and output a one argument macro call to the COMPS file whose argument is a name in the symbol table. On entry:

X1 = macro name

X6 = symbol table ordinal

2.7 FMAC

Format and output the macro whose arguments have been saved via SVARG calls. On entry:

X1 = 10H macro name

NARGS = number of arguments

3.0 Diagnostics And Messages

NAMELIST STATEMENT SYNTAX ERROR

BAD GROUP NAME

GROUP NAME NOT IN SLASHES

CURRENT OBJECT NOT A VARIABLE

PRESENT USE CONFLICTS WITH PREVIOUS APPEARANCE

VARIABLE DIMENSIONS NOT ALLOWED IN NAMELIST

NAMELIST STATEMENT IS NON-ANSI

ENTRY STATEMENT IN A DO LOOP

ENTRY STATEMENT IS NON-ANSI

PREVIOUS USE OF NAME IN ANOTHER CONTEXT

SYNTAX ERROR

ENTRY STATEMENT IN MAIN PROGRAM

LABELED ENTRY STATEMENT

BAD SYNTAX IN STOP OR PAUSE STATEMENT

4.0 Environment

The statement processors expect the statements in E-list starting at the location contained in SELIST. The symbol table will be at SYM1 and the dimension table at DIM1, CLABEL holds the current statement label. NRLN the next available R-list number.

5.0 Processing

5.1 NAMELST

On entry, we switch to the DATA. block. If the first E-list item is not a slash, then bad syntax is diagnosed. If the second item is not a name, then a bad group name is diagnosed. The group name is entered into the symbol table. If it is already in the table, the message "prior usage in another context" will be elicited. The word B bits for a namelist group name will be combined with the address in the DATA. block and word B will be updated. Then, an ADDREF call is made if R=2 or 3.

Next, F1AMAC is called to output the group name macro. If the group name is not followed by a slash, a diagnostic will be produced.

Next comes a loop to process the items in the list of the namelist. A name is extracted and PNV is called to process the namelist variable. Upon return, a check is made for a comma. If one occurs, the previous process is iterated. Otherwise, the group is terminated.

If the last item was a slash, we restart at group name processing. If it was not, an end of statement a syntax error is produced.

5.2 PNV - Process Namelist Variable

A symbol call is made. For first occurrence, the type and VAR bits are set. If the symbol is ordinal one and this is not a function subprogram, an error is produced. An error is produced if the item is type ECS, a function or an external. Next, the DEF bit is set and the DIMP field extracted. A SVARG call is made with the variable name. Next, we prepare the type and SVARG its value. If the item is equivalenced arguments 3 (BASE) and 4 (BIAS) are prepared and saved. For a formal parameter, argument five is saved. An error message is issued at this point if the item is variably dimensioned. Otherwise, D1, D1*D2 and D1*D2*D3 are computed and saved as necessary. Finally, the number of arguments is saved, a reference collected, and the macro formed via FMAC. Then the routine exits.

5.3 ENTRY

If this is a main program, we issue a fatal error for an entry statement. If it is labeled, another fatal error is produced. A basic syntax check for a name followed by an end of statement marker is made. The name is placed into the symbol table and the type set to entry. The ordinal is placed into the entry macro for R-list and in O.CEP (ordinal of current entry point, used by RTNPROC).

If no executable statements have occurred, we make the address of this entry the same as the main and issue an FEQU macro to the COMPS file. Then references are gathered, the symtab ordinal placed in the ENTR table and a non-ANSI error flagged before exiting.

When executable code has occurred, additional processing is required. A check is made for formal parameters or RETURNS. If this routine has formal parameters, we must place FTNNOP. and NOPS. into the symbol table as well as write proper values for them to the COMPS file. Having written the DATA values to COMPS, O.SPEC is cleared to prevent issuing them again for a subsequent ENTRY statement. If we are within a DO loop, a diagnostic is produced. If this is not the case, we join terminal processing for the no executable case having written the R-list entry macro.

5.4 STOP, PAUSE

The object routine name is placed in X1 (STOP. or PAUSE.) and PSP (Process STOP, PAUSE) is called. For STOP statements, the no return bit is set in word B of the symbol table entry for STOP.

5.5 PSP

Enter the name in the symbol table setting the external bit if the symbol is first entered. The ordinal is placed in the R-list buffer. If an EOS occurs next, we use blanks for the message string. If the next item is not a constant, an error is issued. The item after the constant must be an end of statement. If the constant is not integer, it is assumed to be a Hollerith constant. More than five digits or a non-octal digit will also produce errors. After validating the constant, it is converted to H form, placed in the constant table, and the base, bias saved. Finally, the R-list macro is formed and written.

5.6 SVARG

The argument number and the conversion mode plus argument are combined and stored in ABUF.

5.7 F1AMAC

The number of arguments is set to one, the argument saved and FMAC called.

5.8 FMAC

After initializing registers, we compute the difference between the last argument and this argument number. That many commas are added to the string under contruction. Then the argument is unpacked and control passed to the proper argument processor for conversion (symbol table entry, octal constant, integer constant). After conversion, the characters are appended to the string. When NARGS arguments have been handled, we append a zero byte and write the entire line to the COMPS file.

ENDPRO

1.0 General Information

ENDPRO is called when the END card is encountered. All phase two cleanup, diagnostics and terminal processing occur here. In addition, all RETURN statements are processed within ENDPRO.

2.0 Entry Points

2.1 END

This entry is called from PS1CTL when the END card is encountered.

2.2 ECGS

This subroutine enters a compiler generated symbol into the symbol table. Type is set to CGS, RL=1, and RB=CODE. On entry, X1=8R name.

2.3 ENTRY.D

This cell holds the RL, RA and RB of ENTRY. . It is set in PH1CTL.

2.4 OSC

This routine outputs storage for symbols in a table. On entry:

```
X5 = pseudo op word
X6 = FWA of table
X7 = length
```

Table entries are formatted as follows:

```
VFD  6/J, 18/word count, 18/symtab ordinal, 18/J
```

Where J is ignored by OSC.

2.5 BTOCT

Converts a binary numbers to octal. On entry, X1 holds the number. On exit, , X6 and X7 hold the display coded octal constant.

2.6 BEFTB

Base address of the basic external function table.

2.7 L.BEFTB

Zero word at the end of the basic external function table.

2.8 RETURN

Entry in ENDPRO to process the FORTRAN RETURN statement.

3.0 Diagnostics and Messages

END STATEMENT ACTING AS RETURN IS NON-ANSI

FUNCTION NOT DEFINED

RETURN STATEMENT IN MAIN PROGRAM

RETURNS STATEMENT MUST BE IN A SUBROUTINE

ILLEGAL NAME IN RETURNS STATEMENT

RETURNS STATEMENT IS NON-ANSI

4.0 Environment

4.1 Common Blocks

MACBUF - Used by RETURN in constructing the R-list macros for return code.

4.2 Externals

The major externals are listed below with an indication of their use.

DOEND	Located in DOPROC. Called when the END card is found so as to detect unterminated DO nests.
SYMORD	Holds the number of entries in the symbol table
ST.	Holds the ordinal of ST. in the symbol table
CON.	Holds the ordinal of CON. in the symbol table
DATA.	Holds the length of the DATA. block
DATA..	Holds the length of the DATA.. block
O.CBT	Origin of the common block table
N.FP	Holds the number of formal parameters for this routine
DFLAG	Debug mode indicator
MACFLAG	Indicates E or O options selected
RSELECT	Indicates R=2 or 3 selected
ERPRO	Routine to issue fatal errors
ERPROI	Routine to issue informative errors
ASAER	Routine to issue ANSI errors
LWAWORK	Holds the last word address of working storage
WB.ECGS	Word B for compiler generated symbols
WB.PROG	Word B for a program entry

WB.FP	Word B for a formal parameter
WB.FMT	Word B for a format
LSFLG	Set non-zero if the last statement before the END card resulted in a transfer of control (RETURN, GO TO, etc.,)
SAVTBL	Address of list of tables to be saved in high core before loading pass two
PTU	Routine in LSTPRO to pack tables to high core
N.TLAB	Number of entries in the trace label table (non-zero only in DEBUG made)
O.SCR	Origin of the scratch table
O.DIM	Origin of the dimension table
O.SCA	Origin of the saved common address table
O.FPBL	Origin of the formal parameter block length table
O.CON	Origin of the constant table
O.DATA	Origin of the usage defined variable in DATA statement table
O.EXT	Origin of the external table
O.UDV	Origin of the usage defined variable table
	An entry of L.XXX corresponds to each O.XXX above and holds the table length.
LABEL.	Symbol table ordinal of the symbol LABEL.
TEMPAO.	Holds the symbol table ordinal of TEMPAO.
ENTRY.	Holds the symbol table ordinal of ENTRY.
VALUE.	Holds the symbol table ordinal of VALUE.
O.CEP	Holds the symbol table ordinal of the current entry point

WRWDS	Routine in FTN to write to a file
DO.	Holds the ordinal of DO.
OT.	Holds the ordinal of OT.
IT.	Holds the ordinal of IT.
UCODE.	Holds the shifted name of the use block CODE.
OUTUSE	Routine to issue a USE name if needed
SYMBOL	Find or enter a symbol in the symbol table
ADDREF	Routine to collect references when R=2 or 3
CTBLOVL	Control passes to this external if compiler tables overlapped
F1AMAC	Form one argument macros on COMPS
UDATA	Holds the shifted name of the use block DATA.
Z.SCR	Number of the scratch table
ALLOC	Table manager routine to allocate memory

5.0 Processing

5.1 END

On entry, DOEND is called to clean up and diagnose any DO loops still unterminated. Then IAC is called to insert the addresses of common variables into their symbol table entry. (This need be done only in debug mode for common variables with no DIM table entry). Next PSS is called to process special symbols. In particular, this routine will issue the R-list macro to produce a RETURN or RJ END. A check is made here to ensure that the function name has been defined at least once in the function subprogram. Formal parameter block lengths accumulated during namelist processing are moved to word B of the formal parameters.

Now an end of R-list code is written to the R-list file. DCT is called to dump out the constant table. PST is called to process the symbol table. Here, the external

and usage defined variable tables are constructed. All DIMTAB entries are linked to the SYMTAB entry. Special characters are appended to selected externals. Addresses are defined and storage issued for usage defined variables. Move DIMTAB address definition fields into word B of the symbol table. Define the address of usage defined variables which first appeared in DATA statements.

DO., IT., and OT. are entered in the symbol table and the use block switched to CODE. Finally all vital tables for pass two are packed to high core and an exit is taken.

5.2 IAC - Insert Addresses into Common Variables

When the D option is selected, the addresses of common variables without a DIM entry must be saved in a temporary table until the end of pass one when the debug processor is no longer active. If there are no entries in the saved common address (SCA) table, then IAC exits immediately. Otherwise, IAC loops through the SCA placing the RA field in each affected symbol table entry.

5.3 PSS - Process Special Symbols

a. Exit immediately for a BLOCK DATA program.

b. For a main program:

   (1) Place END. in the symbol table.

   (2) Set the external and no return bits.

   (3) Define the program name with an RA of zero in the CODE. block.

   (4) Issue the RJ END. macro to the R-list file.

   (5) Exit PSS.

c. For a subroutine:

   (1) Set the address of the entry into word B using ENTRY.D

   (2) If the last statement was a RETURN, GO TO, etc., go to e.

(3) Issue a RETURN macro to R-list which will restore A0 if needed.

(4) Collect a reference to the current entry point if R=2 or 3.

(5) Issue an informative diagnostic for no RETURN statement.

(6) Go to e.

d. For a function subprogram:

(1) Set the address of the entry into word B using ENTRY.D.

(2) Issue an error if the function name was never defined in the routine.

(3) If the Last statement was a RETURN, GO TO, etc., go to e.

(4) Output R-list for a RETURN statement.

(5) Issue an informative diagnostic for no RETURN statement.

e. Exit if there are no formal parameters.

f. Move the FP block length accumulated during NAMELIST processing to word B (RA field) of the formal parameters.

g. Set up RL, RB fields in each FP's word B for pass two.

h. Exit PSS.

5.4 DCT - Dump Con Table

a. Place the address relative to DATA. block in word B for CON.

b. If there are no constants, go to e.

c. Increment the length of the DATA. block by L.CON.

d. Call ODW to output data words in the CON. table.

e. Exit DCT if no Labels are being traced.

f. Allocate space in which to construct the label table.

g. Increment the length of DATA. by N.TLAB and define the address for LABEL.

h. Scan the symbol table for statement numbers with trace ordinals and make entries in the scratch table of the form VFD 30/NNNNN, 30/line number on which label is defined.

i. Call ODW to dump this table.

j. Exit DCT.

5.5 ODW - Output Data Words

Outputs line images of the form DATA value for each word in the table whose FWA is in X1 and whose length is in X2. A3 holds the first word address of a label for the table the label definition is moved to working storage and the use block switched to DATA. A table entry is picked up and BTOCT used to convert it to octal. The octal is concatenated with a DATA pseudo up and a B appended. This continues until the table is exhausted or working storage fills up and then we dump the images to COMPS. If working storage filled, the remainder of the table is processed after dumping. Then exit ODW.

5.6 PST - Process Symbol Table

(a) Check to make sure there is enough core for END processing.

(b) Initialize registers for the symbol table scan from ST. to the end of the table.

(c) Fetch word A and B, advance to next entry.

(d) If this is the end of the table, go to k.

(e) If this entry is a label, go to c.

(f) For an external, place 2* ordinal into the temporary external table. Go to c.

(g) For types 6-15 and local functions, go to c.

(h) If the entry is dimensional, place the symtab address in word two of the DIM entry and go to c.

(i) If the variable is in common, go to c.

(j) Place the word count and 2* ordinal into the temporary UDV table and go to c.

(k) Define O.UDV and L.UDV as well as O.EXIT and L.EXT.

(l) If no externals, go to p.

(m) Set RL=3 for all externals by stepping through the EXT table.

(n) If the external is a basic external function, a period is appended to the function name.

(o) For the O or E options, issue EXT statements to the COMPS file for each external.

(p) If no dimensional items, go to r.

(q) Move the address information from the dimension table to word B of the symbol table entry. An RL of 1 or 2 is set depending on the common block, or DATA.. is set and the RA is installed in the address definition field of each symbol with a DIM entry.

(r) If there are no usage defined variables, go to v.

(s) Loop through the temporary UDV table incrementing the length of the DATA. block and installing the RL, RA, RB fields.

(t) If O or E options not selected, go to v.

(u) Switch to the DATA. block and call OSC to issue storage to COMPS for usage defined variables.

(v) If no usage defined variables occurred in DATA statements, exit from PST.

(w) Scan the table constructed by DATA installing the RL, RA and RB fields and turning off the common bit in word A of these entries (NOTE: The common bit was turned on so these variables would not appear in the regular UDV table and have storage issued for them. The storage will have already been issued in DATA processing.) At this point, their names will be added to the previous UDV table.

(x) Exit PST.

5.7 RETURN

RETURN processes the FORTRAN RETURN statement. For a main program, an informative diagnostic is produced when a RETURN occurs. If the statement is a normal RETURN, processing differs from that for a RETURNS type statement.

5.7.1 RETURNS processing

(a) Issue an error if this is not a subroutine.

(b) Issue an error if the E-list item is not a name.

(c) Call SYMBOL and produce an error if the name is not found.

(d) Issue an error if the type is not RETURNS.

(e) Generate a non-standard return R-list macro.

(f) Collect a reference (if necessary) for the RETURNS name.

(g) Flag a non-ANSI usage and exit from RETURN.

5.7.2 RETURN processing

5.7.2.1 For a function subprogram:

(a) Generate a single or double precision function return macro on the R-list file.

(b) Collect a reference to the current entry point.

(c) Exit from RETURN.

5.7.2.2 For a subroutine:

(a) Select a return macro to restore A0 or not depending on the presence of a parameter list (TEMPAO. ≠ 0 if restore needed) and generate it on the R-list file.

(b) Collect a reference to the current entry point (if necessary).

(c) Exit from RETURN.

6.0 Table Formats

6.1 EXT Table

```
VFD  60/2*ordinal
```

6.2 UDV Table

```
VFD  6/0, 18/word count, 18/symbol table ordinal, 18/0
```

SCANNER

1.0 General Information

1.1 Task Overview

SCANNER reads each source statement from the input file, determines the statement type from the initial alphabetic keyword (if present), transforms the statement into the E-list intermediate language, lists the statement in the output file and issues suitable diagnostics if errors are found during the lexical scan.

1.2 Significant Changes from Version 3.0

The following major changes have been made to SCANNER since the orignal release of FORTRAN Extended Version 3.0:

1.2.1 IMPLICIT and LEVEL statement processing logic has been added.

1.2.2 SEGMENT, SEGZERO and SECTION statement processing logic has been deleted.

1.2.3 Quote (64B) delimited Hollerith constants have been added.

1.2.4 END statement formats have been relaxed. An END line may now be continued or may follow a dollar sign (53B) separator.

1.2.5 Internally, many subroutines have been recoded to reduce compilation time and shorten field length requirements. For a typical program "mix", Version 4.0 SCANNER runs approximately twice as fast as Version 3.0, and requires about 1000B fewer words of central memory, excluding new feature additions. Much of the speed improvement was achieved by squeezing out source statement blanks as each statement is burst to the string buffer SBUFF.

2.0 Entry Points

2.1 Executable Code

2.1.1 SCANNER

This entry point is used by all callers except for certain special DEBUG statement processing tasks.

2.1.2 DBGERR

This entry point is used to print out a DEBUG error message.

2.1.3 GETTYPE

This entry point is used by DEBUG routines to obtain the type of a DEBUG statement.

2.2 Communications Cells (in alphabetical order)

2.2.1 CD

This cell holds the source line number (binary) of the beginning line of the last FORMAT statement encountered.

2.2.2 COL

This cell holds the number of blanks between the initial left parenthesis and the first non-blank character after the left parenthesis in the last FORMAT statement encountered.

2.2.3 DUKE1

This cell holds the source line number (binary) of the line currently in the card input area CP.CARD.

2.2.4 FEFLAG

This cell is set non-zero externally when a fatal-to-execution error is found.

2.2.5 N.EQUAL

This cell holds the number of equals (54B) signs found in a statement.

2.2.6 O.LCC

This cell contains the first word address of loader control card information.

2.2.7 TYPFLAG

This cell is set less than zero when a DEBUG statement cannot be typed; its value is otherwise zero.

2.2.8 WORDY

This cell contains the total number of words of loader card information.

3.0 Diagnostics/Error Messages

3.1 Fatal to Compilation Error Messages

TABLES OVERLAP, INCREASE FL

3.2 Fatal to Execution Error Messages

UNRECOGNIZED STATEMENT

ILLEGAL LABEL FIELD IN THIS STATEMENT

STATEMENT TOO LONG

SYMBOLIC NAME HAS TOO MANY CHARACTERS

UNMATCHED PARENTHESES

TABLE OVERFLOW, INCREASE FL

ILLEGAL CHARACTER. THE REMAINDER OF THIS STATEMENT WILL NOT BE COMPILED.

ILLEGAL VARIABLE NAME FIELD IN ASSIGN OR ASSIGNED GOTO

NO TERMINATING RIGHT PARENTHESIS IN LOADER DIRECTIVE

NOT ENOUGH ROOM IN WORKING STORAGE TO HOLD ALL OVERLAY CONTROL CARD INFORMATION

CONSTANT TABLE CONSTORS OVERFLOWED - STATEMENT TRUNCATED. ENLARGE TABLE OR SIMPLIFY STATEMENT

THE STATEMENT IN A LOGICAL IF MAY BE ANY EXECUTABLE STATEMENT OTHER THAN A DO OR ANOTHER LOGICAL IF

( ) WAS LAST CHARACTER SEEN AFTER TROUBLE, REMAINDER OF STATEMENT IGNORED

DEFECTIVE HOLLERITH CONSTANT. CHECK FOR CHARACTER COUNT ERROR, MISSING ≠ DELIMITER OR LOST CONTIN CARD

3.3 Informative Diagnostics

NO END CARD, END LINE ASSUMED

UNRECOGNIZED STATEMENT

3.4 Non-ANSI Diagnostics

7 CHARACTER SYMBOLIC NAME IS NON-ANSI

LOGICAL OPERATOR OR CONSTANT USAGE IS NON-ANSI

OCTAL CONSTANT OR R,L FORMS OF HOLLERITH CONSTANT IS NON-ANSI

DOLLAR SIGN STATEMENT SEPARATOR IS NON-ANSI USAGE

THE FORMAT OF THIS END LINE DOES NOT CONFORM TO ANSI SPECIFICATIONS

4.0 Environment

4.1 Common Blocks

DBGBLK1, DBGBLK2 and NONFTNX - Each block contains a series of communications cells for the DEBUG option.

4.2 Externals

The major externals are listed below, with an indication of their use.

ASAER — Entry point in ERPRO to file non-ANSI diagnostic messages.

CAFLAG — Control card option flag to specify COMPASS assembly.

CIO1. — Entry point in FTN to issue CIO requests.

CONDEC — Entry point to convert a binary value to display code.

CP.CARD FWA of source line image input working storage area.

CP.LINE FWA of source line outpou list working storage area.

DFLAG DEBUG mode option flag.

ERPRO Entry point in ERPRO to file fatal-to-execution diagnostic messages.

ERPROI Entry point if ERPRO to file informative diagnostic messages.

FATALER Entry point in ERPRO to file fatal-to-compilation diagnostic messages.

FTNEND Entry point in FTN to end compilation task.

FWAWORK End of E-list area; E-list grows down from high to low addresses.

LCNT Number of lines remaining on current page of output listing.

LDCOM Entry point in FTN to load and execute the COMPASS assembler.

LIST Entry point in LSTPRO to write a line to the output file.

LWAWORK Cell containing address of last E-list entry; decremented after each entry.

MOVE Entry point to move a block of words in central memory.

NOLSFLG Control card option flag to suppress output listing.

PAGE Current output listing page number.

PUTUPDT Entry point to save an UPDATE identifier (from a source line) for DEBUG.

SAVLINE FWA of block where a complete source statement (20 lines max) can be saved. Block is used only when normal output listing is being

suppressed, so that a statement can be listed if errors are found in it.

SAVLNG — Number of initial alphanumeric characters in a source statement. Used for processing IMPLICIT statements.

TITLE1 — Word in page header line that contains the type of program unit (PROGRAM, SUBOURTINE, FUNCTION, BLOCK DATA).

UFLAG — Control card option flag to generate *DECK card.

WRWDS — Entry point in FTN to write one or more words to a file.

5.0 Processing

5.1 Processing Function

SCANNER performs the initial lexical scan of each source statement to determine the statement type and to condense the statement into an elemantal form, termed E-list, that can be rapidly processed by external statement processing routines. The statement type information will be used by the calling phase controller (PS1CTL or PH2CTL) to determine which statement processor to call. That processor, in turn, will use the E-list as input data for R-list generation.

To produce the E-list, SCANNER extracts the meaningful symbols from each source statement and reformats them into a series of one-word list entries plus, in some cases, auxiliary table entries. Blanks, comments and other irrelevant data are discarded. Each entry in E-list represents one syntactically significant element in the original source statement, such as a variable name, a constant, an operator or other quantity. The original source symbols may or may not appear explicitly in the E-list entry. A variable name, for example, is retained in the entry. A constant, in contrast, is not, because it may be too long to fit in one word. Accordingly, constants are stored in an auxiliary table, CONSTOR; the E-list entry for the constant provides its length and location. Every E-list entry includes a type code number that identifies the syntactic element type. This code

number is used by the statement processors to expedite compilation. See paragrpah 6.1 of this section for detailed E-list formats.

5.2 General Processing Flow

SCANNER processes one complete source statement in response to each call (RJ SCANNER). This may include up to 20 active source line images, plus an essentially unlimited number of embedded comment cards or blank cards trailing. Upon return to the caller (and assuming no errors were found), the statement will be typed, listed, E-list produced, and the statement number saved. The first source line of the next statement wil have been read and partially scanned, basically to determine if it is a continuation of the current statement, to obtain its statement number (useful for certain code optimization) and to see if it is a DEBUG statement. Upon initial entry after loading, SCANNER enters a initialization mode that will continue until the first recognizable FORTRAN source statement is encountered and typed. In this mode, DEBUG external packet lines are processed; abnormal cards are diagnosed; embedded COMPASS subprograms are either copied to the COMPS file or assembled directly by COMPASS, depending on control card options selected; loader control cards are reformatted and stored at the end of E-list. When a source line is encountered that does not fit the above categories, SCANNER types it to determine if it is a valid program unit header line, such as PROGRAM or SUBROUTINE. If so, the program unit type and name are extracted and stored in a skeleton title header line that will appear at the top of each page of the output listing. SCANNER then abandons the initialization mode and transfers into the middle of the normal or main mode of operation. Any remaining information in the initial source line, such as program file names, subroutine parameter lists, etc., will be posted to E-list at this time, in the normal fashion. If the initial line was not a header line, SCANNER bypasses the header line initialization and continues directly in the normal mode. In this case, default header values will be stored later by PS1CTL. When all non-blank characters of the first source statement have been processed, the first line of the next source statement will have been read and burst to the string buffer; this was necessary to determine if it was a continuation of the initial statement.

At his time, SCANNER has prepared the following information for the caller:

Register B7 and TYPE(RA+24B) will hold the primary statement type. ATYPE (RA+51B) will hold any constant type associated with the statement. If the statement is a logical IF, the type of the statement following the logical expression is found in LTYPE (RA+21) and the starting address in LELIST (RA+34B). The starting address of E-list for the primary statement is found in SELIST. The last location used for E-list is found in ELAST(RA+14B). CLABEL(RA+23B) holds in display code the statement label, if any, left justified and blank filled. NLABEL (RA+60B) holds the label, if any, in thsame form for the next statement. If no label is present , NLABEL and CLABEL will be zero. On subsequent calls, SCANNER processes a complete statement each time entered. Since the first line of the new current statement has already been burst to the string buffer, beginning at SBUFF, it is not necessary to initiate an immediate read. Instead, SCANNER performs a brief internal initialization at SCANNER1 (and SCN2 if in DEBUG mode), a secondary initialization at CONT, and obtains the first non-blank character of the new statement at BOSS, via GET. STATE1 verifies that the statement begins with an alphabetic character and, for a non-DEBUG card, calls PACK30 to pack the initial alphanumeric string that begins the statement. At this point, the processing activities become quite variable, depending on the type of statement. PCK 30 packs the string as GOTO25, and senses that the next non-blank character is not a legal FORTRAN character (01B-57B), since it is in reality the end-of-line (EOL) sentinel. PACK30 calls PGCOM who, after recognizing that it is an EOL sentinel, in turn calls NEXT to obtain a new source line image. NEXT either saves (list option off) or lists (list option on) the current source line and calls READCARD for a new line. READCARD obtains the line via READL and proceeds with a series of housekeeping tasks. If the new line is a comment line, it is listed and another line fetched. If not a comment line, it is checked to see if it is a DEBUG or continuation line. Suitable flags are set, the statement label is processed, and the line is burst to the string buffer beginning at SBUFF. Then, after updating the source listing line number, READCARD returns to NEXT. Assume that the next line was, in reality, a new source statement. NEXT exits to NEWS, who will wrap up the processing of the current statement. NEWS checks

for a series of pack-in-progress events, and finds that PACK30 was interrupted by the EOL sentinel. NEWS terminates the interrupted pack and stores the packed character string in E-list. (NOTE: This particular packed character string will subsequently be logically erased from E-list, since it contains the statement keyword GOTO. E-list is merely used as a temporary convenient storage location; it is not, in this case, receiving a final entry.) After some additional legality checks, NEWS determines that the statement is untyped. NEWS then exits to the type determining routine associated with the next processing state that would have gained control if the current statement had not ended. In this case, the next state would have been STATE2, as declared by STATE1 before calling PACK30. NEWS then exits to the STATE2 type determiner, D1. D1, after some manipulation, calls SEARCH, who types the statement and exits to ADJ. ADJ separates the keyword GOTO from the label 25 that follows, enters the label in CONSTOR, and makes an E-list entry indicating the label location. ADJ exits to STATE0, who posts an end-of-statement entry to E-list and returns to the caller.

This is necessarily a brief description of the processing for a simple statement. Had the statement been more complex, additional STATE processors would have come into play. The STATE processors are, in general, a series of jump tables that indicate the suitable action to take, based on the last non-alphanumeric character seen, plus a series of small action routines to make various E-list entries. Implicit in the structuring of the STATE processors is the expected statement syntax; when a source statement contains characters that violate that syntax, suitable error messages are issued.

5.3 Structure

The open and closed subroutines that comprise SCANNER are listed below in the order of appearance.

5.3.1 ADWORD

Called by the ELPUT macro to enter a word in E-list.

5.3.2 ADD1

Called by the STATE processors to fetch an E-list entry, make the entry, obtain the next non-blank character of the source line, and exit to a STATE processor.

5.3.3 SCANNER1

Perform common initialization chores for all entries to SCANNER.

5.3.4 SCAN2

Performs initialization for DEBUG statements .

5.3.5 SCAN3 thru SCAN6

Performs initialization only for the first entry to SCANNER.

5.3.6 SCAN7 thru SCAN9

Read and process the first source line image. If the first line is blank or abnormal, processing continues until a valid line is found.

5.3.7 AFC1 thru LFC

Diagnoses and lists abnormal first card(s).

5.3.8 IRB

Checks for existence of a COMPASS subprogram; tests for IDENT line.

5.3.9 CP0 thru CP3

Writes *DECK (program name) line to the COMPS file.

5.3.10 CPA, CPB

Copies input line images to the COMPS file until an END line is found.

5.3.11 P

Initializes line counts; obtains (via PACK 30) the keyword string from the first line image; initializes for loader control card search.

5.3.12 PICOVER

Tests keyword string for loader control card; calls PICHIT if found.

5.3.13 PA thru NOTT

Initiates typing of first FORTRAN source line; if a program unit header card is found, enters the unit type and name in the skeleton title for output listings.

5.3.14 CONT

Performs secondary initialization tasks for all normal entries (non-initial) to SCANNER. Updates line numbers; clears error flags.

5.3.15 BOSS

Common transfer point among STATE processors. Fetches next non-blank character of source line via GET; Exits to next STATE via register B1.

5.3.16 STATE1

Insures that a new statement begins with an alphabetic character. If the statement is a DEBUG statement, calls ISITDBG to type the statement. For non-DEBUG statements, calls PACK30 to pack the initial keyword string and continues to STATE2.

5.3.17 STATE2

STATE2 is a jump vector which transfers control to the proper routine depending on the condition that terminated the statement identifier packing. The characters + - *) . cause an unrecognized statement diagnostic.

5.3.17.1 S. A slash terminates the string. SEARCH is called to check for DATA N/, COMMON N/, or NAMELIST/. After successful typing and adjusting, control is returned to STATE3.

5.3.17.2 L1. A left parenthesis terminates the string. If the string is FORMAT and the statement was labeled, control is transferred to FORMAT to process the statement. If not, a parenthesis count is started and control is transferred to STATE3.

5.3.17.3 D1. The typing routine for an alphanumeric statement. SEARCH is called to look for any form of: CONTINUE, STOP, ECS, GOTO, PAUSE, CALL, READ, REAL, ENTRY, PRINT, PUNCH, RETURN, COMMON, DOUBLE, REWIND, COMPLEX, ENDFILE, INTEGER, LOGICAL, PROGRAM, TYPEECS, EXTERNAL, TYPEREAL, BLOCKDATA, BACKSPACE, SUBROUTINE, TYPEDOUBLE, TYPECOMPLEX, TYPEINTEGER, TYPELOGICAL, ASSIGN, DOUBLE PRECISION, TYPEDOUBLEPRECISION. After successful typing and adjusting control is transferred to STATE0.

5.3.17.4 E1. An = sign terminates the string. If the string is from 8 to 14 characters long, a check is made for a DO statement. If so, control is passed to ADJDO. If the string is less than 8, control is passed to STATE6.

5.3.17.5 CC1. A comma terminates the string. SEARCH is called to look for the forms of: ECS, GOTO, CALL, REAL, DATA, READ, PRINT, PUNCH, COMMON, DOUBLE, COMPLEX, INTEGER, LOGICAL, TYPEECS, EXTERNAL, TYPEREAL, SUBROUTINE, TYPEDOUBLE, TYPECOMPLEX, TYPEINTEGER, TYPELOGICAL, DOUBLEPRECISION, TYPEDOUBLEPRECISION. After typing and adjusting control is returned to STATE8.

5.3.18 STATE 3

STATE 3 transforms symbolic names, constants, operators, and delimiters into E-list until the parenthesis count is zero, then control is passed to STATE5.

5.3.19 STATE 5

STATE5 contains a jump vector to pass control to the processing routine depending on the character that appears immediately after the parenthesis count goes to zero. The characters + - *) blank and . will cause an unrecognized statement diagnostic.

5.3.19.1 P5. Is entered when an alphabetic follows when paren count goes to 0. If the string length before the first left paren is 2, a check is made for a logical IF. If so, control is passed to STATE8. If not, SEARCH is called to look for any of the forms of: GOTO, READ, WRITE, ENCODE, DECODE. After successful typing and adjusting, control is passed to STATE8.

5.3.19.2 N3. Is entered when a digit follows as paren count goes to 0. Check the string before the first left parenthesis

for IF and if so, assume an arithmetic IF, then pass control to STATE8.

5.3.19.3 SD. A slash causes SEARCH to be called to look for any of the forms of DATA and COMMON. After typing and adjusting, control is passed to STATE8.

5.3.19.4 L3. A left parenthesis causes SEARCH to look for any proper form of READ, WRITE, ENCODE, DECODE, BUFFERIN, BUFFEROUT and after typing and adjusting, pass control to STATE8.

5.3.19.5 D3. The statement is terminated at parenthesis count = 0. After checking for WRITE and EQUIVALENCE, SEARCH is called to look for any form of: READ, DATA, ECS, CALL, REAL, COMMON, DOUBLE, COMPLEX, INTEGER, LOGICAL, PROGRAM, TYPEECS, TYPEREAL, FUNCTION, DIMENSION, SUBROUTINE, TYPEDOUBLE, TYPECOMPLEX, TYPEINTEGER, TYPELOGICAL, REAL FUNCTION, DOUBLEFUNCTION, COMPLEXFUNCTION, INTEGERFUNCTION, LOGICALFUNCTION, DOUBLEPRECISION, TYPEDOUBLEPRECISION, DOUBLEPRECISIONFUNCTION, and after typing and adjusting, pass control to STATE0.

5.3.19.6 E3. The = sign here causes the type to be set replacement and control is passed to STATE8 after making the string before the first left paren into a symbolic name entry.

5.3.19.7 CC3. The comma causes a check made for EQUIVALENCE and then SEARCH is called to look for the forms of: DATA, GOTO, ECS, CALL, REAL, DOUBLE, COMMON, COMPLEX, INTEGER, LOGICAL, TYPEECS, TYPEREAL, DIMENSION, SUBROUTINE, TYPEDOUBLE, TYPECOMPLEX, TYPEINTEGER, TYPELOGICAL, DOUBLEPRECISION, TYPEDOUBLEPRECISION and after typing and adjusting, pass control to STATE8.

5.3.20 STATE6

STATE6 determines if the statement is a replacement or a DO. A jump vector passes control depending upon the first character after the = sign. A / *) = $, cause an unrecognized statement diagnostic to be issued.

5.3.20.1 P6. An alphabetic causes PACK to be called to pack a symbolic name then pass control to STATE7.

5.3.20.2 N4. A digit causes DIGIT to be called to make the E-list entry for a constant and then pass control to STATE10.

5.3.20.3 A + - (or . cause the statement to be typed replacement and control passed to STATE8 after making a symbolic name entry of the string before the = sign.

5.3.21 STATE7

STATE 7 has a jump vector and passes control depending upon the character that terminated the symbolic name after the = sign. A + - * (= or . cause the statement to be typed replacement and control passed to STATE8 after making a symbolic name entry of the string before the = sign. A) will cause an unrecognized statement diagnostic to be issued.

5.3.21.1 CC4. A , will cause a check of the string before the = sign for a DO. If the first two characters are DO a jump is made to ADJDO to make a constant and symbolic name entry and then pass control to STATE8.

5.3.22 STATE8

The remaining elements of the statement are transformed into E-list and stored until the statement is terminated either by an error occurring, or a new statement being sensed.

5.3.23 STATE10

STATE10 contains a jump vector and passes control depending upon the character that terminated the constant that appeared after the first = sign.

A + - * / (or . will cause the statement to be typed replacement and control passed to STATE8.

A) will cause an unrecognized statement to be issued.

5.3.23.1 CC5.A , will cause the string on the left of the = sign to be checked for a DO statement by calling ADJDO.

5.3.24 STATE0

STATE0 inserts the end-of-statement terminator in E-list. If the statement was a logical IF, followed by any statement except a DO or another logical IF, E-list pointers are rearranged to their correct value. STATE0 exits to the caller via the entry point SCANNER.

5.3.25 D$PROC

Called when a statement is terminated by a dollar sign. Transfers NLABEL to CLABEL, and sets NLABEL to zero. Posts an informative non-ANSI diagnostic message.

5.3.26 DBGERR

Writes an error message to the output listing file, interspersed with the source line image listing. Called by the DBGERR macro; used only to report errors in a DEBUG statement.

5.3.27 DBGTLE

Enters DEBUG PACKET in the output listing title line skeleton.

5.3.28 D.IDSAVE

Extracts the UPDATE identifier field from a source line and copies it to the location specified by the caller. Used for saving line identifiers for the DEBUG option.

5.3.29 D.IDSPEC

Special extension of D.IDSAVE (see above), required for comment lines and all-blank lines.

5.3.30 ISITDBG

Packs, via PACK7, the initial alphanumeric string that begins a DEBUG statement and searches a table of DEBUG keywords, in an attempt to type the statement. Contains an entry point for external callers to request the same service.

5.3.3 PACK7

Packs up to 7 consecutive alphanumeric characters from a source line, adds the E-list type code for a variable name, blank fills the low-order bits, and posts the string to E-list. Packing is terminated when a character outside the range 01B-44B is encountered. If the terminating character is a legal FORTRAN character, 45B-57B, return is to the caller. If the character is outside this range, PACK7 calls PGCOM to make further checks.

5.3.32 PACK30

Packs up to 30 consecutive alphanumeric characters from a source line and posts the results to E-list in successively lower (descending) locations. The packed string is left-justified to bit 59 of the first word, and is left with zero fill. No E-list type code is added, because E-list is merely being used as an interim repository for the packed string. The string will later be logically removed (a responsibility of the caller, directly or indirectly). Normally used to pack the keyword string that begins each source statement. String terminating conditions are the same as those for PACK7, above.

5.3.33 GET

Fetches the next non-blank character from a source line. Returns to the caller with the character in B2, provided it is in the range 01B-57B. If outside this range, calls PGCOM to make further legality tests.

5.3.34 PGCOM

Common routine, shared by PACK7, PACK30, GET, and ADD1 to handle the special cases when a source line character is outside the normal range 01B-57B. If the character is actually an end-of-line sentinel in the string buffer, PGCOM saves registers via SCNSAVE and calls NEXT to obtain the next source line. If the line is a

continuation line, NEXT returns to PGCOM, who in turn returns to the caller after restoring registers via RESTO. If the string terminating character is a quote mark (64B), this marks the beginning of a quote-delimited Hollerith constant. PGCOM changes the character to a 55B to save space in the STATE processing jump tables and returns to the caller. If the string terminating character is any other value, PGCOM exits to a fatal-to-execution error routine.

5.3.35 LCARD

Lists a complete source statement (up to 20 lines) when an error is discovered in the statement and the output listing is suppressed. The saved statement is located in a special area beginning at location SAVLINE.

5.3.36 POINT

When a period (57B) is encountered, POINT is called to process the following character string as a logical, relational or Boolean operator. If the string can be successfully verified as such an operator, a suitable E-list entry is made for the operator.

5.3.37 PACKC

Assembles a numeric string for CONSTOR entry. One character is passed to PACKC per call; when ten characters have been accumulated, PACKC enters the accumulated string in CONSTOR.

5.3.38 PACKT

Completes the CONSTOR and E-list entries for a constant.

5.3.39 DIGIT

Determines the constant type of a numeric string, based on the appearance of a decimal point or the letters B, D, E, H, L, O, or R. Calls PACKC and PACKT to assemble the constant and make suitable CONSTOR and E-list entries. Calls HOLLRTH to process Hollerith constants.

5.3.40 ENDP

Called to perform a series of special-case tasks associated with END line processing. ENDP is entered at

ENDP if an end-of-section/partition/information status is encountered on the input file, and is entered at END3 if the keyword END is followed by other than EOS/P/I. ENDP verifies that a valid END line has been found. If so, the statement is typed as a normal END line. If not, an invented END line is forced to the source input file, so that subsequent processing tasks can follow a semi-normal course of action. If ENDP is entered before any valid FORTRAN source statements have been located, and no illegal source statements have been found, ENDP will immediately terminate the compilation with a suitable dayfile message.

5.3.41 HOLLRTH

Assembles either a standard or a delimited Hollerith constant string, making suitable E-list and CONSTOR entries.

5.3.42 FORMAT

Beginning with the first left parenthesis that follows the characters FORMAT, packs the remainder of the statement, ten characters per word, and stores it in E-list. The last word is filled with blanks.

5.3.43 READCARD

After calling READL to read the next source line from the input file, READCARD checks for comment, DEBUG and continuation lines. Comment lines are listed directly without bursting, so speed processing. DEBUG and continuation lines are flagged for later processing. If a statement label exists, it is leftjustified in a packing register, blank filled, and stored in location NLABEL; the previous contents of NLABEL were moved to CLABEL. Columns 7 thru 72 of the line are burst to the string buffer beginning at location SBUFF. Only non-blank characters are actually stored. When blanks are encountered, their count is simply accumulated until a non-blank character or the end of line is encountered. Then the accumulated blank count, plus a bias of 1, is packed into the exponent field of the non-blank character, and the result is stored in SBUFF. The end-of-line sentinel has an arbitrary value of -1. After bursting, the source listing line number for the line is updated, and READCARD then returns to the caller.

5.3.44 LISTCARD

Lists a source line image from the input area beginning at C.LINE+2 (formerly LINEOUT). Calls external routine LIST to perform the actual I/O task.

5.3.45 READL

Reads a source line from the input file.

5.3.46 PICHIT

Reformats a loader control card and enters it at the end of E-list.

5.3.47 CKCSTOR

Called to verify that CONSTOR storage limits have not been exceeded.

6.0 FORMATS

6.1 E-list format

Element	E-list Format
constant	VFD 12/2000B, 3/t, 6/s, 11/0, 10/n, 18/Pointer
symbolic name	VFD 12/2001B, 48/Name
)	VFD 12/2002B, 48/0
,	VFD 12/2003B, 48/0
end-of-statement	VFD 12/2004B, 48/0
=	VFD 12/2005B, 48/0
(	VFD 12/2006B, 48/0
.OR.	VFD 12/2007B, 48/2
.AND.	VFD 12/2010B, 48/3
.NOT.	VFD 12/2011B, 48/4
.LE.	VFD 12/2012B, 48/5

```
.LT.                    VFD 12/2013B, 48/5
.GE.                    VFD 12/2014B, 48/5
.GT.                    VFD 12/2015B, 48/5
.NE.                    VFD 12/2016B, 48/5
.EQ.                    VFD 12/2017B, 48/5
-                       VFD 12/2020B, 48/6
+                       VFD 12/2021B, 48/6
*                       VFD 12/2022B, 48/7
/                       VFD 12/2023B, 48/8
**                      VFD 12/2024B, 48/10
```

For a constant entry, t = 0 for logical, 1 for integer, 2 for real, 3 for double precision, 5 for octal, and 6 for Hollerith. When T = 6, s = 0 for the H form, s = 1 for the L form and s = 2 for the R form.

n is the number of characters in the constant string and Pointer is the starting address of the string in CONSTOR. For logical constants, the Pointer field will hold -1 for TRUE and 0 for FALSE and the n field is 0 and no CONSTOR entry is necessary.

6.2 Statement Type Codes

Each statement has an associated type code which has the following significance; it is the ordinal in a jump vector of the statement processing program. The elements that actually appear in E-list are underlined.

Statement Code Number	Statement and E-list Entries
0	PROGRAM <u>s</u>
	PROGRAM <u>s (...)</u>
	BLOCK DATA

BLOCK DATA s

SUBROUTINE s

SUBROUTINE s (a1, a2, ..., an)

SUBROUTINE s, RETURNS (b1, b2, ..., bn)

SUBROUTINE s, (a1, a2, ..., an)
RETURNS (b, b, ...,bm)

1 t FUNCTION s (a1, a2, ..., an)

2 LEVEL n ,(v1, v2, ..., vn)

3 IMPLICIT t1(a1), t2(a2), ..., tn(an),
where (an) is of the form (b),
(b1-b2) or any combination
(b1-b2, b3 ...)

4 COMMON /x1/a1/.../xn/an

5 DIMENSION v1, v2, ..., vn

6 EXTERNAL v1, v2, ..., vn

7 EQUIVALENCE (k1), (k2), ..., (kn)

8 INTEGER, TYPE INTEGER, REAL, TYPE REAL,
COMPLEX TYPE COMPLEX, DOUBLE, TYPE
DOUBLE, DOUBLE PRECISION, TYPE DOUBLE
PRECISION, LOGICAL, TYPE LOGICAL, ECS
or TYPE ECS

v1, v2, ..., vn

9 FORMAT (...)

10 DATA k1/d1/, ..., dn/dn/
or
(r1 = d1), ..., (rn = dn)

11 NAMELIST /y1/a1/.../yn/an

12 f (a1, a2, ..., an) = e or v=e

13 END

14	ASSIGN k TO i
15	GO TO k
	GO TO i, (k1, k2, ..., kn)
	GO TO (k1,k2,...,kn)i or e
16	IF (e) ki, k2, k3
17	IF (e) S
18	not used
19	CALL s
	CALL s (a1, a2, ..., an)
	CALL s, RETURNS (b1, b2, ...,bm)
	CALL s (a1, a2, ..., an), RETURNS (b1, b2, ..., bm)
20	RETURN
	RETURN i
21	CONTINUE
22	STOP
	STOP n
23	PAUSE
	PAUSE n
24	DO n i = m1, m2, m3
25	READ f,k
	READ (u) k
	READ (u,f) k
	READ (u,f)
26	WRITE (u) k

	WRITE (u, f) k
27	BUFFER IN (u, k) (A, B)
28	BUFFER OUT (u, k) (A, B)
29	ENCODE (n, f, A) k
30	DECODE (n, f, A) k
31	REWIND u
32	BACKSPACE u
33	ENDFILE u
34	PRINT f, k
35	PUNCH f, k
36	ENTRY s
37	END card assumed for end-of-record

Statement types 0, 1, 3, and 8 require additional type code information. This added information will be stored by SEARCH in ATYPE (RA+51B).

For statement type 0, the additional type code may assume the following values:

0	BLOCK DATA
1	SUBROUTINE
2	PROGRAM

For statement types 1, 3, and 8, the additional type code (termed the arithmetic type) may assume the following values:

0	LOGICAL
1	INTEGER
2	REAL
3	DOUBLE PRECISION
4	COMPLEX
5	ECS (illegal for the IMPLICIT statement)

Note that only the first IMPLICIT statement type is coded in ATYPE. Additional types are stored in E-list, using the standard entry format for a variable name (DOUBLE PRECISION is truncated to DOUBLE).

6.3 SEARCH Table Formats

The SEARCH program utilizes three tables. Each condition that requires a search has two distinct tables plus a third table common to all conditions. The conditions that use the search are:

(1) An all alphanumeric statement.

(2) A , after an all alphanumeric identifier.

(3) An identifier, then statement terminated a zero parenthesis count.

(4) An identifier, parenthesis count equal zero, then a slash.

(5) An identifier, parenthesis count equals zero, then a left parenthesis.

(6) An identifier, then a slash.

(7) An identifier, parenthesis count equal zero, then an alphabetic character.

(8) An identifier, then parenthesis count equal zero, then a ,.

(9) The initial statement.

The search keys on the number of alphanumeric characters that appear in the initial string. Table 1 thus has one word containing the number of statement possibilities as determined by the length of the string. In addition to this Table 1 has a pointer to the Table 2 location that contains the following information: The location (in Table 3) of the display code representation of the statement identifier and the location to jump to upon a successful match. The format of Table 1 is:

```
VFD  12/200nB, 48/Table 2 location
```

n = the number of identifier possibilities

The format of Table 2 is:

```
VFD  30/jump address, 30/Table 3 location
```

The format of Table 3 is:

```
VFD  12/200mB, 48/statement code

VFD  60/display code picture of identifier
```

m = the number of characters in picture

Thus for a given condition, the n Table 2 entries (in sequence) are used to find the pictures to compare to the string.

CONVERT

1.0 General Information

CONVERT converts the display code representation of a constant to its internal binary form. The binary form is placed in a table and the user now refers to the constant by the I, H of the table name and the CA of the location of the constant in the table CON..

2.0 Usage

2.1 CONVERT

Determines which of the three options is desired.

2.1.1 The constant is converted to binary form, placed in CONLIST, if not already there and the caller informed of I, H and CA to be used to reference the constant. Conversion and add to CONLIST.

2.1.2 The constant is converted to binary form and returned to the caller. Conversion only.

2.1.3 The constant in the form supplied by the caller is placed in CONLIST if not already there and the caller informed of I, H and CA. Add to CONLIST only.

2.2 Calling Sequence and Returns.

The calling sequence is RJ CONVERT. Case 2.1.1 expects register B1 to be +0 and the E-list entry for the constant to be in register X1. Upon successful return, register X1 holds H in bits 0=17, I in bits 18-29 and CA in bits 30-47, all other bits being 0.

Case 2.1.2 expects register B1 to be negative and the E-list for the constant to be in register X1. Upon return, the converted form of the constant is held in X1, and X2 if the constant is a two word element.

Case 2.1.3 expects register B1 to hold 1 or 2 the number of words in the caller supplied constant and X1 and X2 to hold the one or two word element, X1 the first part of the constant and X2 the second part.

2.3 Processing Flow Description

CONVERT quickly determines the option desired. The first call for either case 2.1.1 or 2.1.3 will cause the symbolic name CON. to be placed in the symbol table. For Case 2.1.1, the display code of the constant is formed by DEC. DEC is called to convert the constant, the CONLIST is searched for the converted form of the constant. If the constant already appears, the I, H and CA is returned to the caller. Otherwise, the constant is placed in CONLIST. For Case 2.1.2, DEC is called to convert the constant. DISPLAY is the routine called to process all forms of Hollerith constants. In case 2.1.2 the first word of the constant is returned in X1. For cases 2.1.1 and 2.1.3, the constant is put in the COMPASS file following a USE HOL.. The constant instruction is a DIS n, except for the last word which is a HOL nH, nR, or nL depending on the constant type (HOL is opsyned to DATA) depending upon the form of the Hollerith constant. The first call to DISPLAY will cause the symbolic name HOL. to be placed in the symbol table.

3.0 Diagnostics Produced

3.1 Fatal to Compilation

CONLIST TOO BIG. TOO MANY CONSTANTS. MORE MEMORY REQUIRED.

3.2 Fatal to Execution

CONSTANT CONVERSION ERROR.

4.0 Environment

CONVERT expects CON1 (RA+26B), DO1 (RA+30B), DOLAST (RA+31B) and ELAST (RA+14B) to be set prior to being called. CONVERT maintains CON1 and CONLAST, the first and last locations used by CONLIST. 100 locations are initially reserved for CONLIST. If more room is required, the DO tables are moved 100 locations, if possible, and the pointers maintained. When 100 more locations are not available ((ELAST) being the highest +1 address that can be used), a fatal to compilation diagnostic is issued via FATALER.

5.0 Structure

5.1 CONVERT determines if the option is "store only" and if so, jumps to PACK. If not, a check is made for the constant being any form of Hollerith and if so, a jump is made to DISPLAY. For the "convert only" option, a jump is made to PRECON to arrange the input to DEC. For the "convert and store" option, a jump is made to PRECON, then PUT.

5.2 PACK determines the first call for a store and calls SYMBOL to put the name CON. in the symbol table and retain its ordinal for use as the H field in the I, H and CA information.

5.3 PRECON arranges the display code of the constant as follows: digits are packed a maximum of seven per word left adjusted to bit 59 and zero filled, + -. or B are stored in bits 0-5 with zero fill, and E or D are stored in bits 54-59 with zero fill.

5.4 PUT places the one or two word converted constant (or caller supplied constant) into CONLIST if the constant is not already in CONLIST. Initially 100 locations are reserved for CONLIST and will be expanded 100 locations at a time moving the DO tables if necessary until the time when 100 locations are not available (CONLIST) or the DO tables running over ELAST when a fatal to compilation diagnostic is issued.

5.5 DEC does the actual conversion.

5.6 DISPLAY determines the first call for a storing option and calls SYMBOL to place the name HOL. in the symbol table and retain the ordinal to use as H in the I, H and CA information. For the convert only option, the first word of the Hollerith constant is returned to the caller in register X1. For any storing option, the constant is placed in the COMPASS file and the user returned the I, H and CA information. Any ten character part of the constant is issued with a HOL nH or L or R n being the number of characters. The first Hollerith constant put in the COMPASS file will be preceded by a HOL. BSS 0B line. A DATA instruction is put in the COMPASS file to terminate each constant with a word of zeros. Finally a USE DATA. is put in the COMPASS file.

6.0 Formats

6.1 I, H and CA word returned to the caller is:

```
VFD  12/0,18/CA,12/I,18/H
```

6.2 CONLIST is the name of the table of converted or user supplied constants. .TRUE. is converted to -1 and .FALSE. is converted as +0.

DATA

1.0 General Information

DATA resides in Phase 2 of Pass 1 and processes DATA statements after SCANNER has transformed the statement into E-list format. Output from DATA consists of DATA statements being sent to the COMPS file along with ORG and REPI macro calls. Examples are given later in the sections discussing output in more detail.

Data statement syntax is described in first page of the program listing and in the reference manual.

2.0 Overall Structure

Upon entry, RAS is called to do a quick backwards scan of the statement to locate the beginning of the constant item lists and the variable element lists. The pointers are saved in a table, and the main loop entered. First, a constant list is scanned, the constants converted to binary numbers and placed in a temporary table built in working storage. Next the corresponding variable list is scanned and as each element or nest of loops is processed, output routines are called to issue ORG macro calls to set the FWA for data placement and to extract items from the data table and output DATA and REPI pseudo ops. Last of all, we check to see if the number of items in the variable list match the number of items in the constant list and if they don't, we issue an informative diagnostic to that effect and loop to process the next pair of lists.

2.1 Listing Structure

Each section in the program listing is preceeded by a sub-title. A breakdown of the sections is as follows.

Data statement syntax definition

Error message ordinals and error exits

Description of non ANSI extensions

Macro definitions

Local variable definitions, grouped by routine that defines them

Main loop

Data item list processing

Data variable list processing

Output subroutines

PDV - data variable processing

3.0 Subroutines

3.1 RAS - Remove Alternate Syntax

Function - To build up a table of pointers to the start of the variable and data item lists.

Method - A EOS marker is placed at the beginning at the statement, and the statement is scanned backwards for the start of the data item and variable lists. The pointers are saved in working storage in the DIL table. When the alternate syntax is encountered, the ='s sign and right parenthesis are replaced by slashes.

ENTRY/EXIT conditions

On exit, the DIL table has been built starting at FWAWORK, its length set in the location N.DIL and the FWA for the data item table, O.DIT set.

The format of the DIL is:

24/0,18/FWA of var list, 18/FWA of con list.

cells set

N.DIL = number of initialization lists

O.DIT = FWA of data item table (FWAWORK)

NONANSI = initialized to 0. Set if alternate syntax is encountered.

Error Messages

SYNTAX ERROR IN DATA STATEMENT

RAS is called once per data statement from the main loop. It calls STD to scan to the start of the constant item list or variable list.

BIT - Build data item table

BIT scans the constant list for syntactic correctness and calls the subroutines ADIT and ADDCON to make entries in the data item table. The format of the DIT is described in the comments preceeding the routine. It also keeps track of the number of items in the data list (N.ITEM).

Most of BIT is rather straight forward and we will only discuss the processing of repetition lists and complex constants.

3.2 REP List Processing

Recognition: a REP list is recognized when we encounter an unsigned integer constant followed by a * or (at paren level 0. At this point, we set the following flags CLOSREP = -1 if the rep list is not enclosed in parentheses, i.e., 5*10, else CLOSREP = 1.

An initial entry is made in DIT where the rep count is saved. REPFLAG = 24/0,18/N.ITEM,18/pointer to DIT entry for rep list start, and N.ITEM is cleared.

When the end of a rep list is encountered, which is after processing the next constant if CLOSREP = -1 or a parenthesis if CLOSREP =1, CRL is called to close out the repetition list.

It performs the following functions:

Updates the DIT entry for the rep list to include the number of items in the rep list, the index to the next list and sets the constant item length flag (CIL) for the list if all the data items have the same length (word count). This last flag is used by the output routines when they are trying to output REPI macro calls. It also updates N.ITEM where N.ITEM = RF * RL+N.ITEM and clears CLOSREP and REPFLAG.

3.3 CFCD

When a (is encountered, the updated paren level counter (PL) is compared to CLOSREP to see if this left paren started a rep list or it must be the start of a complex constant.

If it is the start of a rep list, then we jump back to the main loop to process the next item, else we call CFCD to look ahead and check for a complex constant, and return false, or convert the constant and return true. In the case that CFCD returns false, either the left parenthesis is meaningless and ignored, or an attempt to nest 2 parentheses groups and flagged as a syntax error.

If CFCD returned true, then the parenthesis level is decremented by 1 and ADIT is called to add the constant to the DIT.

Entry

A5 = (SELIST)

A4 = points to start of con list, the /.

Exit

DIT built, length in L.DIT and number of items in N.ITEM.

Note for the list /1,3(1,2)/ N.ITEM = 1+3*2 = 7

PL = REPFLAG = CLOSREP = 0.

Error messages

Syntax error in data item list

Illegal item following + or - sign

2 nested rep lists.

Illegal separator following a constant item.

CALLS - CRL, CFCD, CHKSC,CADIT and ADDCON

3.4 CHKSC - Check for Small Constant

CHKSC checks to see that the given constant is type integer or octal, and that the converted value is between 1 and 377777B. It returns the converted value of the constant in X6.

Error messages

Do limit or rep factor in a DATA statement must be an integer or octal constant between 1 and 131K.

CALLS - CONVERT

3.5 ADDCON

ADDCON calls convert to convert the constant to binary, exclusive ors in the value of the sign (+ or -) and calls ADIT to add the constant and prefix word to the DIT. In the case of Hollerith constants the number of words in the constant and the remainder is calculated and a DIT entry is made for the prefix word and the E-list for the constant. In the case that a Hollerith constant is preceeded by a - sign, the CONSTOR's entry is complemented.

Entry

X4 = E-list for the constant

X1 = E-list for the constant with upper 12 bits = 0

X7 = value of the sign (+0 or -0)

Exit

Constant added to ADIT

Registers restored by a call to macro ≠ GETE ≠, point past the constant.

CALLS CONVERT, ADIT

3.6 ADIT

ADIT is called to add up to 3 words to the DIT. The first word is in X6, the second in X1, and third in X2. B1 = number of words -1 to be added. ADIT updates L.DIT and checks for memory overflow.

Error messages

DATA TABLE MEMORY OVERFLOW, INCREASE FL

BVT

BVT scans the variable list for syntactic correctness, accumulates information about each variable or nest of loops in the list, and calls the output routine, MDL, to match up the variable and data item lists.

The processing of implicit DO loops will be discussed here.

PDV is called to process the array name and set cells containing the dimensions, etc.

PSS is called to process th subscript list of the array and set up the subscript table (SST) information containing the constant multipliers (C1's), constant addends (C2's) and index variables. Then, we process the loop variable and limits, matching up the index variable with the variables appearing in array subscript expression and converting the loop limits to binary. Next, we check to see that for each subscript C * 11+C2≥1 and then reduce the loop to normal form.

The set of loops:

```
(((A(C11*I1+C12,C21*I2+C22,C31*I3+C32), I=l11,ul1,in1),
                                        J=l12,ul2,in2),
                                        K=l13,ul3,in3)
```

where I1, I2, I3 is some permutation of IJK, can be reduced to:

```
(((A'(m1*I1,m2*I2,m3*I3), I'=1,t1), J'=1,t2), K'=1,t3)
```

which we will call normal form.

The formulas for reduction to normal form are:

1. $I=in_1*I'+ll_1-in_1, J=in_2*J'+ll_2-in_2\cdots$

2. $M_i=in_1*C_{1i}$

3. $A'=A+\sum_{j,C_{1j}\neq 0} dmj*\{C2j+C_{1j}(llj-incj)\}$

   $+\sum_{j,C_{1j}=0}\{dmj*(C_{2j}-1)\}*2^{sdpf}$

4. $t=(ul_i+in_i-ll_i)/in_i$, trip count for the loop if $ll_i \leq ul_i$

   Where sdpf is the single/double precision flag (0 or 1) and the dmj are the dimensional multipliers for the array.

   $dm_1=1, dm_2=dim_1, dim_3=dim_1*dim_2$

   The difference A'-A is the bias due to subscript calculation and accumulated in the cell BIAS.

   One may derive 2 and 3 from 1 and the definition of LOCF

   $A(i_1,i_2,i_3)=LOCF(A)+\{\sum_j dmj*(_{ij}-1)\}*2^{sdpf}$

The code from BVT9 to BVT14 performs this reduction. The loop information for the ith loop is combined and saved in LPINF(I) whose format is described in the comments preceeding BVT.

For the loop nest

((A(j,i), i = 1,5), J = 2,6,2)

which reduces to

((A'(j,i), i = 1,5) J = 1,3)

we would have:

LPINF (1) = 6/1, 18/1, 18/5, 18/dim1

LPINF (2) = 6/0, 18/2, 18/3, 18/dim2

The next sequence of code (BVT 15 - BVT 18) collapses the innermost loops when the subscripts are in standard order (IJK).

As an example:

((A(i,j), i=1, dim1), J=1,N) may be collapsed to

(A(i), i = 1 dim1*N) which is easier for the output routines to process.

Finally, one calculates the sum

$$BIAS + \{ \sum_j dmj(mj \cdot tj - 1) \cdot 2^{sdpf}$$

checks it to see that it is less than MAX(dimj, EQUIV extended of the array), then calls MDL to match up the lists.

Entry

A5 = SELIST

A4 = points to start of var list

N.ITEM = number of items in data list

N.ITEM = 0 lists match

less than 0 if var list longer than con list

greater than 0 if con list longer than var list

CALLS - PDV, PSS MDL, CHKSC

3.8 PSS

PSS processes the subscript list associated with an array element or appearing in a DO nest up to the closing paren. PSS consists of three phases. First, the subscripts are syntax checked and the E-list for the constant multipliers, addends and subscript variables saved in the subscript into block, SST. Next, the constant multipliers and addends are converted to binary. Finally we search for multiple appearences of a subscript variable, and if found eliminate them by reducing the number of variable subscripts and adjusting the constant multipliers and addends.

Entry

A4 points to (following the array name.

Exit

SELIST points past the closing).

B7 = N.VSUB = number of variable subscipts

N.SUBS = number of subscripts

CON1 = constant multipliers

IVAR = elist of subscript variables

CON2 = constant addends

Example A(2*I-1,3) results in

N.VSUB = 1

N.SUBS = 2

CON1(1) = 2, CON1(2) = 0, CON1(3) = 0

IVAR(1) = "I", IVAR(2) = 0, IVAR(3) = 0

CON2(1) = -1, CON2(2) = 3, CON2(3) = 0

CALLS - CHKSC

3.9 PDV

PDV is called from BVT process names occuring in data statements which are not dummy variables occurring in DO nests. The functions of PDV are to return to the calling routine information of interest to it, which include: symtab ordinal, bias due to equivalence, words of storage/element, number of dimensions, dimensions, etc. PDV must also define the address of usage defined variables that occur in DATA statements and call ADDREF to add definitions to the reference map.

The flow structure of PDV is straight forward but modifiers should be careful not to destroy information being saved in the B registers, or X1 and X2 which contain the symtab entry.

When a usage defined variable occurs in a data statement, we must define its address immediately, instead of waiting until ENDPRO time, since we will issue storage for it. Since there is no space in the symbol table entry during pass 1 (the RA field is used by the DEBUG processor), we save the address we assign it and the symbol table ordinal in a separate table which is processed at ENDPRO time. PDV also sets the common bit in the symbol table entry so further occurrances in DATA statements do not cause the address to be defined again and so ENDPRO doesn't attempt to define its address. After ENDPRO processes the symbol table (PST) and defines the address's of the usage defined variabels that have not occurred in DATA statements, if processes the DATA table and turns off the common bit and inserts the saved address in word B of the symbol table entries.

Entry

Registers point to variable name (X1, X4, A4, A5).

Exit

Registers restored for syntax scanning and point past variable name.

The following locations are set:

SNAME — E-list for the variable name.

(Used for error messages to point to the last good name processed).

SDPF — 0 if the name is a single precision variable or array (1 word/element), 1 if double or complex.

N.DIMS — 0 if a simple variable, else = to the number of dimensions.

DIM — This block holds the dimensions of the array.

DIM.MUL — This block holds the dimensional multipliers for subscript calculations. (1,DIM1,DIM1*DIM2)

DVT — This two word block holds the symtab ordinal, the bias due to equivalence, the number of elements in the array, the number of words of storage allocated to the symbol and some miscellaneous flags (see description preceedubg BVT).

EEL — 0 unless the name is an array reference appearing in a DO nest. In this case, EEL = equivalence class length - bias of the array relative to the equivalence class.

CALLS - SYMBOL, ADDWD, CFO, ADDREF

Error Messages

This name may not appear in a data statement.

Name may not be function, external, formal parameter or in blank common.

4.0 Output Routines and Methods

At present, all the data processor output goes to the COMPS file in the form of display code line images. The information put out consists of setting the FWA for data placement followed by the data. To achieve the first objective, the ORG pseudo was redefined so that it would act as a NAME BSS 0 statement in the case that the variable had not yet had storage issued for it (in DPCLOSE) and as a ORG in the case that storage had been previously allocated.

In order to output the data in the most efficient way (minimize the size of the binary file), one wishes to make use of the REPI pseudo whenever possible. The analysis necessary to do this in the one dimensional case is contained in the subroutine OIC. Code necessary for the 2 and 3 dimensional cases was not done due to lack of time, but extra analysis could be included in MDL.

The definitions of the ORG and REPI macros used by the compiler can be found in FTNMAC.

Data Productions

1. Single element DATA A(C) / CON /

```
ORG A,C-1
DATA CON
```

2. 1 dimensional loop (A(m*l), l = 1,t) / con list /

   a. con list = C1, C2,...,Ct

```
ORG A + m-1      where redundant "ORGS"
DATA C1          are suppressed if the
ORG A + 2*m-1    length of the data item = m
DATA C2
.
.
.
ORG A, t*m-1
DATA Ct
```

   b. con list = rf * (C1, C2,..., Crl)

where all the Ci are the same length, CIL, here there are two cases and we look at the address difference between consecutive elements

Note that the dmp contribution comes in since BVT has reduced loops with constant subscripts, like

(A(2,i), i=1,5) to the form $A^1(i)$, i=1,5)

The number of times we can use the data list in the loop is

n = min (rf, t/rl)

where n > 1 if we are to use the REPI pseudo.

The cases are as follows:

1. CIL>2 odd, no replication

2. RL = 1 or AA = CIL

We output a ORG, followed by a DATA followed by a REPI macro call

```
    ORG A,BIAS                  m'=RL*AA
S.  SET  *
    DATA  C1,C2,...,Crl
    REPI S/S.,+m', B/rl.CIL, C/n-1, I/m'
```

3. RL>1 and AA>CIL

For this case, the data will not occupy contiguous locations in storage. Since the REPI pseudo is not capable of moving a non-contiguous sequence of data, we must issue a separate ORG and REPI for each piece of data.

The sequence that OIC expands issues to the COMPS file is:

```
X   SET 0
    DUP RL
    ORG A(m*(1+X))          m' = AA.RL
S.  SET *
    DATA Cx+1
    REPI S/S.,B/CIL, C/n-1, I/m', D/S.+m'
```

Output for the case when the data list consists of single items and replication lists intermixed simply consists of breaking down the loop into a sequence of loops and processing each sub loop in sequence.

Processing of an irreducible nest of loops such as

```
((AA(j,i) i=1,10), j=1,5)
```

consists of calling OIC to process the inner loop n times with the outer loop variables being incremented after each call to OIC.

Output Subroutines

Here the subroutines fall into three classes:

1. Output information to the COMPS file

   ODV — Outputs ORG macro call.

   ORP — Output REPI macro call.

   ODI — Output a data item to the COMPS file calls IDW or OHC to issue the data words to the COMPS file.

   IDW — Output a single data word to the COMPS file. ODV and ORP call the subroutine FMAC format and translate the information to display code IDW calls FMAC and BTOCT.

2. Maintain pointer to the data item table.

GNI performs this function and OIC decrements the rep count when it decides it can use the REPI pseudo op to replicate a list.

3. Controllers

MDL - decides what case we are processing and calls the output routines or OIC.

OIC - outputs data initalization code for the sequence of elements described by the loop.

```
(A (m*i), i = 1,t)
```

CALLS GNI, ODI, ODV ORP

ERPRO and FORMAT

1.0 General Information

1.1 Task Description - Error Processing

The error processing during compilation is divided into two routines: ERPRO in PASS 1, and PS2CTL in overlay 1,3. The final output resulting from detecting an error will be the card sequence number (compiler generated) on which the error occurred, the severity (fatal to execution, etc.), an optional symbol, a name or word to further clarify the message, and the actual diagnostic (up to 106 characters). ERPRO is located in the 1,1 overlay.

1.2 Task Description - FORMAT Scanner

The FORMAT scanner processes FORMAT statements and checks for errors at compilation time. The scanner squeezes out blanks and redundant commas. Before scanning the FORMAT for validity, the statement label is checked for validity, recorded in the symbol table, and sent to the COMPS file.

2.0 Usage

2.1 Entry Point Names - Error Processing

In general the calling sequence is:

```
SB6     error number
SB7     return
EQ      entry point
```

Using the above calling sequence, ERPRO would expect an E-list entry in X4, or if X4 is zero a display code message in X3 (bits 48-59 zero, bits 0-47 display code).

Using the following calling sequence, no message is expected in X3 or X4.

```
SB6     -error number
SB7     return
EQ      entry point
```

The parameter in B6 is a symbol or number which is equated to the ordinal of the error in the error directory table in overlay 1,3.

2.1.1 ERPRO

This entry is used for all messages which resulted from errors which are fatal to execution.

2.1.2 ASAER

This entry is used for all messages which denote non-ANSI usage.

If the X option is not selected, this entry point is changed to a JP B7 instruction by PH1CTL.

2.1.3 FATALER

This entry is used for all messages which resulted from errors which are fatal to compilation.

A reference to this entry will result in making an entry in the error table and setting the fatal to compilation flag (FX). Control does not return to the caller. Calls to SCANNER are then made until an END card is encountered, then a request to load PASS 2 is made.

2.1.4 ERPROI

This entry point is used for all diagnostics which are informative in nature.

Informative diagnostics up until 12 minus the maximum are placed in the table and at that time an informative diagnostic is issued stating that no more informative diagnostics will be put in the table.

2.2 FORMAT has one entry point.

2.2.1 Upon entry with a legal statement label, FORMAT scanning takes place. FORMAT is entered by both PH1CTL and PS1CTL, since formats may be among both executable and nonexecutable statements.

2.2.2 Calling Sequence

FORMAT is entered via a return jump and upon completion of its tasks, exits through its entry point.

2.2.3 Flow of Processing

The characters which comprise a FORMAT, beginning with the left parenthesis, are scanned sequentially until the matching right parenthesis or an irrecoverable error condition is encountered.

3.0 Diagnostics

3.1 Error Processing

Number 206 Informative DUE TO THE NUMEROUS ERRORS NOTED, ONLY THOSE WHICH ARE FATAL TO EXECUTION WILL BE LISTED BEYOND THIS POINT
Number 110 Fatal to Compilation ERROR TABLE OVERFLOW

3.2 FORMAT

3.2.1 Fatal to Execution

PRECEDING CHARACTER ILLEGAL AT THIS POINT IN CHARACTER STRING. ERROR SCAN FOR THIS FORMAT STOPS HERE.

ILLEGAL CHARACTER FOLLOWS PRECEDING FLOATING POINT DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.

ILLEGAL CHARACTER FOLLOWS PRECEDING A,I,L,O OR R DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.

ILLEGAL CHARACTER FOLLOWS TAB SETTING DESIGNATOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.

ILLEGAL CHARACTER FOLLOWS PRECEDING SIGN CHARACTER. ERROR SCANNING FOR THIS FORMAT STOPS HERE.

PRECEDING CHARACTER ILLEGAL, SCALE FACTOR EXPECTED. ERROR SCANNING FOR THIS FORMAT STOPS HERE.

PRECEDING HOLLERITH COUNT IS EQUAL TO ZERO. ERROR SCANNING FOR THIS FORMAT STOPS HERE.

FORMAT STATEMENT ENDS BEFORE LAST HOLLERITH COUNT IS COMPLETE. ERROR SCAN FOR THIS FORMAT STOPS AT H.

FORMAT STATEMENT ENDS BEFORE END OF HOLLERITH STRING. ERROR SCANNING STOPS HERE.

PRECEDING HOLLERITH INDICATOR IS NOT PRECEDED BY A COUNT. ERROR SCANNING STOPS HERE WITH FORMAT INCOMPLETE.

ZERO LEVEL RIGHT PARENTHESIS MISSING. SCANNING CONTINUES.

PRECEDING FIELD WIDTH OUTSIDE LIMITS FOR RECORD SIZE. SCANNING CONTINUES.

PRECEDING RECORD OUTSIDE OUTER LIMITS FOR RECORD SIZE. SCANNING CONTINUES.

TAB SETTING IS OUTSIDE OUTER LIMITS FOR RECORD LENGTH. SCANNING CONTINUES.

3.2.2 Non-ANSI

PLUS SIGN IS AN ILLEGAL CHARACTER.

PRECEDING FIELD DESCRIPTOR IS NON-ANSI.

FLOATING POINT DESCRIPTOR EXPECTED FOLLOWING SCALE FACTOR DESIGNATOR.

TAB SETTING DESIGNATOR IS NON-ANSI.

HOLLERITH STRING DELINEATED BY SYMBOLS IS NON-ANSI.

3.2.3 Informative

SEPARATOR MISSING, SEPARATOR ASSUMED HERE.

X-FIELD PRECEDED BY A BLANK, 1X ASSUMED.

X-FIELD PRECEDED BY A ZERO, NO SPACING OCCURS.

PRECEDING FIELD WIDTH IS ZERO.

PRECEDING FIELD WIDTH SHOULD BE 7 OR MORE.

FLOATING POINT DESCRIPTOR EXPECTS DECIMAL POINT SPECIFIED. OUTPUT WILL INCLUDE NO FRACTIONAL PARTS.

FLOATING POINT SPECIFICATION EXPECTS DECIMAL DIGITS TO BE SPECIFIED. ZERO DECIMAL DIGITS ASSUMED.

REPEAT COUNT FOR PRECEDING FIELD DESCRIPTOR IS ZERO.

FIELD WIDTH IS OUTSIDE INNER LIMITS. CHECK USE OF THIS FORMAT TO ASSURE DEVICE CAN HANDLE THIS RECORD SIZE.

PRECEDING SCALE FACTOR IS OUTSIDE LIMITS OF REPRESENTATION WITHIN THE MACHINE.

SUPERFLUOUS SCALE FACTOR ENCOUNTERED PRECEDING CURRENT SCALE FACTOR.

RECORD SIZE OUTSIDE INNER LIMITS. CHECK USE OF THIS FORMAT TO ASSURE DEVICE CAN HANDLE THIS RECORD SIZE.

FIELD WIDTH OF PRECEDING FLOATING POINT DESCRIPTOR SHOULD BE 7 OR MORE THAN DECIMAL DIGITS SPECIFIED.

NUMERIC FIELD FOLLOWING TAB SETTING DESIGNATOR IS EQUAL TO ZERO, COLUMN ONE IS ASSUMED.

NUMERIC FIELD OMITTED IN PRECEDING SCALE FACTOR. ZERO SCALE ASSUMED.

NON-BLANK CHARACTERS FOLLOW ZERO-LEVEL RIGHT PARENTHESIS. THESE CHARACTERS WILL BE IGNORED.

TAB SETTING MAY EXCEED RECORD SIZE DEPENDING ON USE.

3.2.4 Each error message will be preceded by a 48 bit message stating the card and column number of the error encountered. Computation and the form of this message is described in Section 8.

4.0 Environment

4.1 Error Processing

4.1.1 Information provided by other processors

In location 46B and 47B, SCANNER places information regarding the current card number. Location 46B contains in display code the line number as printed on the listing, location 47B contains in binary an offset count which ranges from 1 to 10.

The location of the error table is an entry point name in FTN called ERTABL.

NASAFLG (issue non-ANSI usage errors) is set by the control card cracker.

4.1.2 Information generated by ERPRO.

N.FERR contains the number of errors in binary encountered during a single compilation.

4.2 FORMAT

FORMAT scanner expects characters to be packed ten characters per word in display code, where the first character is a left parenthesis. FORMAT expects the first word of information at the location specified by SELIST, and the last word of information at the location specified by ELAST. FORMAT allows a maximum of three levels of parentheses, an input record length of 150 characters, and an output record length of 137 characters. In general, formats must be in accordance with ANSI FORTRAN standards, with the addition of the tab setting and Hollerith string capabilities. Legitimate format field descriptors are of the following form:

```
((+ / -) (n) P) (r) <D / E / F / G> w.d
(r) <A / I / L / O / R> w
nHh1h2...hn
(n) X
*...* or ≠...≠
Tm
```

where:

1. In the above description, a slash separates alternatives; angle brackets denote that one and only one of the enclosed alternatives must be chosen; parentheses denote that none or one of the enclosed alternatives may be chosen.

2. The letters D, E, F, G, A, I, L, O, R, H, and X indicate the manner of conversion or editing between the internal and external representations and are called the conversion codes.

3. w and n are integer constants representing the width of the field in the external character string.

4. d is an integer constant representing the number of digits in the fractional part of the external

character string. If d is omitted, it is assumed to be zero.

5. m is an integer constant representing the tab setting for the external character string.

6. r is the repeat count (an optional, non-zero integer constant) indicating the number of times to repeat the succeeding basic field descriptor.

7. (+ / -) (n) P is optional and represents a scale factor to be applied to the processing of the succeeding conversion code if a D / E / F / G.

8. Each hi is one of the characters capable of representation by the processor.

9. *...* or ≠...≠ encloses hollerith information (excluding an asterisk), up to one record in length.

10. For all descriptors other than *...*, or ≠...≠ field width must be specified; for descriptors of the form D / E / F / G w. must be greater than or equal to d+7.

Format field separators are the slash and the comma. Field separators are used to delimit field descriptors. Field separators are optional in the following cases:

1. after *...*, ≠...≠
2. after nHh1h2...hn
3. after nX
4. after (+ / -) (n) P
5. after another field separator
6. before or after a right parenthesis

In all other cases, a field separator is expected, and a diagnostic is issued if the separator is missing. Scanning of the format will continue in such a case. Blanks and commas, where unnecessary, are squeezed out of the format specification.

5.0 Structure

5.1 Major subroutine names in ERPRO.

5.1.1 ERPRO

This subroutine checks if room exists in the table and determines type of parameter that accompanies the message.

5.1.2 OPER

This subroutine decodes the E-list element.

5.1.3 TABOFLO

This subroutine issues diagnostic 110 and makes the calls to SCANNER.

5.1.4 PK

This subroutine sets up the entries in the error table and updates the cell (ERLOC) which contains the address of the next available cell in the error table.

5.2 FORMAT

5.2.1 Transition Diagram

Format scanner has been implemented utilizing transition diagram oriented processing. A transition diagram describes action to be taken for each syntactic type encountered in a string. The transition diagram consists of circles, boxes, unbroken, and broken line segments where:

:: = a NODE, or state in the flow which has been reached at some point in the string.

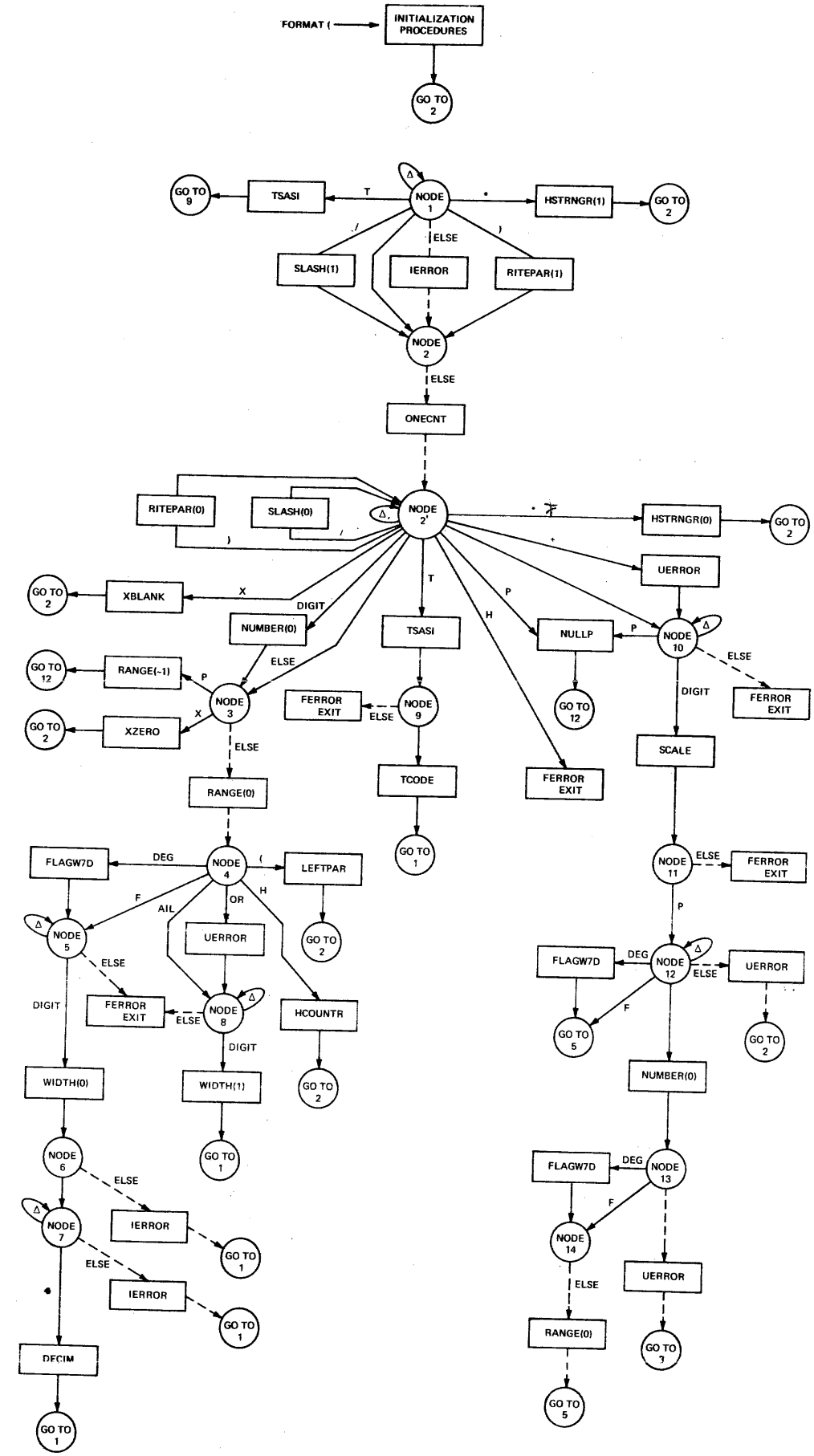
:: = a set of intermediate processing on the string between nodes, or states, which can be made analogous to FORTRAN subroutine.

:: = action in processing the string. Over a solid line segment, character advancement takes place; over a broken line segment, character advancement does not take place. The character(s), or group of characters (i.e. digit :: = (0,1,2,3,4,5,6,7,8,9)) which direct the processing to a

particular state are inscribed on the line segment.

Character advancement can also occur in intermediate processing.

The transition diagram which traces the flow of processing for the format scanner follows.


FORMAT (
INITIALIZATION PROCEDURES
GO TO 2
NODE 1
TSASI
GO TO 9
HSTRNGR(1)
GO TO 2
SLASH(1)
IERROR
RITEPAR(1)
ELSE
NODE 2
ONECNT
NODE 2'
RITEPAR(0)
SLASH(0)
HSTRNGR(0)
GO TO 2
UERROR
XBLANK
GO TO 2
NUMBER(0)
TSASI
NULLP
NODE 10
RANGE(-1)
GO TO 12
NODE 3
XZERO
GO TO 2
FERROR EXIT
NODE 9
GO TO 12
FERROR EXIT
SCALE
RANGE(0)
TCODE
FERROR EXIT
GO TO 1
FLAGW7D
DEG
NODE 4
LEFTPAR
GO TO 2
NODE 5
UERROR
FERROR EXIT
NODE 8
HCOUNTR
GO TO 2
WIDTH(0)
WIDTH(1)
GO TO 1
NODE 6
IERROR
GO TO 1
NODE 7
IERROR
GO TO 1
DECIM
GO TO 1
NODE 11
FERROR EXIT
FLAGW7D
NODE 12
UERROR
GO TO 5
GO TO 2
NUMBER(0)
FLAGW7D
NODE 13
NODE 14
UERROR
RANGE(0)
GO TO 5
GO TO 1

special symbols omitted.

5.2.2 Micro Definitions for Transition Diagram

Micro definitions for the format transition diagram are formed in the following manner:

node micro::= branch(branch) otherb otherb

branch::= char test mask test ignore : transfer designation

otherb::= /(char):transfer designation

transfer designation::= node name ,routine name (param)

char test::= = char expr

char ::= A B C ... * (+ - * / $, . ( )

expr ::= compass expression designated by more than one character.

mask test::= (char) (char)

char ::= A B C ... 8 9 + - * / $, . ( )

ignore::= ,(char)

char ::= A B C ... * (+ _ / $, . ( )

node name::= name

routine name::= name

name::= letter (letter number $.)

letter::= A B C ... Z

number::= 0 1 2 3 4 5 6 7 8 9

param::= compass expression

For example, at node 7:

= ,NOPACK:NODE7 - a blank is not packed, the flow is advanced one character and sent back to node 7.

(0123456789):NODE1,DECIM - a digit is packed, the flow is advanced one character, and sent to NODE1, via a set of intermediate processing, DECIM.

/ELSE:NODE1,IERROR(7) - any other character at this node inhibits character advancement, and flow is sent to NODE1, via IERROR, the informative error processor, with a parameter of 7.

5.2.3 Table Formed from Definitions and Table Processor

The micro definitions generate one word table entries, which are acted upon by the transition diagram table processor, TRANSIT, all of which is located in FLY. TRANSIT processes the character string along the path defined by the micro definitions of the transition diagram, fetching and storing characters where required.

5.2.4 Intermediate Processing, i.e. Subroutines Used

5.2.4.1 NUMBER

Converts a string of display code numerical digits into a binary number which is stored relative to location NUMN, with a displacement of the input parameter (-1,-0,1); the input parameter specifies the number to be decimal digits, a repeat count or skip span, or a field width. Control is returned to the address specified.

5.2.4.2 RANGE

Checks for valid result of NUMBER routine; range to be checked is specified via the calling parameter. If number is out of range, the error processor is called. Control is returned to the address specified.

5.2.4.3 FLDCHEK

Checks range of field elements; computes total field length and checks the range; record length is increased by the length of the total field. Record count is saved in a pushdown table which saves information for the 3 levels parentheses. If the record count is longer than one record, an informative error is produced. Control is returned to the address specified.

5.2.4.4 WIDTH

Field descriptor width handler; calls NUMBER(0), RANGE(1), and FLDCHEK(1). Parameter (0) implies a floating point descriptor, and if the field width is not 7 or greater, an informative error is produced. Parameter (1) for other descriptors, and no test is made. Control is returned to the address specified.

5.2.4.5 DECIM

Handles decimal digits portion of floating point descriptors; calls NUMBER(-1), and if descriptor is D, E, or G. a check is made for field width greater than or equal to 7 + decimal digits specified. If the descriptor fails this test, an informative error is produced. Control is returned to the address specified.

5.2.4.6 FLAGW7D

Called to turn on a flag indicating a D, E, or G type field descriptor. The flag is utilized by DECIM to determine whether or not to perform a test comparing field width with decimal digits specified. Control is returned to the address specified.

5.2.4.7 ONECNT

Initializes temporary count storage for repeat count, field width, and decimal digits, and turns off flag indicating a D, E, or G specification encountered. Control is returned to the address specified.

5.2.4.8 DELCOM

The last character stored in the string is fetched. If the character was a comma, it is squeezed out of the output string. Control is returned to the address specified.

5.2.4.9 XBLANK

An X descriptor was preceded by a blank, and an informative error is issued to that effect. FLDCHEK is then called to update the record length count. Control is returned to the address specified.

5.2.4.10 XZERO

The skip count is tested for zero; if so, an informative error is issued. If the count is non-zero, FLDCHEK is called to update the record length count. Control is returned to the address specified.

5.2.4.11 TSASI

DELCOM is called to squeeze out redundant commas. A non-ANSI error is produced, and control is returned to the address specified.

5.2.4.12 TCODE

NUMBER(0) is called to convert the tab setting pointer to binary. If the result is zero, an informative error is produced. Otherwise, RECCHEK(1) is called, where the record count is accordingly checked and modified. Control is returned to the address specified.

5.2.4.13 SCALE

NUMBER(0) is called to convert scale factor to binary; then RANGE(-1) is called to check for validity of scale factor. Control is returned to the address specified.

5.2.4.14 NULLP

An informative error is initiated and zero scaling is assumed. The scale flag is turned on; if previously on, and unused, another informative error is produced. Control is returned to the address specified.

5.2.4.15 HCOUNTR

The Hollerith count is fetched, each character is checked against an end-of-statement; if an end of statement is encountered, an error exit is taken. Otherwise, the character is stored, the count decremented, and the loop continued until the count is depleted to zero. FLDCHEK is then called to add to the record count. Control is returned to the address specified.

5.2.4.16 HSTRNGR

Each character is compared with the end of statement and the Hollerith string indicator. While no match is made, character advancement continues. If an end of statement is encountered, an error exit is taken. When a matching

Hollerith indicator is encountered, the character count is sent to FLDCHEK where it is added to the record count. Control is returned to the address specified.

5.2.4.17 SLASH

DELCOM is called when the input parameter indicates, and RECCHEK(0) is called to check for legal record size. Values are checked and modified in a pushdown table which saves record size information for each parenthesis level. Current record count is reinitialized. Control is returned to the address specified.

5.2.4.18 RECCHEK

Current record count is checked for legal record size. If entry was from SLASH, control is then returned to the address specified. If entry was from RITEPAR because of a first level right parenthesis, control is sent to FINISH where the format is sent to the COMPS file. Otherwise, entry was from TCODE, and the current record count is set to the tab setting. The record saving pushdown table is modified, and control is returned to the address specified.

5.2.4.19 LEFTPAR

The parenthesis level is incremented and checked for validity. An invalid parenthesis level causes an error exit to be taken. If the parenthesis level is valid, the level repeat count is preserved in the pushdown table. Control is returned to the address specified.

5.2.4.20 RITEPAR

DELCOM is called to delete redundant commas where appropriate. Parenthesis level is checked for zero level. If so, RECCHEK(-1) is called, and control is sent to close out procedures. Otherwise, appropriate record size updating is performed on the pushdown table. The parenthesis level is decremented by one, and control is returned to the address specified.

5.2.4.21 FINISH

Control is received by scan when a zero-level right parenthesis is encountered. A check is made for extraneous characters. The last word of the format is

packed. If no fatal errors were encountered in the process of scanning, the E-LIST string is inverted and 6 word blocks of COMPASS images are sent to the COMPS file. Entry conditions are restored, and control is returned via a jump to FORMAT.

5.2.4.22 IERROR, UERROR, FERROR

All are entries to the error processing routine, depending upon the type of error incurred. The type is preserved, along with the error number. All critical registers are saved; then the card number and column number in which the error occurred are computed and merged into the 48 bit message word. Control is then released to the appropriate error processor. On return, the critical registers are restored, and control is returned to the address specified by the caller.

6.0 Table Formats

6.1 Error Table Format

Word 1 VFD 12/2,6/Error Number,48/Message

Word 2 VFD 30/Line Count,30/Offset

6.2 FORMAT

Memory Pointers and Flags

DEGFLAG - Flag turned on when D, E, or G descriptor is encountered, is used to determine when field width adequacy tests should be made.

COLCNT - Contains count for current record length; is checked in RECCHEK.

FLAGPON - Flag turned on when scale factor is encountered; turned off when utilized. Checked each time scale factor encountered.

FE - Flag turned on when a fatal error condition has been encountered in a format. This flag inhibits packing the format for the COMPS file.

LEVEL - A counter which keeps track of the parenthesis level, where the first level is level zero.

NUMD - Location which saves the decimal field of floating point descriptors.

NUMM - Location which saves tab settings, and repeat counters.

NUMW - Location which saves the width of format descriptors.

PUSHDOWN - A table which contains four fields of information per word, one word per parenthesis level. The information is used to calculate accumulated record length when an end of record is encountered. For each parenthesis level, the following information is saved:

SL indication of presence or absence of slash in level

GR the group repeat count

NL column count following last slash in level

N1 column count preceding first slash in level

The table will be structured as follows:

```
VFD  6/SL(0),18/GP(0),18/NL(0),18/N1(0)
VFD  6/SL(1),18/GP(1),18/NL(1),18/N1(1)
                .
                .
                .
VFD  6/SL(Max),18/GP(Max),18/NL(Max),
     18/N1(Max)
```

7.0 Modification Facilities

7.1 Error Processing

ERRMAX is an EQU in OPTIONS, controls the size of the error table.

7.2 FORMAT

7.2.1 EQU's

```
MAXMAX    EQU  150  maximum read record length
MINMAX    EQU  137  maximum written record length
PMAX      EQU  615  maximum size scale factor
LEVMAX    EQU  2    maximum parenthesis level
HOLLER    EQU  1R*  hollerith string indicator
```

These limits may be changed by simply modifying the EQU's.

7.2.2 Allowable Formats

Additions and/or changes to the forms allowable for format descriptors may be made by adding to and/or changing the micro definitions in FLY, and/or adding to and/or modifying the specific subroutine handler(s) involved.

7.2.3 Character Manipulation

Characters are fetched and stored using two macros: GETCH and PUTCH, from words packed ten characters per word to words packed ten characters per word, with a one character delay, SAVECHAR, on storage. These macros may be modified without disturbing the rest of the logic of the scanner.

8.0 Method Used

Format scanner is a left-to-right, character by character, one pass scan, implemented through TRANSIT, the main routine in FLY, which sends the format to the part of code indicated appropriate by the transition diagram. The approved format is packed, ten characters per word, and sent six words per line, to the COMPS file. The scan operates on a character recognition basis. Recognition causes control to be sent to an appropriate set of intermediate processing, which expects a particular combination of characters, previously referred to as field descriptors. Permissible descriptors are itemized in Section 4. At the end of a set of

intermediate processing, control is returned to the appropriate state in the flow of the scanner. Scanning terminates when an end of statement is encountered, or an illegal character or character sequence is encountered. A running count is kept of the length, in characters, of the current record described by the format. Calculation of total record length involves utilization of the information stored in the PUSHDOWN table described in Section 6. Calculation and checking is done whenever a slash or a zero-level right parenthesis is encountered. When an error is encountered in the scanning process, the error processor is called, where the card and column number in which the error occurred is calculated. They are computed using the following formula:

```
CD = 21 - CONTS
COL = COLS - 8        where CONTS AND COLS are
                      computed in SCANNER

N =    FWA format - current address
                    *10 + (60 - (6 character
                    pointer))/6

  = CURRENT COLUMN POINTER


WD = (COL+N-1) = RELATIVE WORD POINTER

CDNO = CD+WD = CURRENT CARD POINTER

COLNO = COL+N+5-(66*WD) = CURRENT COLUMN OF CURRENT
                          CARD POINTER
```

This information is packed in the lower 48 bits of the error word in one of the following forms:

```
NNCDNNNN
NNCDbNNN
NNbCDbNN
NNbbCDbN
```

where the first field is the column number and the second field is the card number. This information is then sent to the standard error processing routine.

9.0 Restrictions and Other Remarks

9.1 ERPRO

None

9.2 FORMAT

Register Usage

Caution must be taken by the modifier of FORMAT scanner with respect to register usage. The following registers are used by TRANSIT, and must be preserved in FORMAT scanner:

	A0=mask base	X0=77------- 7700B
B1=1 B2=shift input	A1=input address	X1=input word X2=input character
B3=node address		X3=subroutine parameter
B4=return address B7=shift output	A7=output address	X7=output word

Caution must also be taken with respect to TRANSIT utilization of scratch registers. The following registers are used as scratch registers by TRANSIT:

	A3	
	A4	X4
B5		X5
	A6	

The return mechanism in all cases is via register B4. All intermediate processors save and restore B4 when it is utilized before a return.

LISTIO

1.0 General Information

Processes all forms of input/output statements which may occur in a FORTRAN program. These include READ, WRITE, PRINT, PUNCH, BACKSPACE, ENDFILE, REWIND, ENCODE, DECODE, BUFFER IN and BUFFER OUT.

FTN 4.0 produces an aplist structured type of calling sequence for all I/O statements. Each aplist is composed of a sequence of I/O macros (IOM's) which define file, format, and list item information to the actual I/O object time routines. The structure of the IOM is explained in section 6.

2.0 Entry Points

2.1 Code entry points

2.1.1 CNVT

Converts a binary number into a BCD string.

2.1.2 ENDFILE

Processes the ENDFILE statement.

2.1.3 REW

Processes the REWIND statement.

2.1.4 BKSP

Processes the BACKSPACE statement.

2.1.5 PUNCH

Processes the PUNCH statement.

2.1.6 PRINT

Processes the PRINT statement.

2.1.7 READ

Processes all form of the READ statement.

2.1.8 WRITE

Processes the WRITE statement forms.

2.1.9 BUFIN

Processes the BUFFER IN statement.

2.1.10 BUFOUT

Processes the BUFFER OUT statement.

2.1.11 DEC

Processes DECODE statements.

2.1.12 ENC

Processes ENCODE statements.

2.1.13 DOITX

Entry for return from DOPROC after processing the beginning of an implied loop.

2.1.14 DONEX

Entry for return from DOPROC after processing the end of a loop.

2.1.15 IARC

Processes input aplist restart call.

2.2 Non-code entry points

2.2.1 APLRST

Entry containing the store to I/O aplist flag.

2.2.2 BLEXP

Entry containing the binary list expression flag.

2.2.3 HOLCON

Entry containing hollerith constant information for processing hollerith constants in I/O lists.

2.2.4 Entry containing the indirect indicator flag.

2.2.5 IOEXP

Entry containing I/O expression flag.

2.2.6 IONAME

Entry containing header address for the I/O macro to be issued.

2.2.7 ITEMCT

Entry containing the item count for an aplist item entry word.

2.2.8 PARCNT

Entry containing the parameter count for each I/O list processed.

2.2.9 TYPEFG

Entry containing the variable or expression type of the I/O list item.

3.0 Diagnostics And Messages

CONFLICTING USE OF A NAME

BAD UNIT NUMBER

I/O STMT SYNTAX ERROR

FORMAT NUMBER SYNTAX ERROR

MISSING I/O LIST OR SPURIOUS COMMA

NON ANSI I/O STATEMENT

CHARACTER COUNT ERROR IN ENCODE/DECODE STATEMENT

PARITY NUMBER MUST BE 0 OR 1

FORMAT SPECIFICATION IS NON ANSI

UNIT NUMBER NOT BETWEEN 1 AND 99

DO CONTROL VARIABLE MUST BE A SIMPLE INTEGER

DO PARAMETER MUST BE AN INTEGER CONSTANT OR VARIABLE

ARRAY REFERENCE OUTSIDE DIMENSION BOUNDS

VARIABLE FOLLOWED BY (

ARRAY REFERENCED WITH FEWER SUBSCRIPTS THAN DECLARED

TOO MANY SUBSCRIPTS IN ARRAY REFERENCE

NO MATCHING RIGHT PARENTHESIS

4.0 Environment

All statement processors expect the statement to have been converted to E-list starting at the location contained in SELIST. A number of externals in DOPROC are referenced. Their functions are:

DOCALL	mark an external reference
DOCALL	mark an enternal reference
DODEF	mark a variable as defined
LABCON	convert a label to internal form
DOIT	process an implied DO loop
DOGOOF	compress the DO table after an I/O list error
DONE	terminate an implied I/O list
INTVAR	check for and enter an integer variable in the symbol table

5.0 Processing

5.1 IOSETUP

The setup routine is called prior to processing of each type of I/O statement. The I/O aplist number is incremented, and the aplist header line issued to the COMPS file. A USE DATA. is sent to the file ahead of the aplist header to ensure that the subsequent I/O aplist will be relocated in the correct block.

5.2 CNVT

Converts a binary member in X2 into BCD format, leaving the result in X7 upon exit. B1 contains an appropriate shift count when entered.

5.3 IXFNL

This is a local version of the IXFN routine in ARITH called to process each list item. Upon exit the registers are set up as if exited from a SYMBOL call.

5.4 CFSIV

Checks for a simple integer variable. Issues a diagnostic if the inputted variable is not type integer.

5.5 NAMLIST

Process NAMLIST I/O. Issues the group name to the COMPS file. Sends I/O macro for the call to the RLIST file.

5.6 PVARNAM

This routine is called to process variables used as file names, parity indicator names, format names, or character count names. It issues an appropriate IOM macro to the COMPS file for the variable processed. On entry, B4 contains a 0/1 flag indicator determining the variable usage, and X1, X2 contain the symbol table entry of the name.

Processing follows these steps:

a) Check to determine if the name is a formal parameter. If it is not, go to g.

b) Compute the formal parameter offset, and save it in the argument list for the IOM macro. If not a file name or parity indicator name, to to d.

c) Set the file bit for the IOM macro, and exit the routine.

d) Determine if an LCM variable is being processed, and if not, go to f.

e) Set the LCM bit for the IOM macro.

f) Set the variable bit for the IOM macro. Save any constant bias associated with the variable in the argument list for the IOM macro. Exit the routine.

g) Determine if the variable is equivalenced or not. Save the symbol table ordinal and any constant bias in the argument list for the IOM macro.

h) If a file name or parity indicator name is being processed, go to i. If the variable is not an LCM variable, go to j.

i) Set the file or LCM bit for the IOM macro. Exit the routine for the case of a file name or parity indicator name.

j) Set the variable bit for the IOM macro. Exit the routine.

5.7 FMTNO - Process Format Number

If the format item is a variable, a symbol call is made. In the not found case, the type and var bits are set in word B and a non ANSI flag set. Then processing joins with the found path. If the type is namelist, an exit is taken with X0 equal to zero. IXFN is called to process the name. Upon return the APLRST flag is tested to determine if store to I/O aplist code must be generated and calls are made to PSTAPL and STIOM if the flag is set. Otherwise PVARNAM is called to generate an IOM macro for the format name, and the macro is issued to the COMPS file before exiting FMTNO. A non-ANSI diagnostic is produced for non-dimensioned variable formats.

A constant format number is processed in a different manner. First, checks are made to ensure that:

a. The next item is a constant.

b. It is an integer constant.

c. It has no more than five digits.

If these conditions are met, LABCON is called to format and enter the label in the symbol table. On first occurrence, the following are done:

a. Set the type to label.

b. Set the referenced as format number (RFN) bit.

c. Generate an IOM for the format number.

d. Issue the IOM macro to the COMPS file.

e. Save the label ordinal in TEMP.

f. Collect references if necessary.

g. Reload B1 from TEMP to satisfy the exit condition.

h. Exit FMTNO.

For second and subsequent appearances, a check is made for the defined as statement number (DSN), referenced as statement number (RAS) and DO loop terminator (DLT) bits. If any of these are set, an error message is produced. Otherwise, processing begins with step a for the first appearance.

5.8 UNITN - Process The Unit Number or Parity Indicator

On entry, X7= zero if unit and one if parity. On exit, X3= zero if variable parity, or X1= binary number. If the next E-list item is not a name or constant, an error is issued. If the item after the name is not a right parenthesis or a comma, a diagnostic is produced. For a variable unit or parity indicator, CFSIN is called to validate the variable type. PVARNAM is called to generate an IOM macro for the name, and the macro is issued to the COMPS file. Exit is made through the entry point.

Constant unit or parity indicator must be integer or a diagnostic will be given. CONVERT is used to produce a binary integer from the constant value. At this point, we will exit the routine if we are converting a parity indicator. For a unit number greater than 99, an error is produced. Then the number is converted to display

code, an equivalence sign appended, and PLFN called to process the name. Upon return, exit is made from UNITN.

5.9 PLFN - Process Logical File Name

On entry, X1 contains 8R file name. On exit, , symbol 2 in the macro holds the symbol table ordinal of LFN and R number 2 contains an R number for the load. Initially, the logical file name is placed in the symbol table. On the not-found exit, file name bits are set into word B of the entry. The address of word B is saved, the ordinal placed into symbol two of the macro, and the next R-list number placed in the macro. If the SYSEDIT= FILES option was not selected, we simply collect an I/O reference if R=2 or 3 and exit. For the Files option in a main program, we simply exit after reference accumulation since the FET names will be local symbols. However, extra processing is needed for subrcutines. Remove all blanks and the special character from the file name and enter this into the constant table. Generate an IOM macro to the file name, and issue the macro to the COMPS file.

5.10 IOCM

This routine issues an I/O call macro to the RLIST file. On entry X1 contains the name of the execution routine to be called. The name is added to the symbol table, and the macro is built in the MACBUF created area, then written to the RLIST file.

5.11 ENDFILE

Set the object routine name to ENDFIL. and call PERB.

5.12 REW

Set the object routine name to REWINM. and call PERB.

5.13 BKSP

Set the object routine name to BACKSP. and call PERB.

5.14 PERB - Process ENDFILE, REWIND, BACKSPACE

First, the routine name is saved in TEMP and IOFLAG set one for a positioning reference. The macro op code is setup and a DOCALL is made to mark an external reference.

If something occurs after the unit number, an error is produced. A fake right parenthesis is added to the E-list to keep UNITN from producing a diagnostic UNITN is called to process the unit number and IOCM to issue the I/O macro to the R-list file.

5.15 PUNCH

Set the standard file name to PUNCH , call PROFL and exit.

5.16 PRINT

Set the standard file name to OUTPUT , call PROFL and exit.

5.17 PROFL

Processes I/O statements of the form keyword n, list. On entry, X1 holds 12/IOFLAG, 48/8R name of associated file. Initially, the value of IOFLAG is extracted and saved. MACOP= now sets the macro opcode to that for READ, WRITE, PRINT, PUNCH initial calls. Then, PLFN is called to process the logical file name. A non-ANSI usage is flagged and DOCALL is called to mark an external reference. Then FMTNO is called to process the format number. For namelist names, NAMLIST is called and then PROFL is exited. For standard format items, FMODE is used to set the file usage mode formatted. If the next E-list item is not a comma or an end of statement, an error is issued. Should the item after the comma be an EOS an informative error will result. Next, the name of the I/O routine is extracted from IOTAB using the value in IOFLAG. IOLIST is called to process the list and then exit is made from PROFL.

5.18 READ

If the first E-list element is not a left parenthesis, this is a read of the form READ n, list and PROFL is called with a file name of INPUTC . Otherwise, PRORW is called with an IOFLAG of 1S59. Upon return, processing is complete and an exit is taken.

5.19 WRITE

Set an IOFLAG if zero and call PRORW. Upon return, exit to the phase controller.

5.20 PRORW - Process READ and WRITE Statements

Save IOFLAG which is in X6 on entry and set the macro op to an initial call. Inform DO of an external reference. Generate an error if the next E-list item is not a left parenthesis. Clear all mode indications (LFNA) and call UNITN to process the unit number. If, upon return, the next item is a right parenthesis, set the file mode binary, adjust the macro opcode and set the mode flag (X3) to 0 (binary). If the item after the unit number is not a comma, an error is issued. Otherwise, FMTNO is called to process the format number. The item after the format number must be a right parenthesis or an error will be produced. If the format number field was a namelist group name, we go to NAMLIST for further processing. Otherwise, the mode is set to formatted and the mode flag (X3) to 2 for coded. Finally, we extract the name of the appropriate object routine and call IOLIST. Upon return, exit is made from PRORW.

5.21 BUFIN

Set the IOFLAG (X7) to input (1), call PBUF and exit.

5.22 BUFOUT

Set the IOFLAG (X7) to output (0), call PBUF and then exit.

5.23 PBUF - Process Buffer I/O Statements

Save IOFLAG and call DOCALL to mark an external reference. Issue an ANSI violation error and set the macro opcode to buffer I/O. Call UNITN to process the unit number and set the mode to buffer. If the next item is not a comma, issue an error. Otherwise, call UNITN to process the parity indicator. In the case of a constant parity value greater than 1 an error is diagnosed. An IOM macro is generated for the constant, and issued to the COMPS file. A check is made of the next E-list item. If not a right parenthesis, an error is issued. Next a left parenthesis must occur. IXFN is called to process the FWA name. If a store to I/O aplist is required, PSTAPL and STIOM are called to issue the appropriate macros to the RLIST file and an IOM macro to the COMPS file. Otherwise PVARNAM is called to generate the IOM macro for the name. For an input operation, the defined bit is set. The next item must be a comma. IXFN is

called again to get the last word address. Again either PSTAPL and STIOM or AVARNAM are called to complete processing of the LWA name. If the ending address is type double, an adjustment is made to the RLIST macros or the generated IOM to increment the address by 1. Next a right parenthesis must occur followed by an end-of-statement marker. Load the name of the object routine and call IOCM to produce the macro. Finally, processing exits from PBUF.

5.24 PSTAPL

This routine generates and issues to the RLIST file a sequence of macro to perform a store to an I/O aplist. On entry, X2 contains word B of the symbol table entry of the name being processed, $x is a zero/non-zero flag indicating whether the name represents a buffer I/O LWA, and X6 contains the result number returned by the IXFN call for the variable loaded.

PSTAPL builds in the MACBUF storage area a sequence of RLIST macros which will generate an I/O aplist entry word and store it into the desired area in an I/O aplist. These macros are the logical conclusion of the sequence begun when the IXFN call was made.

5.25 STIOM

Output to the COMPS file an IOM -1B to represent a position in the I/O aplist which will be the object of a store.

5.25.1 DEC - Process DECODE Statement

Set the IOFLAG to input (1) and call PED; exit upon return.

5.26 ENC - Process ENCODE Statement

Set the IOFLAG to output (0) and call PED; exit upon return.

5.27 PED - Process ENCODE/DECODE

Call DOCALL to mark an external reference and then issue a non-ANSI usage error. Set the macro opcode to ENCODE/DECODE. Advancing over the left parenthesis

(guaranteed to be there because of SCANNER'S algorithm), the character count field is examined.

For a constant character count:

a. Verify the constant to be integer.

b. Use CONVERT to get a binary value.

c. Issue an error if the character count is zero.

d. Issue an IOM macro to the COMPS file for the character count.

e. Verify that the next item is a comma.

f. Call FMTNO to process the format number.

g. Issue an error if the format item is a NAMELIST group name.

h. Verify that the next item is a comma.

i. Call IXFN with the complement of IOFLAG in X2.

j. Call PSTAPL and STIOM, or call PVARNAM to process the name of the target area.

k. Set the defined bit if this is an ENCODE statement.

l. Verify that the next item is a right parenthesis.

m. Load up the name of the the object time processor and call IOLIST to process the list.

n. Exit upon return.

For a variable character count:

a. Verify the next item to be a comma.

b. Use IXFN to obtain the address.

c. Call PVARNAM to process the character count name.

d. Issue the generated IOM for the character count to the COMPS file.

e. Verify that the count is a simple integer variable.

f. Join the processing for constant count at step e.

5.28 IOLIST

Processes the I/O list and outputs macros to R-list to call the execution time routines to transfer data to or from the input/output devices. On entry, A1 = address of two words containing the names of execution time routines and SELIST pointing to the first element of the list. Upon entry, if a binary write statement is being processed, the word count computation code is branched to. Otherwise the address of the macro header for a general external function call is placed in IONAME for later IOCM calls when issuing the macro to the RLIST file.

At the beginning of the main loop a check is made for a name. If a name is found, control branches to the name item processing. If not, the item is checked to be a left parenthesis, and control transfers to the DO processing code if it is. When these two tests fail and the item is not an end of statement marker, it can be assumed that an expression or constant item is being processed, and control transfers down the path of the name item. Otherwise, for an EOS item, an end-of-I/O macro (EIO) is issued to the COMPS file to terminate the I/O aplist, the DATA. block is incremented, and IOCM is called to issue the I/O call macro to the RLIST file.

5.28.1 NAME item

If the item following the name is an equal sign, go to DOEND to close out the loop. Set the value of SERF (it will be zero if the next element is a left parenthesis). Then call IXFN to process the address and save the R number of the result. For an input operation, the defined bit is set at this time and DODEF is called to inform of the redefinition of a variable. Next, we compute the value of the single/double precision flag (0 if single, 1 if double). A series of checks are performed to determine whether the list item processed by the IXFN call falls into one of three classes: 1) an array item requiring a store to I/O aplist, 2) an expression, or 3) a hollerith constant. The processors for each of these classes of list items will be explained shortly.

If none of these conditions were satisfied, and the next item was a left parenthesis, the item count flag (ITEMCT) is set to 1, and LSTITM is called to process the aplist item. However, if the next item was a left parenthesis and the name is dimensioned, the DIM table entry (word two) is examined. Special processing for non-variable dimensions will appear in an upcoming section.

After issuing the aplist, we go back to the main loop if the next item is a comma. For an end of statement, we go to issue a final call. If the item is neither of these, and it is not an IOLIST right parenthesis, an error condition exists.

5.28.2 Variable Dimension Array Transfers

First, a determination is made to see if any constant dimensions are present. Consider the symbol CONF to be 1 if constant dimensions or a double/complex array occurs, else zero. Next compute the number of variable dimensions plus one divided by two + CONF + 1. This yields the number of words in the body of the macro. The macro number is computed from the base number minus one + number of variable dimensions + 3 times CONF. Combining these the macro header word is formed. For all three dimensions variable, the IH fields are extracted from the DIM word, and the macro constructed. For a mixture of constant and variable dimensions, the product of the constant portions with the word count for the item (1 or 2) is computed. In addition, symbol table words are constructed for placement in the macro. Next, we provide an R number for the resister store in the macro and write the macro to the R-list file. Then setup and issue a variable word count intermediate call macro instead of the ordinary intermediate call macro.

5.28.3 Store to I/O Aplist List Item

This code generates in the MACBUF storage area an RLIST macro to combine the variable type information along with the variable address obtained by IXFN. PSTAPL is called then to generate the actual store macro, thereby completing the code sequence initiated by the IXFN call. Finally an IOM macro is sent to the COMPS file indicating that the aplist item will be provided at execution time.

5.28.4 Expression List Item

When IXFN processes a list expression, the final result is stored into an ST.. This code simply generates an IOM macro for the ST. entry. An ANSI diagnotic is issued to flag the occurrence of an expression in an I/O list.

5.28.5 Hollerith Constant List Item

This routine converts the character count for the Hollerith constant inot a word count, and issues an IOM macro to the COMPS file defining the position of the constant in the HOL. block.

5.28.6 Standard Aplist List Item

This is the analog of the PVARNAM routine, but processes only list items found in the I/O statement. It generates an IOM for the list item, and issues it to the COMPS file. The processing is similar to that in PVARNAM, and therefore will not be described in any detail here.

5.28.7 Implied DO Loop Processing

Code to process implied DO loops will attempt to collapse statemnts of the form:

(((A(I,J,K),I=I1,I2,I3),J=J1,J2,J3),K=K1,K2,K3)

If more than three levels of parentheses occur before a name is found, the loop is non-collapsible and control passes to code for this type of loop. The number of parentheses is saved in COLLAPS. The current line number and the E-list address of the array name are compared with the contents of NOCAL (indicator of the last name address and line number found to be non-collapsible). If a match is found, processing goes directly to the non-collapse code.

At the start of collapse processing, a number of cells are cleared. The cells and functions are:

NAMDEX (3 words) - E-list for I,J,K

INDX (12 words) I,I1,I3,J,J1,J2,J3,K,K1,K2,K3

ARNAM (1 word) array name (OR format)

If the item after the name is not a left parenthesis, mark the loop non-collapsible and process accordingly. A

SYMBOL call is made to enter the name in the symbol table and an error will be produced if a first appearance return is taken since this implies the item was never given a dimension. For the found return, a check is performed to insure the array is dimensioned. Using the type field, a single/double precision flag is computed (SDPF = 0 for single, 1 for double word items). The flag is saved in the entry point word of REW since this will be a safe temporary during IOLIST processing. If the array is double, the bogus CONLIST entry is changed to 2.

The number of dimensions are extracted from the DIM word and the word saved in DIMWRD and DIMVAL. NODIMS is set to contain the number of dimensions. Next, the subscripts of the array are scanned. If the item is a name, it is placed in NAMDEX. For a constant this is omitted. If the item is not a name or a constant and it is not a right parenthesis, the loop is non-collapsible. The only acceptable thing for the next element is a comma or a right parenthesis. If the number of dimensions referenced exceeds three, an error is produced and flow returns to the phase controller. If the argument count is still proper, processing goes back to get the next subscript and repeats the previous steps.

When the right parenthesis is encountered, a check is made to verify that at least one subscript appeared. The word count of one or two is installed in the first macro constant parameter. Should the item after the parenthesis not be a comma collapse is abandoned. Similarly, after the comma a name must appear. The name of the induction variable is placed in INDX. Next, an equal sign must occur for collapse to continue.

The next portion of the list contains I1, I2, I3, and processing continues:

a. Extract an item.

b. Issue an error if it is not a name or a constant.

c. Save the name or constant in INDX area.

d. Loop back to a if the next item is a comma and there are three or less indexes so far. If the index count reaches four, issue an error.

e. When the right parenthesis is encountered, reduce the parentheses level.

f. Go back to processing for the next loop control variable until the parentheses level is satisfied.

At this point, the formal collapse processing begins. Processing proceeds as follows:

a. If the subscript name field (I) is a constant, exit to mark no further collapse.

b. DOVAR is called to ensure that the subscript is a legal integer variable.

c. Set the defined bit in word two of the symbol table.

d. Compare the dimension with the index. If they do not match, terminate collapse processing.

e. If the increment (I3) is variable or not a constant one, collapse is terminated.

f. VALTYP is called to validate the do increment value for a constant and returns the value in X1.

g. If R=2 or 3, two references are collected for the control variable.

h. Next a check is made to see if I1 is a variable or a constant.

   For a constant:

   (1) Use VALTYP to validate it.

   (2) Save the constant in the macro.

   (3) For I1=1, full collapse is still possible. If not, set this as the final collape level (TENCOL).

   For a variable:

   (1) Inhibit collapse for double word arrays.

   (2) Mark this the last collapse level.

(3) Change the macro op to variable.

(4) Use DOVAR to validate I1.

(5) Call EQUIVP to place the correct base and bias into the macro.

i. Load up dimension information and save the constant dimension for this time. Shift the contents right 18 bits to set up for the next iteration.

j. If the dimension is variable, perform special handling.

k. Check I2 for variable or constant for I2 constant.

(1) VALTYP is used to convert and validate the type.

(2) For I1 variable, set macro constant three to the constant value of I2 plus one and go try to collapse further levels.

(3) For I1 constant compute (I2-I1) + 1 * previous MACLK1 and place the result in MACLK1.

(4) For a zero or negative word count, set the count to the value of the SDPF + 1.

(5) Clear I1 from macro parameter three.

(6) If the constant value of I2 does not match that for the dimensionality, inhibit further collapse and produce an informative message if it exceeds the declared bound.

For I2 variable:

(1) Suppress collapse on double arrays only if I1 is not one.

(2) Reduce constant three in the macro (I1) by 1 if MACOPC is still zero (I1 not variable). If collapse has not been terminated bump MACOPC by three to get a c*v macro.

(3) Now bump MACOPC by one and mark no further collapse.

(4) Call DOVAR to validate I2.

(5) Use EQUIVP to place base-bias in the macro.

(6) Collect a reference to I2 if necessary.

l. Try to collapse remaining levels (COLAPR8) Place the current E-list address and statement number in NOCAL.

m. Check to see if maximum collapse level has been reached. If so, go issue macros.

n. If collapse inhibit is marked (TENCOL = -1), process this as a normal I/O list loop.

o. Bump the collapse level and restart at a.

This level is not collapsible (COLAPR9). If it is level one, abandon all collapse. Set the flag for no further collapse. Reduce the current level by one and go to step 1 above.

Issue macro code for the collapsed list:

a. Compute the macro number using MACOPC + base of collapsed I/O macros. (Put this in MACOP).

b. If the macro has a multiplier of one, change it to a macro to omit the multiply.

c. Allocate an R number for the macro.

For LWA +1-FWA type macro the following occurs:

(1) Generate E-list for the subscripted array reference used to denote the last word address to be used.

(2) Use IXFN to compute the address and save it in R number two of the macro.

(3) Replace I1 where it belongs.

d. Call ARYCONS to produce an array reference for the index function that will be short listed because of the collapse.

e. Produce necessary index function code using IXFN and save the result register in the macro.

f. Call MACOUT to issue the macro code.

g. Restore the E-list pointer and return to process the next E-list item.

Variable dimension collapse handling:

a. Force the macro to a LWA+1-FWA type.

b. For I2 a constant:

(1) Place the symbol name, right justified in TEMP.

(2) Inhibit further collapse.

(3) Go to normal I2 constant handling.

For I2 a variable:

(1) Inhibit collapse if this is a double word array.

(2) If the dimension subscript is different from I2, no further collapse is possible.

(3) Call DOVAR to validate I2.

(4) Use EQUIVP to obtain base-bias in the macro.

(5) Collect a reference if necessary.

c. Proceed to try a collapse of remaining levels.

ARYCONS

This routine modifies the array reference so that corresponding subscripts for collapsible levels reflect the initial value of the item. For example, A (I) becomes A (I1) and A(I,J) becomes A(I1,J1) provided both the I and J levels are cllapsible.

5.28.8 Non-Collapsible I/O DO Loops

a. Scan the body of the I/O loop for an equal sign at level zero and a right parenthesis at level -1.

b. If no equal sign was found, this must be something of the form (var, var, var) and can be processed as simple elements.

c. For an I/O loop, the final parenthesis is changed to a special right parenthesis, and we go to DOBEGIN for initial loop processing.

d. DOIT returns to DONEX and the E-list pointer is advanced to the special right parenthesis, SELIST updated and control passed to process the next I/O element.

5.29 DOBEGIN

Issues an I/O call, if necessary, before generation of the DO-begin code by the DO processor. ARIOCM is called to issue the restart call, IONAME is adjusted to reflect a change of routines being called, and finally DOIT is called for the loop code generation.

5.30 DOEND

Issues an I/O call, if necessary, before generation of the DO-end code by the DO processor. ARIOCM is called to issue the restart call, and DONE is called for the loop code generation.

5.31 ARIOCM

Issues a soft end of I/O list macro to the COMPS file, increments the DATA. block size, calls IOCM to issue the I/O call, and calls IOSETUP to generate a new aplist header label for subsequent processing.

5.32 IARC

Issues a restart call when a list item is used as an array item subscript later in that same list. Called from ARITH.

5.33 MACOUT - Output I/O Macro To R-list

This routine produces a macro with four symbols, six R numbers and 3 constants. The parameters are obtained from the area from MACLS1 through MACLK3 and packed up. Then the area from MACL52 to MACLK3 is cleared and the macro dumped to R-list.

6.0 Structures

6.1 Aplist Parameter Element Expansions

6.1.1 Unit name/number pointer

```
VFD     1/VAR,1/FP,40/0,18/FITADR
```

6.1.2 Format pointer

```
VFD     1/LCM,1/FP,1/VAR,33/0,24/FMT
```

6.1.3 Mode pointer

```
VFD     42/0,18/MODEWD
```

6.1.4 Buffer I/O FWA

```
VFD     1/LCM,1/FP,1/VAR,33/0,24/FWA
```

6.1.5 Buffer I/O LWA

```
VFD     1/LCM,1/FP,1/VAR,33/0,24/LWA
```

6.1.6 String pointer (encode/decode)

```
VFD     1/LCM,1/FP,1/VAR,33/0,24/DATSTR
```

6.1.7 Count pointer (encode/decode)

```
VFD     1/LCM,1/FP,1/VAR,33/0,24/CNT
```

6.1.8 List pointer

```
VFD     1/LCM,1/FP,1/IND,3/0,6/TYPE,18/NBREL,6/0,
        24/ITEM
```

6.1.9 Record length

```
VFD     42/0,18/WDCNT
```

6.1.10 End of aplist

```
VFD     60/END
```

where the above terms are defined as:

VAR - denotes the item is a variable

FD - denotes the item is a formal parameter

LCM - denotes the item is large core resident

TYPE - denotes the item type

```
          0 - reserved
          1 - logical
          2 - integer
          3 - real
          4 - double
          5 - complex
  6 - 63 - reserved
```

IND - denotes indirect item reference

NBREL - denotes the number of contiguous elements in the list item if IND=0

NBREL - denotes the SCM address of the list element count if IND=1

END - denotes the end of an I/O list if +0

END - denotes an intermediate interruption in an I/O list if -0

If the FP bit is set, the address field is interpreted as

```
18/BIAS,6/FPORD,
```

where FPORD denotes the ordinal of the formal parameter in the parameter list for the subprogram, and BIAS is any offset associated with that particular formal parameter reference.

3.2 IOM Definition

The IO, macro defines the element expansions listed above. The macro call is of the form

```
IOM    BASE,BIAS,TYPE,COUNT,B59,B57,BASE2,
```

where

BASE - aplist item base address
BIAS - aplist item bias if BASE is present
     - formal parameter ordinal if BASE is nul

TYPE - aplist item count
COUNT - aplist item contiguous element count if
B 57 is nul
- element count offset if B57 is present
B59 - LCM/file name bit
B57 - variable/indirect bit
BASE2 - formal parameter offset if B57 nul and a
formal parameter has been determined
- base field for item element count if B57 present

ARITH

1.0 General Information

Task Description

The function of the ARITH statement processor is to translate E-list for an arithmetic replacement into R-list and issue appropriate macros to the R-list file. It also translates any arithmetic, logical, relational, or masking expressions which may legally appear in any type of statement. ARITH calls an external routine to process arithmetic statement functions, and then translates the expanded statement function.

2.0 Entry Points

IDORDL	contains the symbol table ordinal of an ID name
NAMFWA	contains the address of word A of a symbol table entry for a name
DBGAPL	debug aplist table used by the debug processors to format aplist information for debug calling sequences
APLRT	code block called to format and issue an aplist instruction to the ARLIST buffer
GEFCM	code block called to format and issue a general external function macro to the ARLIST buffer
DARLIST	code block called to output the ARLIST buffer to the R-list file
CVDB	code block called to issue R-list macros to compute the total bound of a variably dimensioned array
STRIP	code block called to remove a trailing dollar sign from a name
FSTRIP	FORTRAN entry point for the STRIP routine

IXFN code block called to process an item in an I/O list

ACALL code block called to process the argument list in a subprogram CALL statement

ARITH primary entry point of the arithmetic statement processor

INITR code block called to initialize memory cells in ARITH

IFE code block called to process arithmetic IF statements

IFL code block called to process logical IF statements

OPSTACK operator stack for ARITH contains, information at DPCLOSE time required for address substitution of the ARLIST buffer

3.0 Diagnostics Produced

3.1 Fatal to compilation: none

3.2 Fatal to execution

A CONSTANT ARITHMETIC OPERATION WILL GIVE AN INDEFINITE OR OUT-OF-RANGE RESULT.

EXPRESSION TRANSLATOR TABLE (JAMTB1) OVERFLOWED. SIMPLIFY THE EXPRESSION.

TYPE ECS NOT AVAILABLE IN THIS VERSION OF FTNX.

ILLEGAL USE OF THE EQUAL SIGN.

VARIABLE FOLLOWED BY LEFT PARENTHESIS.

NO MATCHING RIGHT PARENTHESIS.

NO MATCHING LEFT PARENTHESIS.

THE OPERATOR INDICATED (-, +, *. /, or **) MUST BE FOLLOWED BY A CONSTANT, NAME, OR LEFT PARENTHESIS.

A NAME MAY NOT BE FOLLOWED BY A CONSTANT.

MORE THAN 63 ARGUMENTS IN ARGUMENT LIST.

A CONSTANT MAY NOT BE FOLLOWED BY AN EQUAL SIGN, NAME, OR ANOTHER CONSTANT.

EXPRESSION TRANSLATOR TABLE (OPSTAX) OVERFLOWED. SIMPLIFY THE EXPRESSION.

LOGICAL OPERAND USED WITH NON-LOGICAL OPERATORS.

NO MATCHING RIGHT PARENTHESIS IN SUBSCRIPT.

LOCAL ENTRY POINT REFERRED TO AS EXTERNAL FUNCTION.

INTRINSIC FUNCTION REFERENCED MAY NOT USE A FUNCTION NAME AS AN ARGUMENT.

ARGUMENT NOT FOLLOWED BY COMMA OR RIGHT PARENTHESIS.

A FUNCTION REFERENCE REQUIRES AN ARGUMENT LIST.

ILLEGAL CALL FORMAT.

EXPRESSION TRANSLATOR TABLE (FRSTB) OVERFLOWED. SIMPLIFY THE EXPRESSION.

THE OPERATOR INDICATED (.NOT. OR A RELATIONAL) MUST BE FOLLOWED BY A CONSTANT, NAME, LEFT PAREN, -, or +.

BASIC INTRINSIC FUNCTIONS WITH AN INCORRECT ARGUMENT COUNT.

EXPRESSION TRANSLATOR TABLE (ARLIST) OVERFLOWED,. SIMPLIFY THE EXPRESSION.

ILLEGAL INPUT/OUTPUT ADDRESS.

RIGHT PARENTHESIS FOLLOWED BY A NAME, CONSTANT, OR LEFT PARENTHESIS.

MORE THAN ONE RELATIONAL OPERATOR IN A RELATIONAL EXPRESSION.

A COMMA, LEFT PAREN, =, .OR., OR .AND. MUST BE FOLLOWED BY A NAME, CONSTANT, LEFT PAREN, -, .NOT., OR +.

AN ARRAY REFERENCE HAS TOO MANY SUBSCRIPTS.

NO MATCHING RIGHT PARENTHESIS IN ARGUMENT LIST.

ILLEGAL FORM INVOLVING THE USE OF A COMMA.

LOGICAL AND NON-LOGICAL OPERANDS MAY NOT BE MIXED.

DIVISION BY CONSTANT ZERO.

A COMPLEX BASE MAY ONLY BE RAISED TO AN INTEGER POWER.

USE OF THIS SUBROUTINE NAME IN AN EXPRESSION.

SUBROUTINE NAME REFERRED TO BY CALL IS USED ELSEWHERE AS A NON-SUBROUTINE NAME.

TOO MANY SUBSCRIPTS IN ARRAY REFERENCE.

LEFT SIDE OF REPLACEMENT STATEMENT IS ILLEGAL.

THE TYPE OF THIS IDENTIFIER IS NOT LEGAL FOR ANY EXPRESSION.

A CONSTANT OPERAND OF A REAL OPERATION IS OUT OF RANGE OR INDEFINITE.

THIS COMBINATION OF OPERAND TYPES IS NOT ALLOWED IN THIS VERSION.

DOUBLE OR COMPLEX OPERAND IN SUBSCRIPT EXPRESSION NOT ALLOWED.

DOUBLE OR COMPLEX ARGUMENT NOT LEGAL FOR THIS INTRINSIC FUNCTION.

.NOT. MAY NOT BE PRECEDED BY NAME, CONSTANT, OR RIGHT PARENS.

3.3 Informative

ARRAY NAME OPERAND NOT SUBSCRIPTED. FIRST ELEMENT WILL BE USED.

THE NUMBER OF ARGUMENTS IN THE ARGUMENT LIST OF A NON-BASIC EXTERNAL FUNCTION IS INCONSISTENT.

THE NUMBER OF ARGUMENTS IN A SUBROUTINE ARGUMENT LIST IS INCONSISTENT.

A HOLLERITH CONSTANT IS AN OPERAND OF AN ARITHMETIC OPERATOR.

3.4 Non-ANSI

MORE THAN ONE EQUAL SIGN.

ARRAY NAME REFERENCED WITH FEWER SUBSCRIPTS THAN THE DIMENSIONALITY OF THE ARRAY.

HOLLERITH CONSTANT APPEARS OTHER THAN IN AN ARGUMENT LIST OF A CALL STATEMENT OR IN A DATA STATEMENT.

NON-ANSI SUBSCRIPT.

MASKING EXPRESSIONS ARE NON-ANSI.

THE TYPE COMBINATION OF THE OPERANDS OF AN EXPONENTIAL OPERATOR IS NOT ANSI.

A RELATIONAL HAS A COMPLEX OPERAND.

THE TYPE COMBINATION OF THE OPERANDS OF A RELATIONAL OR AN ARITHMETIC OPERATOR (OTHER THAN **) IS NOT ANSI.

4.0 Environment

4.1 Low core cells

SYM1	(12B)	starting address of symbol table
DIM1	(17B)	starting address of dimension information table
TYPE	(24B)	type code of current statement. (Different statement types have different legal syntax at the end of expressions)
SELIST	(32B)	address of next E-list element
CDCNT	(37B)	line number of first card of current statement

NGLN	(52B)	next available generated label number
NRLN	(64B)	next available result number
		SELIST and NRLN are also referred to as EPOINT and NARN.

4.2 Common blocks

4.2.1 /NAALN/

NAALN next available APLIST number

4.2.2 /STSORD/

STSORD next available statement temporary store number (Reset to 1 by PH2CTL at the start of each statement)

4.2.3 /CLNFO/ (Used only by ARITH and CALL)

SUBFWA address of the first word of the symbol table entry for the name of the subroutine being called

SUBH symbol table ordinal of the subroutine name

ARGCNT number of arguments in paratmeter list

NARGSF argument list flag - equals 0 if there is an argument list

SUBNAME name in E-list format of subroutine being called

ARLPT ARLIST buffer pointer - number of words in buffer for current statement

4.2.4 //

DEBUG base address for referencing debug tables

4.3 Externals

ADDREF code block in PS1CTL called to note a reference for a variable, array, or function name

ALLARR debug cell in DBGPHCT used to indicate whether subscript references for all arrays are to be checked unconditionally

ALLFUNC debug cell in DBGPHCT used to indicate whether function references are to be traced unconditionally

ASAER code block in ERPRO called to issue a non-ANSI usage diagnostic

BEFTB entry point in ENDPRO indicating the beginning of the basic external function table

BKSP code block in PRINT called to process a BACKSPACE statement

BUFIN code block in PRINT called to process a BUFFERIN statement

BUFOUT code block in PRINT called to process a BUFFEROUT statement

CBNFLG cell in FTN used to indicate whether the trace option has been selected

CFO code block in DBGPHCT called to check debug usage of variable names with actual program usage

CONVERT code block called to place a constant in the CON. table

CON. cell in LSTPRO containing the symbol table ordinal for CON.

DEC code block in PRINT called to process a DECODE statement

DFLAG debug cell in FTN used to indicate whether the debug option has been selected

DOCALL code block in DOPROC called to inform it that an external reference has occurred

DODEF code block in DOPROC called when an integer variable appears as the object of a replacement statement

DOFLAG	cell in PS1CTL containing a DO loop nesting level count
DOGOOF	code block in DOPROC called after encountering a fatal error while processing the list of an implied DO
DOLABR	code block in DOPROC called to inform it of a reference to a statement label
DOSYM	code block in DOPROC called when an integer variable appears as an operand
D.SAASI	cell in DBGPHCT containing the base address of the arrays and stores information table
ENC	code block in PRINT called to process an ENCODE statement
ENDFILE	code block in PRINT called to process an ENDFILE statement
ERPRO	code block called to issue fatal error diagnostics
ERPROI	code block in ERPRO called to issue informative diagnostics
FP.	cell in LSTPRO containing the symbol table ordinal for FP.
FSTEX	cell in LSTPRO used to indicate when the first executable statement has been reached
GOTO	code block called to process a GOTO statement
IGCALL	code block in CALL called to form an issue R-list for a subroutine call
IPH2	code block is PS1CTL called to initialize phase 2 process of pass 1
LABEL.	cell in LSTPRO containing the symbol table ordinal for LABEL.
L.BEFTB	cell in ENDPRO containing the length of the basic external function table

L.CON cell in LSTPRO containing the length of the constant table

N.EQUAL cell in SCANNER containing the equal sign count for the current statement being processed

N.FP cell in LSTPRO containing the number of formal parameters in an argument list

OPTLVL cell in FTN containing the level of code optimization selected

O.CON cell in LSTPRO containing the starting address of the constant table

O.GCON cell in LSTPRO containing the starting address of the global constant table used in DEBUG mode

PAUSEP code block in STMTP called to process a PAUSE statement

PH2RETN code block in PS1CTL returned to after a fatal error diagnostic has been issued

PRINT code block in PRINT called to process a PRINT statement

PUNCH code block in PRINT called to process a PUNCH statement

READ code block in PRINT called to process a READ statement

RETURN code block in ENDPRO called to process a RETURN statement

REW code block in PRINT called to process a REWIND statement

ROPFLAG cell in FTN used to indicate whether the round option has been selected

RSELECT cell in FTN used to indicate whether either of the long reference map options has been selected

STOPP code block in STMTP called to process a STOP statement

ST. cell in LSTPRO containing the symbol table ordinal for ST.

SYMBOL code block in LSTPRO called to make a new entry into or search for an existing entry in the symbol table

TRACEL debug cell in DBGPHCT used to hold the level number for the TRACE debug statement

VALUE. cell in LSTPRO containing the symbol table ordinal for VALUE.

WRITE code block in PRINT called to process a WRITE statement

WRWDS code block in FTN called to perform the writing of R-list macros to the R-list file

5.0 Subroutines used by ARITH

5.1 External Routines

5.1.1 WRWDS

Used to make entries to the R-list file.

5.1.2 SYMBOL

SYMTAB search and entry routine.

5.1.3 CONVERT

Constant conversion and CONLIST entry routine.

5.1.4 ASFREF

Called as each statement function reference is encountered to insert the statement function with actual arguments replacing dummy arguments into the E-list block.

5.1.5 DODEF

Called to inform DOPROC of the definition of a variable.

5.1.6 DOCALL

Called to inform DOPROC of an external function reference.

5.1.7 DOSYM

Called to inform DOPROC of a reference to a variable.

5.2 Local Routines

5.2.1 FUNC5RT

This routine is called when a function reference (other than a statement function) is encountered. The reference might occur in an argument list, so a block of cells (FRLW), used to hold information about argument lists is entered into the OPSTAK followed by the ARGLP operator (which will be popped by the right paren which terminates the list) and an ARGCMA operator (which will be popped by the comma after the first argument or the right paren if only one argument). The FRLW block is initialized. DOCALL is called if it is an external function. If the result of a previously referenced external function has not been saved, an instruction is output to R-list to save the result.

The routine is called by the main line processor and by the exponential operator processor.

5.2.2 CARGPORT

This routine is called by the main line processor and the exponential operator processor. It is called after each argument of a non Statement Function argument list has been scanned (it may be an expression). Intrinsic, basic external, and general external arguments are each processed differently. Intrinsic arguments cause the R name of the argument to be added to the R name table (RNTB), basic external arguments cause register-store instructions to be output to ARLIST (which cause particular X-registers to be associated with the arguments), and general external arguments cause a store to APLIST or assembler to APLIST instruction to be sent to ARLIST.

5.2.3 ARGPIRT

This routine is called by the main line processor and the exponential operator processor after the argument list

has been processed. If the function is general external, it outputs a call by name macro to ARLIST. It then enters ARGP8CR to output loads of functions saved during the processing of the list, if any, to R-list. Then register define instructions are output to ARLIST, giving R-names to the result register(s), X6 (and X7). Then all of the ARLIST for this function reference is output to R-list. The next available location in the ARLIST buffer is adjusted. A psuedo-op giving the name of the function result is then output to ARLIST. Finally, the FRLW block is restored to the values it contained before this function reference. If the function was intrinsic, the contents of RNTB would be used to set up the parameters of the corresponding macro and the macro would be sent to ARLIST. Finally, the FRLW block would be restored.

If the function was basic external, a call by value macro would be sent to ARLIST and ARGP8CR would be entered, as for general externals.

5.2.4 INGEN

Processes binary operations. The input is the address of the operands, and the macro code of the operator. If both operands are real or integer constants and the operator is +, -, or /, then the operation is made on the constants, the instructions which loaded the constants are no-oped and a macro is formed and output to ARLIST to operate on the operands and the operand entries in ARLIST are marked as having been used. Finally, the cells holding the addresses of the last two available operands (RL1 and RL2) are reset.

5.2.5 UINGEN

Processes binary operations, similar to INGEN.

5.2.6 MACOUT

Routine to make entries to ARLIST. The input is: type of result (e.g., Double Precision), the macro descriptor (macro number, number of Rs, IHs, CAs), NARN (next available R name), and the macro parameters in a block called PARAMS. MACOUT forms the ARLIST information word and the macro in the next available locations in ARLIST.

5.2.7 MODCH

Used to generate a macro to convert from one data mode to another. The input is the address of the operand in ARLIST which is to be converted, and the data type to which it is to be converted. The type code of the operand and the new type code are combined to form a vector. The vector table is entered: the correct convert-macro code is selected and one of two possible branches are jumped to. A macro is then output to ARLIST.

6.0 Formats

6.1 ARLIST entries: (ARLIST is the block that ARITH forms R-list subexpressions in).

Word 1:

```
VFD  1/NOP,11/Type of result,1/C,1/GPTU,1/J,1/XMT,
     7/0,1/J,18/# words in this entry,
     18/# words in preceding entry
```

B59 = 1 if this entry is not to be sent to R-list. If B59 -1, the entire contents of word 1 have been complemented.

B58-48 indicate the type of operand as follows:

2000B	Logical
2001B	Integer
2002B	Real
2003B	Double
2004B	Complex
2005B	Octal
2006B	Hollerith

B47 = 1 if the operand is a constant.

B46 = 1 if this entry is temporarily unavailable as an operand.

B45 = 1 if this entry has been used as an operand to a subsequent operation.

B44 = 1 if a transmit instruction should follow this entry if it is the second operand of an equal-sign operator. (e.g., see the R-list macro definition of the intrinsic function REAL).

B36 = 1 if this entry is a replacement "fetch". (Used by JAM8 only).

Word 2: Word 2 is unused at this time. It was initially planned to use this word to further optimize evaluation of logical expressions, but it was found that the optimizations could not be made because of a basic design peculiarity.

Word 3: The first word of the R-list macro (or instruction). If the entry may be used as an operand, B15-0 of this word holds the R-name of the operand. (If a double length operand, R+1 is the name of the second word of the operand).

6.2 OPSTAK

This is the operator stack block. Generally, there is one word per operator. The format of that word is given below:

```
VFD 12/Operator code,4/statement function type,21/0,
    1/CGP,1/S,1/A,1/E,1/GP,18/Operator precedence
```

B59-48 = the operator code. The lowest operator code is 2003B. The codes used to represent source operators are also used by ARITH although ARITH generates some of its own operators.

B47-44 are used with code 2036B to indicate the type of statement function referred to (0 = logical, 1 = integer, etc.).

B22 = 1 if B18 = 1 and this operator has been compared with one in the stack with equal precedence.

B21-19 are used with codes 2006B, 2026B, and 2036B. (These are operators which represent different types of left parens). B21-19 are used to remember whether a subscript, argument, or normal expression was being translated before the left paren occured; this is indicated by B21-19=4, 2 or 1 respectively. The information is needed to know whether a comma is a subscript, argument, or complex constant comma.

B18 = 1 if the operator has been compared with one of higher precedence.

B17 - 0 the precedence of the operator.

Operators	Code in Octal	Precedence	
)	2002	0	
,	2003	0	
E.O.S.	2004	0	(end-of-statement)
=	2005	0	
(	2006	0	
.OR.	2007	2	
.AND.	2010	3	
.NOT.	2011	4	
.LE.	2012	5	
.LT.	2013	5	
.GE.	2014	5	
.GT.	2015	5	
.NE.	2016	5	
.EQ.	2017	5	
-	2020	6	
+	2021	6	
*	2022	7	
/	2023	8	
**	2024	10	
(A	2025	0	(left paren preceding function argument list see 2036)
(S	2026	0	(left paren preceding non-standard subscript)
,S1	2027	1	(comma following first subscript expression in non-standard subscript)
,S2	2030	1	(comma following second subscript expression in non-standard subscript)
,A	2031	1	(comma separating arguments)
U-	2032	6	(unary minus)
R-	2033	6	(reverse-operand minus)
R/	2034	8	(reverse-operand divide)
*	2035	9	(special multiply, e.g., A/B/C/D A/(B*C*D)
(S.F.	2036	0	(left paren preceding Statement-Function argument list)
(X	2037	0	(generated left paren entered at start of IXFN)
(SUBR	2040	0	(left parens preceding CALL argument list)

6.3 FRSTB: Function results saved table. Information about functions which have been saved. One word per entry.

B58 = 1 if the function was Double or Complex.

B33 - 16 = the number of the statement-temporary-storage location in which the function result was saved.

B15 - 0 = the R name of the function result.

6.4 XPNMT

Exponent function name table. This table gives the name of each library function corresponding to the various combinations of operands possible for the ** operator. Each entry is one word. The format is:

B59-56 = type of result of the operation (1 = integer, 2= real, etc.)

B-55-49 (unused).

B-48 = 1 if the combination if non-ANSI.

B47-0 = the name of the function.

There are 16 entries for the 16 possible combinations. Entries for illegal combinations are all zero. To make an illegal combination legal, replace the entry with the necessary information as described above.

6.5 INTFTB

Intrinsic function table. Three words per entry, one entry per intrinsic function. The format of the first and second words is the same as the pass-1 format of SYMTAB entries with the exception that B0 of the second word = 1 if the function contains an RNM R-list instruction and therefore may need a transmit before a store. The third word holds the macro descriptor word (see MACOUT), or, if MAX or MIN type functions, special information about the type of MAX or MIN function.

6.6 BEFTB

Basic external function table. Two words per entry, one entry per function. The format is the same as the first pass format of SYMTAB entries.

7.0 Modification Facilities

EQUs are used for diagnostic ordinals, MACROX ordinals, lower memory cell locations, block sizes, codes, etc. Diagnostic macros are used. All explicit operations and intrinsic function references result in R-list macros rather than separate R-list instructions.

8.0 Method

8.1 There are four kinds of expressions in FORTRAN Extended:

1. Arithmetic
2. Relational
3. Logical
4. Masking

The same translator is used to translate all kinds of expressions. Translation takes place in a single left to right scan of the expression.

Translation is from the E-list form of the source statement to R-list language. The R-list language specifies the machine instructions and registers to evaluate the expression, but the registers are assigned as if there were an infinite number available. The second pass assigns actual registers to the instructions.

8.2 Arith is called by:

1. Phase-2 control for processing of replacement statements
2. Computed GO TO processor
3. IF
4. CALL

These are the only kinds of statements which may contain expressions. If the replacement statement is actually a statement function definition, ARITH will call ASFDEF to save the statement function for later reference as a macro. Statement functions are expanded in-line at each point of reference.

8.2.1 Computed GO TO: ARITH translates the expression, converts to type integer if necessary, outputs the R-list block, and returns to the GOTO processor with the number + 1 of the result-R in a common location.

8.2.2 IF: ARITH translates the expression, outputs the R-list to the R-list block, and returns to the caller with the name and type of the result in a common location.

8.2.3 CALL: CALL calls ACALL which is local to ARITH. ACALL sets up ARITH to process the CALL statement with argument list in much the same way as an external function reference is processed. ARITH outputs all the R-list needed for the arguments and returns to CALL.

8.3 Generalized flow of the translation process.

The basic translation algorithm used is similar to that used to produce reverse Polish notation.

For example:

A*(B+C)-D

is translated to reverse Polish as follows:

Step	E-list Item	Operator Stack Contents	Reverse Polish String
1	A	.(EOS)	A
2	*	.*	A
3	(	.*(	A
4	B	.*(	AB
5	+	.*(+	ABC
6	C	.*(+	ABC+
7	)	.*	ABC+
8	-	.-	ABC+*
9	D	.-	ABC+*D
10	E.O.S.		ABC+*D-.

This algorithm has been modified so that R-list, rather than Polish notation, is produced. The difference is that instead of outputting the name of a variable to a string, an instruction to load the variable is output, and instead of outputting an operator, an operation with operands named is output. For example, taking the expression used in the last example, the results are:

R-list	Corresponding Polish
R1=A	A
R2=B	B
R3=C	C
R4=R2+R3	+
R5=R1*R4	*
R6=D	D
R7+R5-R6	-

Almost all of the R-list generated by ARITH is in the form of R-list macro references. R-list macros are described in detail in the section on the R-list language.

8.4 Since no provision is made for saving intermediate result registers, all external function calls must be made before the remainder of the expression is evaluated. ARITH does this by forming the R-list for the expression in a block local to ARITH called ARLIST and outputting each function reference including argument expression evaluation to the R-list file as each argument list becomes completed.

In general, function results, except for the last function call, are saved in a block called FRSTB. After the entire expression has been scanned, ARITH outputs loads of the saved function results to the R-list file and then the remainder of the contents of ARLIST are output to R-list.

8.4.1 The Exponential Operator, **

For exponential operations which are not done in-line, the ** operator is really an external function reference. Since it has only two arguments, it can be called by value. So, the exponential operator is made to look like a basic external function call and some of the function processing routines are used. A problem arises in an expression like

```
A+(B*(C+D))**E/F
```

because the call to the exponential function must be output first preceded by all of the argument evaluating R-list to the R-list file.

The solution for this case is to save a marker with every left paren entered in the OPSTAK to point to the start of the ARLIST for what might be the first operand of an exponential operator.

For exponentials with integer or real base expressions, and an integer constant power which is greater than one and less than 7, ARITH selects an R-list macro code and outputs a macro to do the exponentiation in-line.

8.4.2 Subscripts

8.4.2.1 Standard: ARITH's subscript processor produces two kinds of array references for standard subscripts: a subscript psuedo-macro which is processed by DOPRE in the second pass, and a simple variable load macro with a constant addend.

8.4.2.2 Non-Standard: If the subscript processor finds that the subscript is not in standard form, it resets the E-list pointer to address the start of the subscript. It then adds a non-standard-subscript operator to the operator stack, and returns control to the general expression scanner (at NEXTE).

There are three kinds of commas that ARITH must deal with: an argument comma, a subscript comma, and a complex-constant comma. To do this, ARITH keeps a cell called EMODE which indicates whether it is in argument mode, normal expression mode or subscript mode. Initially, EMODE is set to normal expression mode. As each left paren is met, the current mode is saved in the left paren entry in the OPSTAK and EMODE is set to normal, or argument, or subscript if the left paren is normal, or follows a function name, or follows a subscript name, respectively. As each left paren is popped from the stack, EMODE is reset to what it was before that left paren was encountered in E-list.

Argument and subscript commas are psuedo operators and when popped from the OPSTAK they initiate the action necessary to complete the processing of the argument or subscript. As each subscript comma is popped, it causes some of the index function R-list to be generated. When the subscript operator itself is popped from the stack, the final index function R-list is produced followed by a macro to load the array element name.

Since a non-standard subscript expression can be any arithmetic expression, it's possible to have subscripted subscripts to any depth. This means that the non-standard subscript processing must be able to operate recursively.

So, when the subscript operator is added to the OPSTAK, it is preceded by information about the subscript currently being processed. This is the same way that the function processor worked.

8.4.3 Relational, Logical, and Masking Expressions

Processing the four kinds of expressions with the same translator presents no serious problems to the basic algorithm. The relative hierarchy of the explicit operators are:

```
**
/,*
-,+
relationals
.NOT.
.AND.
.OR.
```

No distinction is made between the arithmetic operators (/,*,-,and +) and the relationals. The logical operators become masking operators if their operands are non-logical. Since logical operands are only legal for the logical operators, and since .AND. and .OR. must have both operands logical or both non-logical, it is impossible to have an expression that contains both masking and logical subexpressions.

Otherwise, expression types are mixed in any way. For example,

```
A+B.LE.C.AND.L1 (L1 is logical)
```

is a logical expression with relational and arithmetic subexpressions. It is translated as follows:

R-list	OPSTAK
R1=A	+
R2=B	.LE.
R3=R1+R2	.AND.

```
R4=C
R5=R3.LE.R4
R6=L1
R7=R5. AND .R6
```

8.4.4 Distinctions made between operand types in arithmetic and relational expressions.

When an operator is popped out of the OPSTAK, it enters a jump or vector table where an R-list macro code is assigned to it, and it is sent to the appropriate processor. For the relational and arithmetic operators other than **, if the operands of the operator are typed integer, the macro code is increased by one; if they are double precision, the code is increased by two; if complex, by three; and if real, the code is used as is.

Before the macro code incrementation is made, the operand types are compared. If they are not the same, an R-list macro is output to convert the lower type operand to the same type as the higher operand.

8.5 Optimizations

8.5.1 Compile-time Data-type Change

When an operator is popped from the stack, if its operands are of different types, and if the operand of lower type is constant, it will be converted to the higher type and the R-list instruction which loaded it or set, it will be replaced by one that loads the converted constant.

8.5.2 Compile-time Constant Subexpression Evaluation

Before a macro is formed for an operator, if it is an arithmetic operator and if its operands are integer or real constants, the R-list for the operands are no-oped, the operation is performed on the operands, and a load or set of the computed value is output to ARLIST. If this load is subsequently used as an operand with another constant operand, it will be no-oped just as the loads it replaced were no-oped.

8.5.3 Division by Real or Complex Constants

If a real or complex constant is preceded by a divided operator, R-list is output to load its inverted value and the divide is changed to a multiply operator.

8.5.4 Expression Transformations

Some expressions or subexpressions can be transformed to other mathematically equivalent forms which evaluate faster on the 6600 than they would if translated by the basic algorithm.

8.5.4.1 The R-list produced for

```
A*B*C*D
```

would compute the product as if it had been written

```
((A*B)*C)*D
```

and thus not take advantage of the 6600's two multiply units. If it had been written as

```
(A*B)* (C*D)
```

the two products in parentheses would be computed simultaneously. In order to achieve this effect, Arith keeps a flip-flop for popping multiply operators by operators of equal hierarchy. The flip-flop is flipped for every multiply operator encounterd in E-list, so that for

```
A*B*C*D+E
```

the first multiply is popped by the second, as usual, but the second is not popped by the third. The last two will be popped by the plus thus resulting in

```
(A*B)* (C*D)+E
```

8.5.4.2 Normally

```
A*B*C/D
```

results in

```
((A*B)*C)/D
```

which gives no parallel execution. But if divide is given a higher priority or hierarchy than multiply, then

```
A*B*C/D
```

is evaluated as

```
(A*B) * (C/D)
```

and the divide and multiply units are working simultaneously.

Note: It might be well to note at this time that the rules for carrying out these transformations are general rules and are always in effect in the translation algorithm. The translator never looks at a source item in E-list more than once, except for non-standard subscripts. (see 8.4.2.2)

8.5.4.3 -A+B or, to illustrate the preceding note,

```
-(A-B)+C*D
```

becomes

```
C*D-(A-B)
```

which reduces the number of operations from four to three, by the following rule:

> If a unary minus is about to be popped from the stack by a plus, remove the unary minus from the stack and replace the plus with a reverse-operand minus operator. The macros associated with the reverse-operand minus operator are the same as those for a normal minus operator except that the first parameter is subtracted from the second instead of the second from the first.

8.5.4.4 A/B/C becomes A/B*C, replacing a 29 cycle divide with a 10 cycle multiply (6600), by the following rule:

> If the current E-list item is a divide, and the last operand in ARLIST is not type integer, and the last operator in the stack is a divide or a multiply-D operator, then change the current divide to a multiply-D operator which has higher priority than

the divide and specifies a multiplication. Introducing the multiply-D operator allows more than one sequential divide to become a multiply:

```
A/B/C/D/E becomes
A/(B*C*D*E), which happens
to become A/((B*D)*(D*E)) because
the flip-flop also applies to multiply-D
operators.
```

8.5.4.5 The following rules allow a great variety of transformations which allow for more parallel evaluation of expressions. Some examples of the transformations made are:

```
A*B/C*D          (B/C)*(A*D)
A+B/C+D          (B/C)+(A+D)
A+B*C-D          (B*C)+(A-D)
A*(B+C)/D        (A/D)*(B+C)
A-(B*C+D)-E      (A-E)-(B*C+D)
```

The rules are as follows:

When a right paren is encountered in E-list, set the GP (greater priority) flag bit in the operator following the right paren. Or, if the current operator pops an operator with a higher priority out of the stack, set the GP bit in the current operator word.

If the GP bit of the current operator is set, and it is about to pop an operator (other than unary minus) of equal priority, or it is a divide and the last operator in the stack is a multiply, then don't pop the operator from the stack, set the CGP (confirmed GP) bit in the current operator and the GPTU bit in the information word of the last ARLIST entry. If the operator left in the OPSTAK is a minus, change it to a reverse minus. Add the current operator to the stack.

The GPTU bit makes an ARLIST entry temporarily unavailable for use as an operand. After an operator with its CGP bit set is popped, the last ARLIST entry with its GPTU bit set has the bit turned off.

The following example will illustrate the use of these rules.

A*B/C*D (B/C)*(A*D)

ARLIST	OPSTAK
R1=A	*
R2=B	/
R3=C	CGP, GP, *
GPTU R4=R2/R3	
R5=D	
R6=R1*R5	
R7=R4*R6	

8.6 Arith Table Overflow Diagnostics

8.6.1 There is a fatal to execution diagnostic which says:

"EXPRESSION TRANSLATOR TABLE (table-name) OVERFLOWED. SIMPLIFY THE EXPRESSION."

There are three different tables which may become overflowed.

8.6.1.1 OPSTAK table

The size of the stack fluctuates as the expression is scanned. For example, it increases as left parentheses and operators of higher priority occur, and decreases as operators of lower priority occur.

Each operator entered in the stack requires one word of space, except for left parens which require two: one to mark the start of a possible exponential base and the other the operator itself.

The start of each function reference requires nine words which includes the recursive function processor information. If the reference occurs in an intrinsic function argument list, then the opstak is also used to save the R-names of the arguments which have been processed so far, which have been processed so far, which could be up to 62 words if the function is a MAX or MIN type function.

The start of non-standard subscripts requires four words.

The OPSTAK block size may be modified by changing the EQU named MXOSE in the common file called OPTIONS, and reassembling ARITH.

8.6.1.2 FRSTB

This is the function result-saved table. A one word entry is made for each function that has its results saved. For example, in

A=F1(B)+F2(C)+F3(F4(D)+F5(E))+F6(F)

F1 and F2 are saved, and then F4 is saved but is reloaded to add to F5 so the size of the table goes down by one, and finally F3 is saved before calling F6.

The FRSTB block size can be modified by changing the EQU named MXFRSTB in the Option file and reassembling Arith.

8.6.1.3 ARLIST

This is ARITH's R-list block. The size increases as the expression is scanned, but it decreases after each external function reference is output to the common R-list file. A variable load entry takes 6 words; an operation takes four words if single length operands, and 8 if double length, a standard subscript psuedo-macro takes fourteen words.

The size of ARLIST is controlled by the EQU named ARLSZ in the Options file.

8.7 The Register Jam Problem

The second pass was designed under the assumption that there would always be a sufficient number of X-registers to be used in the evaluation of expressions, with the following type of exception:

> A very long expression can be constructed in such a way that the result registers of enough subexpressions must be saved so that a point is reached where enough registers to continue do not exist.

The assumption was correct for integer and real expressions, but it was found that it was quite easy to

run out of registers when evaluating Double or Complex type expressions.

The problem has been partially solved by modifying the expression translator to produce a different kind of output for Double and Complex expressions; partially solved because it is still possible to run out of registers for Real or Integer expressions.

These modifications to ARITH make up over 20% of the total number of source lines in ARITH, so it's important that they be described.

8.7.1 The Solution in General

When the second pass finds that it does not have enough registers to complete a sequence of statements, it reduces the number of statements in the sequence and begins again.

The solution is to break up a double or complex expression into many statements, one statement per operation. For example,

```
D1=D2*D3+D4
```

is made to look to the second pass like

```
ST1 = D2*D3

D1  = ST1+D4
```

where ST1 is statement-temporary one and is treated as a double variable.

8.7.2 The Solution in Particular

8.7.2.1 Double Length Operations

When ARITH is ready to output a macro for an operation, if the operands are double-word-length loads, it no-ops the load instructions and outputs a macro which will load the operand and do the operation. It then outputs a macro to store the results of the operation in Statement-Temporary storage followed by an end-of-statement psuedo-op. This macro is called a DSTR macro. The DSTR macro is in the same format as a double load macro and will be used as an operand to subsequent operators.

8.7.2.2 Mixed Single and Double

An expression with mixed single length and double length operands, for example real and complex, presents a new problem. The expression

```
A+B*(C1+C2)  , (C1 and C2 complex)
```

would not result in

```
R1=A
R2=B
R3, 4=C1 (no-oped)
R5, 6=C2 (no-oped)
R7, 10=C1+C2
ST1=R7, 10
EOS  (end-of-statment op)
R11, 12=CMPX(R2)
Etc.
```

but at the point of the last line, a reference to R2 is made which is defined in the previous statement which may end up in another sequence and therefore be undefined in this sequence. R1 won't be referred to until after the multiply operation which will occur two statements away, and so the chances that it will be undefined are even greater.

This problem has been solved by converting all unused single length operands to type Double Precision in an expression that contains a double or complex operand. The method of doing this is as follows: As each load or operation is output to ARLIST, the type is compared with the type of the last operand in ARLIST. If one is type integer or real and the other is double or complex, then a flag is set to indicate that mixed single and double has occurred, and the contents of the ARLIST block are scanned, inserting macros to convert operands which have not already been used in operations to double precision. Thereafter, the type of each entry to ARLIST is checked, and if not double or complex, a macro is output to convert it.

Mixed Super-and Sub-expressions

A modification to this method of dealing with the mixed single and double problem is necessary because of real or integer argument expressions occurring in a Double or

Complex super-expression. Also, index functions in a double expression should not be forced to be computed in double precision. This makes it necessary to treat argument and subscript sub-expression as an autonomous expression with regard to whether mixed single and double has occurred, and with regard to how far back in the ARLIST to go when the first double operand occurs in a mixed single and double expression. And since subscripted subscripts and function references in argument expressions, etc. exist, it is necessary to keep track of where in ARLIST each subexpression starts, and for each subexpression whether mixed single and double has occurred. This introduces another table which can overflow. The name of the table is JAMTB1. The size is controlled by the EQU named JAMTB1MX in the options file. An entry of two words is made at the start of each argument and non-standard subscript. The table is reduced by two at the end of each argument and non-standard subscript.

8.7.2.3 Consider the statement

```
A(I*J)=D1+D2*D1 (D1 and D2 double)
```

The subscript I*J is non-standard. The R-list produced for the statement would normally consist of the calculation of the index function followed by the evaluation of the expression followed by a store of the result into A(I*J). But since ends-of-statement operators now follow the double plus and double multiply operations, the subscript calculation may end up in another sequence.

So, ARITH now moves the R-list to compute a non-standard index function from the front of the ARLIST block to the end before outputting the store macro.

Because of this and multiple replacement statements, it is necessary to remember the starting point in ARLIST of each replacement variable. The limit on the number of replacements per statement is 75.

8.7.3 Subscripts and Double Length Operands

To make the implementation of this solution to the register depletion problem more feasible, Double or Complex operands in subscripts has been made illegal, and

all Double or Complex array subscripts are considered non-standard.

9.0 Restrictions and Other Remarks

9.1 Basic syntax checking is done by looking at the E-list element following the current E-list element. The following table indicates the syntax rules:

E-list element	may be followed by
CON (constant)	),,, E.O.S., OPS (2.)
ID (name)	),,, E.O.S., OPS, = , (
) (1.)	),,, E.O.S., OPS, =
,, =, (, .OR., .AND.	CON, ID, (, -, +, .NOT.
.NOT., relational ops	CON, ID, (,-,+
-,+,*,/,**	CON, ID, (

(1) If) is the closing parens of an IF expression it may be followed by an ID (if Logical IF) or constant (label). If) is in I/O list (IXFN call) it may be followed by (or ID.

(2) OPS = .OR., .AND., relationals, -, *, /, **

9.2 The format of ARITH in COMPASS

Label field starts in column 2, operator in column 11, and symbols or integer constants in column 21. Instructions (other than 50-57) which have more than one result register (e.g., Unpack) are written so that the B register name starts one character after the end of the operation field. Operand registers start in column 18. This format results in all result registers, operand registers, and symbolic references to be found in column 12-16, 18-20, and 21-72 respectively. Comments start in column 31.

Every conditional branch instruction has a comment stating what condition must be met in order to branch.

9.3 At NEXTE, B1 is set to the address of the next E-list item to process. A large part of ARITH assumes that B1 holds this address. Therefore, B1 should be used very carefully.

Except for B1, in general it may be assumed that any register may be destroyed when calling a subroutine (including B1 for external routines). Any exceptions to this are noted in the introductory comments of the exceptional routines.

9.4 Caution to modifier. ARITH is not laid with booby traps but may appear to be so because of the complexity of the task. Transformations, etc., cause unexpected results. Beware of functions, non-standard subscripts, and exponential operators.

10.0 Debug Subroutines

10.1 Function tracing

10.1.a FNPP - Debug FUNCS Pre-Processor

FNPP is called from ARGPIRT when either the function tracing bits are set in the symbol table entry or the trace all functions flag is set.

a) Generate call to BUGFNN macro via IGCALL with

```
DBGAPL    VFD     42/function name, 18/type
          VFD     60/0
          VFD     60/8RBUGFNN
```

b) Next available aplist number is updated, BUGFNN is entered in the symbol table with type CGS, and the function tracing flag is set.

10.1.b FN - Debug FUNCS Processor

After the call to the function is issued, a call to BUGFUN is generated to indicate return from and value returned by the function.

a) Save RLIST pointers.

b) Issue temporary store for function result (2 for double word results).

c) Generate call to BUGFUN macro via IGCALL with

```
DBGAPL    VFD       42/function name, 18/type-
          VFD  30/ST. ordinal for the value
                    returned 30/IH of ST.-
          VFD  60/0 VFD  60/8RBUGFUN
```

d) Enter BUGFUN in symbol table with type CGS.

e) Issue load and transmit of function result and enter in the saved function results table.

f) Flush debug code via DARLIST.

g) Restore RLIST pointers.

h) Turn off function tracing flag; return.

10.2 ARR - Debug ARRAYS Processor

ARR is called from the subscript processor when either the arrays tracing bits are set in the symbol table entry or the trace all arrays flag is set.

a) Save RLIST pointers.

b) If the last macro was a load, issue transmit.

c) Process saved function results via ARGP8CR.

d) Generate call to BUGARR macro via IGCALL with

```
DBGAPL    VFD       42/array name 18/0
          VFD       60/array bound
          VFD       30/ST. ordinal of dimension
                    being checked 30/IH of ST.
          VFD       60/0
          VFD       60/8RBUGARR
```

e) Enter BUGARR in symbol table with type CGS.

f) Indicate unsaved function result and define result (array dimension being checked) to be in X6.

g) Flush debug code via DARLIST.

h) Restore RLIST pointers.

10.3 STRCK - Debug STORES Processor

STRCK is called from the assignment statement processor when the stores checking bits are set in the symbol table entry.

a) Save RLIST pointers.

b) Get and save variable name and type to build aplist.

c) If variable dimensioned, calculate element and issue temporary store (2 for double word element).

d) From the AASI table determine the frequency count for stores without relational operators. If zero, go to i) to check for stores with relationals.

e) For stores without relations, set up dummy relational operator=9 to indicate no relational expression.

f) Collapse DBGAPL to eliminate word used for test in the relational expression.

g) Generate call to BUGSTO macro via IGCALL with

```
DBGAPL    VFD      42/variable name, 15/relational
                   operator, 3/type
          VFD      60/value stored into the variable
          VFD      30/ordinal in CON. of constant
                   involved in the expression
                   30/IH of CON.
```

Used with stores with relationals Collapsed out in step f for stores with checking operators or stores without relationals

```
          -or-

          VFD      42/0, 18/IH of variable in-
                   volved in the expression

          VFD      60/0
          VFD      60/8RBUGSTO
```

h) Enter BUGSTO in symbol table with type CGS.

i) Check for links for stores with relationals due to interspersed specifications and links due to packet specifications. Exit to step m) when finished.

j) Get relational operator out of the options word. RANGE and INDEF will use the collapsed aplist (step f).

k) If variable after relational operator go to step n).

l) Determine which table the constant is in. If in the global table, enter in the CON table. Enter ordinal in aplist, and go to step g).

m) Restore RLIST pointers and return.

n) Get the variable ordinal and form the aplist with symbol table ordinal instead of a constant table ordinal, and go to step g).

10.4 Debug TRACE Processing

a. When the appropriate IF statement routine is entered, the tracing flag, TRCFLG, is set greater than or equal to zero if the current DO level is less than or equal to the desired DO level of tracing.

b. For an IF statement of the form

IF (expression) l1, l2, l3, or IF (expression) l1, l2

IFBRT is called to process each branch.

In IFBRT if the tracing flag is set, a generated label is reserved for each branch and is stored in order in a temporary buffer. For each branch an aplist is set up containing the label of the branch (30/BCD for the label, 30/binary for the label). Using RTNM, the debug IF macro, a call to BUGTRU is generated. Upon return from BUGTRU, a branch is made to the actual label used in the IF statement.

c. For a logical IF statement of the form

IF (logical expression) statement

a call to BUGTRT is generated via IGCALL with

```
DBGAPL    VFD       60/0
          VFD       60/8RBUGTRT
```

This call is generated before the code for the true side of the IF statement. At object time BUGTRT will issue a message to indicate transfer to the true side of a logical IF.

ASFPRO

1.0 General Information

ASFPRO consists of two independent subroutines ASFDEF and ASFREF. ASFDEF processes all ASF definitions by saving the text in a table; ASFREF processes all references to ASF's by expanding the E-list and inserting in the ASF definition.

2.0 Usages

2.1 ASFDEF

Function of ASFDEF

Calling ASFDEF will process the entire E-list. The modified E-list is moved to ASFTAB.

ASFDEF is called with a return jump. It is only necessary that SELIST (RA+32B) point to the E-list entry containing the ASF name. Return is to PH2RETN either directly or through ERPRO.

Processing flow description

1. The parameter list is checked for proper format.
2. All references (to parameters) on the right of the equal sign are replaced with parameter ordinal indicators. Any entries in CONSTOR are moved to the ASFTAB area.
3. The E-list after the equal sign is moved to the ASFTAB area.
4. The ASF text is linked to any previous ASF texts.

2.2 ASFREF

Function of ASFREF

ASFREF is called whenever a reference to an ASF is encountered by ARITH. ASFREF replaces the reference to the ASF with the ASF text resulting in an expanded E-list statement.

ASFREF is called by a return jump with SELIST pointing to the ASF name within E-list.

Processing flow description

1. The ASF name is checked to see if it has been properly defined.

2. The remainder of E-list is moved to the scratch table.

3. The parameters to the ASF are bracketed and checked for correspondence in number with the definition.

4. The text is expanded and appended to E-list.

5. The part of the statement following the ASF parameter list is E-list.

3.0 Diagnostics Produced

1. Dummy parameter in an arithmetic statement function definition occurred twice.

2. Arithmetic statement function has caused a table overflow while being processed.

3. Arithmetic statement function has more dummy parameters than allowed.

4. Arithmetic statement function has an improperly formed parameter list or no = following the list.

5. A reference to this arithmetic statement function was not followed by an open parenthesis.

6. Insufficient memory was available for the evaluation of this arithmetic statement function reference, possibly a recursively defined ASF.

7. A reference to an improperly specified arithmetic function has been encountered.

8. A reference to this arithmetic statement function has balanced parenthesis within the parameter list.

9. The number of parameters used in referencing this arithmetic function does not correspond to the number in its definition.

4.0 Environment

4.1 External Symbols

ERPRO	called to issue fatal errors
RSELECT	holds a non-zero value for R=2 or R=3
PH2RETN	exit to the phase two controller
IDORDL	holds the symbol table ordinal of the name (ASF)
NAMFWA	holds address of word A of the name (ASF)
PSYM	format a symbol for an error message
FWAWORK	holds the first word address of working storage
LWAWORK	holds the last word address of working storage
S.SCR	size of the scratch table
O.ASF	origin of the ASF table
L.ASF	length of the ASF table
O.SCR	origin of the scratch table
L.SCR	length of the scratch table
Z.SCR	number of the scratch table
ALLAE	routine to allocate almost all storage to a manager table
ADDREF	routine to collect a reference
MOVE	routine to move an area of core or a table
ALLOC	routine to allocate space to a table
Z.ASF	number of the ASF table

5.0 Processing

5.1 ASFDEF

a. Save the ordinal of the statement function in IDORDL and the address of word B in WORDB.

b. If the next E-list element is not a name, issue an error.

c. Increment the parameter count.

d. If the next item is a comma, go to b.

e. If it is not a a right parenthesis, issue an error for a bad parameter.

f. If the next item is not an equal sign, issue an error.

g. Save the number of parameters and first word address of the statement function text.

h. Issue an error if more than M.FP parameters appear.

i. Extract a parameter from the list.

j. Decrement the number of parameters to go.

k. If this E-list item does not match the parameter, go to i. If the item is an end of statement go to o.

l. If the item after the formal parameter is a constant or a name, produce an error.

m. If the parameter matched an E-list item prior to the equal sign, issue an error for a doubly defined formal parameter.

n. Substitute the formal parameter indicator (2077---...nB), where n is the formal parameter number.

o. Advance to the next formal parameter. Go to i.

p. Allocate space in the ASF table to hold the statement function text.

q. Place the number of arguments in the FARG field and the distance from O.ASF in the RA field of word B for the statement function.

r. Move the text into the ASF table at the end.

s. Setup to scan the E-list string for constants.

t. Examine the next E-list element. If it is an EOS, go to y.

u. If it is not a constant, go to t.

v. If it is a logical constant, go to t.

w. Compute the number of words occupied by the constant and increment the total space needed in ASFTAB to hold all constants.

x. Make an entry in the temporary table constructed over the E-list of the form VFD 12/2000B + word count, 48/ASFTAB ordinal of CONSTOR entry. Go to t.

y. Phase two - Move the constants from CONSTORS to the ASF table. If no constants, go to af.

z. Allocate space at the end of the ASF table to hold the number of constant words needed. Increase L.ASF by this size.

aa. Pick up the temporary table entry for the constant.

ab. Pick up the E-list for the constant and install a local pointer into the ASF table to where the constant is located. (Note: This pointer address is absolute.)

ac. Update L.ASF and move the constant from the CONSTORS table to the ASF table.

ad. Decrease the number of constants to be processed.

ae. If more constants remain, go to aa.

af. Collect a reference to the statement function, if necessary.

5.2 ASFREF

a. Pick up word B and extract the ASFTAB index and number of arguments.

b. Allocate all nearby storage to the scratch table.

c. Extract an E-list item.

d. If it is a constant, name or an operator with a precedence, go to c.

e. For a left parenthesis, increment the paren count and go to c.

f. If an end of statement is found, issue an error for unbalanced parens. For paren count = 1, go to h.

g. Go to c if it is not a right paren. Otherwise, decrement the paren count and go to c.

h. If no parameter intervened, diagnose a vacuous parameter.

i. Make an entry in the argument substitution table of the form 24/0, 18/FWA, 18/length. Advance parameter ordinal.

j. Issue an error if storage is exceeded during scratch table construction.

k. Move the actual parameter to the scratch table area above the argument substitution table.

l. If the next E-list element is a comma, go to c.

m. If the parameter ordinal does not match the number of declared parameters, issue an error.

n. Move the remainder of the statement E-list to the scratch table and diagnosis any storage overflow. Adjust CWAWORK.

o. Place a left parenthesis in the now freed up E-list area to simulate a fully parenthesized expression. Decrement SELIST.

p. Setup to scan the ASF text table.

q. Extract a text item. If it is a parameter, go to u.

r. Store the element in the E-list area being constructed.

s. If it is an end of statement, go to aa.

t. Check for storage availability. If there is no problem, go to q, else produce an error message.

u. Obtain the argument substitution word for this parameter.

v. If the actual parameter is more than one word of E-list, go to x.

w. Store the parameter word in the E-list area and go to t.

x. Place a left parenthesis in the E-list area.

y. Copy the expression to the E-list area.

z. Place a right parenthesis in the E-list area and go to t.

aa. Move the remainder of the E-list after the expression, update LELIST, LWAWORK, release the scratch table space and diagnose an error if LWAWORK is less than the origin of the scratch table minus one.

ab. Exit from ASFREF.

CALL

1.0 General Information

Task Description

The function of the CALL statement processor is to translate E-list for a CALL statement into R-list and issue appropriate macros to the R-list file.

2.0 Entry Points

2.1 CALL

This is the main entry point of the CALL processor. It is entered by a return jump instruction and exit is made through this entry point if no fatal errors occur in the statement being processed. Otherwise, control is returned to PH2RETN in PS1CTL after ERPRO has issued a fatal error diagnostic.

2.2 IGCALL

This entry point is called to form and issue the R-list macros for a subroutine call. It is entered by a return jump instruction, and a return is always made back through the entry point.

3.0 Diagnostics And Messages

3.1 Fatal To Compilation

None

3.2 Fatal To Execution

ILLEGAL CALL STATEMENT FORMAT

ILLEGAL RETURNS PARAMETER LIST

3.3 Informative

None

3.4 Non-ANSI

RETURNS PARAMETERS IN A CALL STATEMENT

4.0 Environment

4.1 Low Cores Cells

SYM1	(12B)	starting address of symbol table
SELIST	(32B)	address of next E-list element
DUKE	(37B)	line counter of source cards
NGLN	(52B)	next available generated label number
NRLN	(64B)	next available result number

4.2 Local Cells And Symbols

CALN	temporary for current APLIST number
JUMP	30-bit jump instruction
LOCA	argument list table pointer
SRLIST	starting address of RETURNS list in E-list format
TRCTS	temporary for TRACEL Information word
TRCFLG	debug mode trace flag
TSAPL	temporary for APLIST number
TS1	buffer area for macros
TYPECLL	debug mode call-return flag

4.3 Common blocks

4.3.1 /CLNFO/ (used only by CALL and ARITH)

SUBFWA	address of the first word of the symbol table entry for the name of the subroutine being called

SUBH symbol table ordinal of the subroutine name

ARGCNT number of arguments in parameter list

NARGSF argument list flag-equals 0 if there is an argument list

SUBNAME name in E-list format of subroutine being called

ARLPT ARLIST buffer pointer-number of words in buffer for current statement

4.3.2 /NAALN/

NAALN next available APLIST numbers

4.4 Externals

ACALL code block in ARITH called to process the subroutine name and the argument list

ADDREF code block in PS1CTL called to record a reference for the subroutine name

ALLCALL debug cell in DBGPHCT used to indicate whether subroutines references are to be traced unconditionally

APLRT code block in ARITH called to issue an APLIST macro to the ARLIST buffer

ASAER code block in ERPRO called to issue a non-ANSI usage diagnostic

CONVERT code block called to place a constant in the CON. constant table

DARLIST code block in ARITH called to dump the ARLIST buffer to the R-list file

DBGAPL debug cells in ARITH containing parameter list symbol table information from which the R-list macros are formed

DFLAG debug cell in FTN used to indicate whether the debug option has been selected

DOCALL code block in DOPROC called to record that an external reference has occurred

DOLABR code block in DOPROC called to records that a reference to a statement label has occurred

ERPRO code block called to issue fatal to execution diagnostics

GEFCM code block in ARITH called to issue a general external function macro to the ARLIST buffer

INITR code block in ARITH called to initialize pointers before statement and macro processing

LABEL. cell in LSTPRO containing the symbol table ordinal for LABEL

PH2RETN code block in PS1CTL returned to after a fatal to execution diagnostic has been issued

RSELECT cell in FTN used to indicate whether either of the long reference map options has been selected

STRIP code block in ARITH called to strip off a trailing $ from a symbol table word

SYMBOL code block in LSTPRO called to make a new entry into or search for an existing entry in the symbol table

TRACEL debug cell in DBGPHCT used to hold the level number for the TRACE debug statement

WRWDS code block in FTN called to perform the writing of R-list macros to the R-list file

5.0 Processing

5.1 Initial Processing

Upon entry to the CALL processor through its main entry point, the routines INITR and DOCALL are called to initialize pointers and flags in ARITH and to inform DOPROC about an external reference. A syntax check is performed on the first elements of the E-list string for

the statement, and a fatal to execution diagnostic is issued and the processor exited if a name followed by a left parenthesis, comma, or end-of-statement if not found. If no error conditions are sensed, ACALL is called to process the subroutine name and the parameter list if one exists.

5.2 Intermediate and RETURNS Processing

When a successful return is made from ACALL, the debug flag DFLAG is checked to determine whether the debug option has been selected. If debug mode is not specified, the trace flag TRCFLG is set non-zero and RETURNS processing is initiated. Otherwise, if debug tracing has been specified for this subroutine, the trace flag is set to zero.

Further syntax analysis is performed to determine whether a RETURNS list is present and correctly formatted. Macro processing begins otherwise if there is no list and no error conditions occur.

At the beginning of RETURNS processing, the routine APLRT is called to generate an APLIST macro of minus zero. In case debug tracing is selected for the subroutine, the address of the E-list item for the first RETURNS label is saved in SRLIST. This address is later needed in order to issue debug subroutine calls for non-standard subroutine returns.

As each label is obtained from the E-list string, a check is formed to ensure that the item is an integer constant. Otherwise, a fatal to execution diagnostic is issued and processing terminated. DOLABR is called to process the label and record its occurrence. APLRT is then called again to issue the APLIST macro for the label. The process is repeated until either an error occurs or the RETURNS list is exhausted. If debug tracing has been indicated, additional information is gathered and overwritten in the E-list area for the label being processed. The information consists of a label table ordinal for the label, the symbol table ordinal for the label, and the next available number for a generated label. The current value of the debug trace flag TRACEL is set to zero during this phase of processing so that a label table entry will be created for the label. This entry is used as part of the inputs to the debug routine BUGCLR$.

5.3 Macro Processing

If the TRCFLG is zero, FARGLST is called to set up the DBGAPL table for a debug subroutine call and to call IGCALL to process the table. Otherwise, IGCALL is entered directly to output a 60-bit return jump macro with traceback information. TRACFLG is then tested again to determine whether the second debug subroutine call should be generated. DARLIST is called to dump the contents of ARITH's macro buffer to the R-list file. Processing is finished if there is no RETURNS list present, or if no debug tracing is specified for the subroutine.

When subroutine tracing is selected, the saved debug information in the overwritten E-list string is fetched for each RETURNS labels, and debug subroutine call macros and associated actual parameter list macros are written directly to the R-list file. SRLIST is reset to zero and the processor exited as soon as the terminating parenthesis for the list has been reached.

5.4 FARGLST

This subroutine is called by the processor to form the argument list used to generate debug subroutine calls. The flag TYPECLL is preset before entry to either zero or one to indicate that a subroutine call or standard subroutine return is being traced. The subroutine name and the TYPECLL value are placed in the CON. constant table, and the returned IH, CA information is used to form the first two words of the DBGAPL. The final two words consist of a zero word followed by the address of the debug routine name BUGCLL$. IGCALL is called to process the table and generate the debug subroutine call.

5.5 IGCALL

IGCALL is called from within the CALL processor and from other pass 1 processors to form and write R-list for a subroutine call. Upon entry, A1 contains the address of the parameter list which is to be processed. The IH and CA fields are extracted from each argument word in the list, and this information is passed to APLRT who issues an appropriate APLIST macro. This process continues for each word of the argument list until the zero word of the table is sensed. If a subroutine is being compiled which

has no parameter list, this procedure is bypassed since the first word of the parameter list will be zero.

The subroutine name is then entered into the symbol table, the external bit is set for the name, and GEFCM is called to output a general external function macro. IGCALL exits after DOCALL is called to record that an external reference has been made.

6.0 Table Structure

The debug APLIST table (DBGAPL) is the only table used by the CALL processor. The table is variable in length, and contains information regarding debug calling sequences.

The debug APLIST table is n+2 words long for a generated calling sequence containing n parameters. The first n words are of the format:

```
VFD  12/0,18/ CA of arg, 30/IH of argi.
```

The final two words are of the format:

```
VFD  60/0
VFD  60/addr
```

where addr is the address of the E-list representation of the subroutine name.

GOTO

1.0 General Information

Task Description

This processor translates E-list and issues appropriate macros to the R-list file for all GOTO type statements except unconditional GOTO's on the true side of a logical IF statement. It also processes ASSIGN statements and generates R-list macros for them.

2.0 Entry Points

2.1 PLAB

This entry point is called internally from the GOTO processor to process a list of transfer labels appearing in either computed or assigned GOTO statements. Entry and exit is made through the entry point unless an error condition arises during processing. In this case the routine exits by jumping to the address specified by A0.

2.2 GOTO

This entry point is called to process any of the three forms of the GOTO statement. Entry and exit are both made through the entry point unless error conditions are sensed.

2.3 ASSIGN

This entry point is called to process an ASSIGN statement. Entry and exit are always made through the entry point.

3.0 Diagnostics And Messages

3.1 Fatal To Compilation

None

3.2 Fatal To Execution

GO TO STATEMENT - SYNTAX ERROR

MISSING OR SYNTAX ERROR IN LIST OF TRANSFER LABELS

PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USES

THIS ASSIGN STATEMENT HAS IMPROPER FORMAT, ONLY ALLOWABLE IS. (ASSIGN LABEL TO VARIABLE)

VARIABLE IN ASSIGN OR ASSIGNED GO TO IS ILLEGAL

3.3 Informative

THIS STATEMENT BRANCHES TO ITSELF

3.4 Non-ANSI

GO TO STATEMENT CONTAINS NON-ANSI USAGES

4.0 Environment

4.1 Low Core Cells

SYM1	(12B)	starting address of symbol table
DIM1	(17B)	starting address of dimension table
CLABEL	(23B)	label field of current source line
SELIST	(32B)	address of next E-list element
DUKE	(37B)	line counter of source cards
NGLN	(52B)	next available generated label number
NLABEL	(60B)	label field of next source line
NRLN	(64B)	next available result number

4.2 /MACBUF/

BRSELF	branch to current label flag
ASA	ANSI flag

MACBUF macro buffer

TEMP padding buffer

4.3 /NAALN/

NAALN next available APLIST number

4.4 /STSORD/

STSORD next available statement temporary store number (Reset to 1 by PH2CTL the start of each statement)

4.5 Externals

ADDREF code block in PS1CTL called to record a reference for the index variable

ALLOC code block in PS1CTL called to allocate more table space for the jump table

ARITH called to process the index variable or expression for a computed GOTO

ASAER code block in ERPRO called to issue a non-ANSI usage message

CFO code block in DBGPHCT called to check debug usage of variable names with actual program usage

CONVERT code block called to place a constant in the CON. constant table

DARLIST code block in ARITH called to dump the ARLIST buffer to the R-list file

DFLAG cell in FTN used to indicate whether the debug option has been selected

DOFLAG cell in PS1CTL containing a DO-loop nesting level counter

DOLABR code block in DOPROC called to record that a reference to a statement label has occurred

ERPRO code block called to issue fatal to execution diagnostics

ERPROI code block in ERPRO called to issue informative diagnostics

GOTOSFL debug cell in DBGPHCT used to indicates whether assigned GOTO index checking is to be performed

IGCALL code block in CALL called to form and issue R-list for debug subroutine calls

INITR code block in ARITH called to initialize pointers before statement and macro processing

LABCON code block in DOPROC called to convert a label and enter it in the symbol table

LABEL. cell in LSTPRO containing the symbol table ordinal for LABEL.

L.LTAB cell in LSTPRO containing the length of the generated jump table

OPTLVL cell in FTN containing the selected level of code optimization

O.LTAB cell in LSTPRO containing the first word address of the label table used to generate the jump table

PH2RETN code block in PS1CTL returned to after certain fatal to execution diagnostics have been issued

PSYM code block in DOPROC called to prepare a name for usage in a diagnostic

RSELECT cell in FTN used to indicate whether either of the long reference map options has been selected

ST. cell in LSTPRO containing the symbol table ordinal for ST.

SYMBOL code block in LSTPRO called to make a new entry into or search for an existing entry in the symbol table

TRACEL debug cell in DBGPHCT used to hold the level number for the TRACE debug statement

WRWDS code block in FTN called to perform the writing of R-list macros to the R-list file

Z.LTAB equate in LSTPRO whose value is the number associated with the label table

5.0 Processing

Upon entry to the processor through the GOTO entry point, the E-list is examined to determine which type of GOTO statement is being compiled. If the first element is not a constant, further checks are performed to determine if the element is a name or not. In this manner, the GOTO statement is typed as being unconditional, assigned, or computed.

5.1 Unconditional GOTO's

A fatal to execution diagnostic is issued and the processor exits if an end-of-statement does not follows the statement label. Otherwise, the statement label is compared with the label field of the next source statement. If the labels do not match, processing continues with R-list generation. If the labels match, i.e., the unconditional GOTO branches to the next source statement, optimization is performed unless the debug mode of ompilation has been selected. DOLABR processes the label, the former RSN bit status is restored in the symbol table entry for the label, and the processor exits without needing to issue any R-list macros.

If no optimization can be performed, processing continues by calling PLAB to analyze and process the label and build a single word jump table. The information in this table is then used in formatting an unconditional jump macro which is issued directly to the R-list file. An informative diagnostic is issued if the statement branches to itself. Otherwise, the processor exits immediately.

In the case where debug flow tracing has been detected for the statement, a debug subroutine call is generated before the jump macro is written to the R-list file. This is done by calling INITR to set up pointers and the

ARLIST buffer in ARITH, building a debug argument list table for the call in the AGOCALL buffer area, calling IGCALL to process the table and generate the debug subroutine call, and calling DARLIST to write the generated macros to the R-list file. Processing then continues with the formatting of the jump macro as mentioned in the previous paragraph.

5.2 Assigned GOTO's

Processing begins by performing a syntax check to determine the form of the statement. Fatal to excution diagnostics are issued if an end-of-statement follows the index variable (indicating no label list) or if a left parenthesis does not precede the label list. An ANSI message is generated if the comma does not appear between the variable name and label list, but processing is not terminated. If no fatal error conditions are sensed, AGVAR is called to process the variable name and enter it in the symbol table.

The assigned GOTO macro is formatted and built in the MACBUF buffer from information returned by the AGVAR call. A reference for the variable is recorded by ADDREF if a long reference map option has been selected. PLAB is then called to analyze and process the label list and build a jump table for the labels. A final syntax check is performed to ensure that only an end-of-statement follows the right parenthesis of the label list.

If there are no requirements that the jump tables be present in the generated code, the assigned GOTO macro is written to the R-list file and the processor exited.

Further label table and macro processing is necessary if any of the following conditions are sensed:

- debug index checking
- debug flow tracing
- OPT=2 optimization selection

The generated jump table is used at object time by the debug object library, and is required at compile time for analysis by the super mode optimization pass.

Processing begins by removing redundant labels from the label table to reduce the total required code size needed for the generated jump table. If the table is needed for OPT=2 requirements, the only processing that remains is the writing of the assigned GOTO macro, the jump table, and a JP B0 macro to the R-list file. Otherwise, the two debug options are checked to determine which or if both of the options have been selected. Issuing of debug subroutine calls follows the same procedure as explained in 5.1. The processor exits after all remaining macros are issued to the R-list file.

5.3 Computed GOTO

If the first E-list element seen is a left parenthesis, PLAB is called to analyze and process the label list and build a jump table for the labels. Otherwise, an error diagnostic is issued and compilation terminated. After label processing, a fatal to execution diagnostic is generated if no index variable is found, or an ANSI message is generated if the index is not a stand-alone variable name. ARITH is then called to translate the index expression and issue the resulting R-list macros to the R-list file. The name of the error routine ACGOER. is placed in the symbol table and the external and no-return bits are set when entered the first time.

Debug processing takes a slightly different form if flow tracing has been indicated. The following tasks are performed:

- call INITR to set up pointers and the ARLIST buffer in ARITH
- build transmit and temp store macros in the MACBUF area
- output the macros directly to the R-list file (these macros will save the value of the computed GOTO index in a statement temporary for the debug object routine BUGGTC$.)
- build an argument list table for the debug subroutine call in the AGOCALL buffer area
- call IGCALL to process the table and generate the debug subroutine call

- call DARLIST to write the generated macros to the R-list file
- build a load macro in the MACBUF area
- output the macro directly to the R-list file (this macro will load the index value from the statement temporary so that it is defined for the GOTO macro.)

Processing continues with the formatting and generating of the computed GOTO macro. The macro is issued directly to the R-list file, followed immediately by the generated label table. The processor exits after any remaining ANSI messages are generated.

5.4 ASSIGN

Upon entry to this processor, the following processing occurs:

- obtain the first E-list element for the statement
- issue a fatal to execution diagnostic and terminate processing if the E-list item is not a constant
- call LABCON to convert the label and enter it in the symbol table
- generate a fatal to execution error if the label referenced is defined as or used in previous context as a format label, after calling PSYM to format the label for ERPRO

If no error conditions are sensed, the RSN and RAS bits are set in the symbol table for the label. A two-word assign macro is generated and stored in the MACBUF area.

Further syntax analysis of the statement is performed to ensure that a variable name appears followed by an end-of-statement. AGVAR is called to process the variable, and upon successful return the define bit is set for the variable. The remainder of the assign macro is built from returned information, and the entire macro issued to the R-list file. If either of the long reference map options has been selected, ADDREF is called to generate a reference for the variable. The processor then exits through its entry point.

5.5 PLAB

PLAB is called internally by the GOTO processor to process a list of transfer labels for the three forms of the GOTO statement. Upon entry, the following conditions hold:

- SELIST points to the first word of the label list
- A0 contains the return address in case an error is detected
- B6 contains the error number to be used in case of a syntax error
- B7 contains the E-list code for the list terminator

An initial pass is made through the table to determine the number of labels and diagnose any illegal syntax or non integer constants. ALLOC is called if no errors occur to allocate enough spaces for the label table in working storage.

A second pass of the label list is made to build a jump table which will be placed in the allocated storage just obtained. The processing steps consist of:

- obtain E-list representation of the label
- call DOLABR to process the label and enter it in the symbol table
- set the branch-to-self flag if the label matches the label field of the GOTO statement
- generate an R-list word consisting of an unconditional jump operation code (2054B), the label table ordinal for the label, and the symbol table ordinal for the label
- store the R-list entry in the label table area
- Repeat the above steps until all labels have been processed, then exit PLAB through its entry point.

5.6 AGVAR

AGVAR is called to process the variable name for ASSIGN and assigned GOTO statements. Immediately upon entry, SYMBOL is called to enter the name in the symbol table. If it is the first occurrence of the name and previous usage has occurred in a C$ debug statement, CFO is also called to check the setting of the debug bits. If it is not the first occurrence of the name, checks are performed to ensure that the name is not the name of the program unit, a function, or an external.

In either case, the variable bit is set for the name, and the routine exited if the variable is not equivalenced. Otherwise, the base-bias of the equivalenced entry is obtained before exiting.

6.0 Tables

Only the label table is produced by the GOTO processor. Each entry of the table is an unconditional jump R-list code, and is of format

```
VFD  12/2000B+OP,18/TRO,30/IH
```

where OP is equated to 54B to produce the unconditional jump operation code, TRO is an ordinal into the LABEL. table needed for debug mode processing, and IH is the symbol table ordinal for the label.

DOPROC

1.0 General Information

The DO processor (DOPROC) examines DO statements, DO-implied lists, statement numbers, statement number references, and integer variable definitions. It determines the characteristics of DO's and index functions, diagnoses nesting, syntax, and the use of statement numbers, and generates R-list macro words defining the beginning and end of each DO loop and DO-implied list. The DO statement is the only statement fully processed by DOPROC.

2.0 Entry Points

2.1 DOPROC

DOPROC is referenced by PS1CTL when a DO statement is encountered. DOPROC examines the E-list items of the DO for syntax. If the DO is found to be legal, it generates a label for referencing from the bottom of the DO, sets up the DO table (DOT) with flag and address information concerning the control variable, limits, etc, and generates an R-list macro for processing by the second pass DO processor (DOPRE).

The calling sequence for DOPROC is:

```
RJ   DOPROC
```

Upon entry to DOPROC, it is expected that low core location SELIST will hold the address of the E-list for the DO. Upon a successful exit from DOPROC, the R-list file will contain seven words relating to the DO and if necessary, label information will be filed in SYMTAB. O.DOTAB will hold the address of the list entry in the DOLIST (DOL table).

2.2 DODEF

DODEF is referenced by ARITH, ASSIGN and LISTIO (on input) whenever an integer variable appears as the object of a replacement statement, ASSIGN statement, or input statement. DODEF sets up an integer variable definition

item in the DOLIST (DOL table) and diagnoses illegal redefinition of loop limits.

The Calling Sequence for DODEF is:

```
RJ   DODEF
```

where B1 has been preset to the ordinal of the integer variable in the symbol table.

2.3 DOSYM

DOSYM is referenced by ARITH and LISTIO (output only) when an integer variable appears as an operand. DOSYM causes a search of the DOT table to see if the integer variable is the control variable for any loop and if so marks the control variable for materialization.

The calling sequence for DOSYM is as follows:

```
RJ   DOSYM
```

where B1 has been preset to the ordinal variable in the symbol table.

2.4 DOCALL

DOCALL is referenced by ARITH, CALL, AND LISTIO when a subroutine or function reference, explicit or implicit, is encountered. DOCALL will set a flag indicating a transfer out of the loop.

The calling sequence for DOCALL is:

```
RJ   DOCALL
```

2.5 DOIT

DOIT is referenced by LISTIO when it encounters a DO-implied list. DOIT initializes the loop as described under DOPROC.

The calling sequence to DOIT is:

```
EQ   DOIT
```

where B1 points to the E-list entry for the = sign.

Return is to DOITX.

2.6 DONE

DONE is referenced by LISTIO after processing the list of a DO-implied loop. DONE closes the loop as described under DOLAB.

The calling sequence for DONE is:

```
EQ   DONE
```

Return is to DONEX.

2.7 DOGOOF

DOGOOF is referenced by LISTIO after encountering a fatal error while processing the list of a DO-implied loop. This allows DOGOOF to remove the current nest of DO-implied loops.

The calling sequence for DOGOOF is:

```
EQ   DOGOFF
```

Return is to DOGOOFX

2.8 DOENT

DOENT is referenced by ENTRY. DOENT files an error message indicating that external entry to a loop is being attempted.

The calling sequence to DOENT is:

```
RJ   DOENT
```

2.9 DOEND

DOEND is referenced by ENDPRO to see if all loops have been terminated, all referenced statement numbers and format numbers have been defined as such, and all loops with entries also have exits.

The calling sequence to DOEND is:

```
RJ   DOEND
```

2.10 DOLABCN

DOLABCN is referenced by PS1CTL before each labeled statement is processed. The label is checked for prior definition and, if not found, is entered into the symbol table (SYMTAB). The loop in which the label is defined is entered in the symbol table, and whether or not the label is an entry point to a loop.

The calling sequence for DOLABCN is:

```
RJ   DOLABCN
```

Low core location (23B) CLABEL contains the current statement label justified with blank fill.

2.11 DOLAB

DOLAB is referenced by PS1CTL after each labeled statement is processed. If the label terminates one or more loops, DOLAB closes each loop i.e., it notes exits from the loop, generates an R-list macro, and compresses the DOT and DOL tables.

The calling sequence to DOLAB is:

```
RJ   DOLAB
```

Low core location (23B) CLABEL contains the current statement label left justified with blank fill.

2.12 DOLABR

DOLABR is referenced by IF, CALL, and GO TO processors when reference to a statement label is encountered. If the label is not yet defined, it is entered in the DOL table, otherwise entries and exits are noted.

The calling sequence to DOLABR is:

```
RJ   DOLABR
```

SELIST points to the E-list item for the label, and upon return B1 contains ordinal of SYMTAB entry for that label.

3.0 DOPROC will produce the following diagnostic messages:

(1) Loops are nested more than 50 deep.

(2) The terminal label of a DO must be an integer constant between 0 and 100,000.

(3) The terminal statement of this DO precedes it.

(4) The control variable of a DO or DO-implied loop must be a simple integer variable.

(5) The syntax of DO parameters must be I=M1, M2, M3, or I=M1, M2.

(6) A constant DO parameter must be between 0 and 131K.

(7) A DO parameter must be an integer constant or variable.

(8) This statement number had been used before.

(9) A previous statement in this nest references this statement number illegally.

(10) This statement references a previous statement number in this nest illegally.

(11) A DO loop may not terminate on this type of statement.

(12) A DO loop which terminates here includes this unterminated DO loop.

(13) This statement redefines a current loop control variable or parameter.

(14) ENTRY statements may not occur within the range of a DO statement.

(15) This DO loop is unterminated at program end.

(16) This loop is entered from outside its range but had no exit.

(17) This referenced statement number does not appear on an executable statement.

(18) This referenced format number does not appear on a format statement.

(19) More storage required by DO statement processor.

(20) The variable upper limit and the control variable of this DO are the same producing a non-terminating loop.

(21) Compiler error.

(22) The constant lower limit is greater than the constant upper limit of a DO.

(23) The referenced label is greater than five characters.

(24) Zero statement labels are illegal.

Errors 8,11,13,19,20 and 22 are informative.

Error 21 is fatal to compilation.

The remainder are fatal to execution.

4.0 Environment

DOPROC depends on information from the other processors in varying forms and degrees. It must have the E-list pointer for each DO statement and DO-implied loop. It files entries in the symbol table (SYMTAB) and stores both temporary and permanent information regarding them. It outputs macros and generates label items into the R-list.

Low core cells contain values used in DOPROC. These are:

(1) SELIST (32B) contains the address of the current entry in the E-list.

(2) DIM1 (17B) contains the address of the dimension table.

(3) DUKE (37B) contains the binary line count.

(4) NRLN (64B) contains the next available R number.

(5) SYMEND (13B) contains the address of the last entry in the symbol table (SYMTAB).

(6) SYM1 (12B) contains the address of the unused entry for ordinal zero in the symbol table (SYMTAB).

(7) CLABEL (23B) contains the current statement label, left justified with blank fill.

(8) TYPE (24B) contains the current statement type in binary right justified in the word.

(9) LTYPE (25B) contains the type of the statement on the true side of a logical IF.

5.0 Structure

5.1 DOTOP

DOTOP processes each DO statement and DO-implied list and generates the R-list macro words for the top of the loop.

5.2 SYNCHEK

SYNCHEK performs a syntax check on each DO STATEMENT and DO-implied list.

5.3 INTVAR

INTVAR makes an integer variable assurance check and returns a verdict of integer variable or not.

5.4 IDEF

IDEF checks for illegal definition of loop variables and makes entries in DOLIST (DOL table).

5.5 LIMIT

LIMIT converts loop limits and determines if constant, variable, or illegal.

5.6 IREF

IREF searches the DOT table to determine if the integer variable referenced in this statement is a control variable. If so, it sets flag M.

5.7 GENMAC

GENMAC generates the R-list macro words.

5.8 CHECK

CHECK checks the form of the limits and constructs symbolic-constant representation for each.

5.9 LABEL and SYMBOL

LABEL and SYMBOL search SYMTAB for the given display code entry and return I and H (R-list description) of entry and address.

5.10 ERPRO

ERPRO is used to produce a variety of error messages.

5.11 ETB

ETB converts E-list integer constant items into the corresponding binary constant.

5.12 NAME

Given the ordinal of a SYMTAB entry, NAME checks the E-field of the SYMTAB entry and if zero puts the ordinal into B1 for use by IREF. If E=1, it puts the base and bias from DIMLIST into B1 and B2 respectively.

6.0 Formats

The DO processor receives information from SCANNER in E-list format and using SYMTAB data generates R-list macros for processing by pass two of the compiler. Interim data is stored into two tables:

(1) DOTABLE (DOT) - information on current loops

(2) DOLIST (DOL) - references to undefined labels and integer variable definitions

Each table will be discussed in turn as to content, format, etc.

DOT Table - The DOT table is a rigid table of up to 51 entries consisting of one entry per DO statement. However, each entry requires 3 memory words so that 153 60-bit memory words are consumed. The first entry in the table is not used. Flag and address quantities are defined as follows:

P - an 18-bit index relating the DO statement to the machine address in the DOL table where information about the loop is stored.

CV - A three bit field containing flags b, c, and d, where B, C and D are lower limit, upper limit and increment of the DO respectively. If b, c, or d is set to 1, then the corresponding limit is variable. Otherwise, the limit is constant.

Example:

```
     DO   10   I = 1,N,2

B = 1      b = constant or 0
C = N      c = variable or 1
D = 2      d = constant or 0
```

and CV - 010 in binary

Flags - seven bits of loop description broken into seven one-bit fields.

(1) E - set to 1 if loop may be entered at a point other than the top.

(2) X - set to 1 if loop may be exited at a point other than the terminating statement.

(3) I - set to 1 if loop contains another loop.

(4) M - set to 1 if loop control variable must be materialized (placed in memory).

(5) V - set to 1 if control variable same as incremental limit. Example: DO 10 K = 1,N,K

(6) J - set to 1 if loop contains an implicit or explicit subroutine call.

(7) R - set to 1 if all integer variables are assumed to be redefined within a loop.

N - a 12-bit field containing the number of integer variable definitions.

S - A 12-bit field holding the symbol table (SYMTAB) ordinal of the statement number referred to in the DO statement.

IX- A 12-bit field holding the symbol table ordinal for the control variable entry.

L - A 12-bit field holding the symbol table ordinal of the generated label at the top of the loop.

B - An 18-bit field that is either a binary constant for the upper limit of the DO or the symbol table ordinal of a variable lower limit.

C - An 18-bit field that is either a binary constant for the upper limit of the DO or the symbol table ordinal of a variable upper limit.

D - An 18-bit field that is either a binary constant for the incremental parameter of the DO or the symbol table ordinal of a variable incremental parameter.

Information is aligned within the three words as follows:

```
VFD  12/S,18/0,12/N,18/B
VFD  12/IX,12/0,18/P,18/C
VFD  12/L,20/0,1/b,1/c,1/d,1/E,1/X,1/I,1/M,1/V,1/J,
     1/R,18/D
```

DOL list - The DOL list is a variable length table. The list has two forms of entries:

(1) References to undefined labels.

(2) Integer variable definitions.

All DOL list entries are one word (60-bits) in length and the fields are these:

(1) T - a one bit flag indicating a reference to an undefined label (0) or an integer variable definitions (1)

(2) NAME - a 30 bit field with contents in one of the three forms:

a) T = 0, NAME is the ordinal of the symbol table entry for the undefined label, right justified in the 30 bit field.

b) T = 1, E of SYMTAB = 0, then NAME is the ordinal of the symbol table entry for the integer variable definition right adjusted in the upper 12 bits of the 30 bit field (to align with the base of an equivalenced set of variables).

c) T = 1, E = 1, then name contains the base and bias of the equivalenced variable as 12 bit and 18 bit fields respectively.

The DOL list format is:

```
VFD  1/T,29/0,30/NAME

          or

VFD  1/T,29/0,12/BASE,18/BIAS
```

R-list Macros - DOPROC will generate R-list macros at the top and bottom of each loop, for processing by DOPRE in pass two of the compilation. Each macro will consist of 7 words (60 bits each). The first word is a standard macro reference with fields for OC, IN, etc. (See R-list language description). The second word has the ordinal of SYMTAB where the DO label definition is filed. This field is the rightmost 30 bits and the ordinal is right justified. Beginning with the left half of word two and continuing through word 7 is information relating to the three limiting parameters of the DO (B,C,D where the general form is DO SN I=B,C,D) and the induction variable I.

The parameters for words 2-7 are as follows:

(1) L - a 30 bit label reference of the I AND H variety described in R_LIST description. L is the label attached to the DO.

(2) SB - 30 bits of symbolic (I and H) information related to a variable B or base if variable is equivalenced to another.

(3) SC - 30 bits of symbolic (I and H) information related to a variable C or base address if variable is equivalenced to another variable. Bias appears in CC.

(4) SD - 30 bits of symbolic (I and H) information related to another variable. Bias appears in CD.

(5) SI - 30 bits of symbolic (I and H) information related to a control variable (induction variable) or base address if variable is equivalenced to another variable. Bias would appear in CI.

(6) P1 - 16 bit R number used in storing the induction value.

(7) CB - 18 bits of binary constant representing the constant B or the bias if B is an equivalenced variable.

(8) CC - 18 bits of binary constant representing the constant C or the bias if C is an equivalenced variable.

(9) CD - 18 bits of binary constant representing the constant D or the bias if D is an equivalenced variable.

(10) CI - 18 bits of bias if the induction variable is equivalenced to another variable. If variable does not appear in an equivalence statement, then CI is zero.

DPCLOSE

1.0 General Information

DPCLOSE is called to complete processing of the declaratives encountered during phase one. It will:

a. Assign addresses and allocate storage for arrays, common blocks, and equivalence classes.

b. If R=2 or 3 then format and save the information needed to produce the common-equivalence portion of the map.

c. Issue storage to COMPS, if necessary.

d. Initialize for phase two processing.

2.0 Entry points

2.1 DPCLOSE

Primary entry to initiate terminal phase one processing.

2.2 O.CBT

First word address of the saved common block table. This overlays a portion of DPCLOSE starting at (DPCLOSE + 3* maximum DO loop nesting level, presently equal to 50). Since DPCLOSE will be immediately preceeded by DOPROC, this is the method by which DOPROC's table is allocated.

2.3 DBLEPREC

This location is non-zero if double precision statements appeared in the declaratives.

3.0 Message and Diagnostics

SUBSCRIPTS FOR ADJUSTABLE DIMENSIONS MUST BE OF INTEGER TYPE

LEVEL 3 VARIABLE MUST BE IN COMMON

LEVEL EQUIVALENCE ERROR

NOT ALL ITEMS IN THE BLOCK ARE THE SAME LEVEL

CONFLICTING LEVELS IN THIS COMMON BLOCK

COMMON/EQUIVALENCE ERROR

NUMBER OF SUBSCRIPTS IS GREATER THAN THE NUMBER OF DIMENSIONS

ILLEGAL COMMON BLOCK EXTENSION

CONTRADICTORY EQUIVALENCE RELATIONSHIP

ARRAY NAME USED WITHOUT SUBSCRIPTS, FIRST ELEMENT ASSUMED

DIMENSIONAL RANGE EXTENDED BY EQUIVALENCING

4.0 Environment

Externals

PH2CTL	Phase two controller. Directs processing of all phase two statements.
PSYM	Prepare a symbol for use in an error message.
CTBLOVL	Called for storage overflow on the saved common table.
ALLAE	Allocate almost all storage to one table.
ADDWD	Add a word to a managed table.
R=FLAG	Reference map level number.
SCF	Reduces common table back to last error free segment.
CON.	Ordinal of CON. in the symbol table.
MACFLAG	Logical OR of C and E option flags.
SFF	Reduces equivalence table back to last error free segment.

INITBL	Initialize tables for a phase.
UDATA..	Use block indicator for the DATA.. block.
DFLAG	DEBUG mode flag.
SYMORD	Number of symbols in the symbol table.
LOWCORE	Holds the starting address of lowest address available for scratch storage.
N.FERR	Number of fatal errors.
WRWDS	Routine to write information to a file.
ASAER	Called to issue an ANSI severity error.
C.BLOCK	Current use block indicator.
N.FILES	Number of files on the program card.
NOGOFLG	GO/NO GO flag for debug mode.
OUTUSE	Routine to issue a USE name if it is necessary to switch blocks.
ERPROI	Called to issue informative errors.
DATA..	Holds the length of the DATA.. block.
N.FP	Number of formal parameters.
UDATA.	Use block indicator for the DATA. block.
ERPRO	Called to issue fatal errors.
ORGTAB	Base of the table of common block names and information link.
PNORD	Symbol table ordinal of the program entry point.
MOVE	Routine to move a region of core.
DINPH2	Initializes for DEBUG mode processing.
ST.	Symbol table ordinal of ST.

N.COM	Holds the number of common blocks.
WB.CON	Word B format for constants.
SYMBOL	Search and enter symbols routine.
VARDIM	Set if variable dimensions occurred.
OSC	Output storage for a block to COMPS.
F.LFN	Word B entry for file names.
ALLOC	Allocate space to a table.

5.0 Processing

5.1 General Processing

The main processing of DPCLOSE proceeds as follows. First, SCF and SEF are called to obtain error free common and equivalence tables. Initialize table space for DPCLOSE processing by calling INITBL. Next, perform error checking which had to be deferred until all declaratives were found. Assign the address for all common variables and then process the equivalence tables. Next, we assign local addresses and save common addresses, if necessary. At this point, storage is issued to COMPS for arrays and common blocks. To initialize for phase two, the common and equivalence tables are discarded if the reference map level is not 3. The local address table is unconditionally discarded. The address of the DIM table is established and the FP block length table cleared. Create the entry point table placing the main entry and any files into the table. Enter CON. into the symbol table. For debug mode, DINPH2 is called to prepare for phase two debug processing and the NOGO flag is set if fatal errors were found in phase one processing. Finally, the use block is switched to DATA. and processing exits to phase two.

5.2 PDC - Perform Deferred Checks

1. If there are no double precision arrays, proceed to section two.

Scan the DIM table and for each double precision array, which is not variable dimensioned, double the total word count in the dimension table.

2. If there are no LEVEL statements go to section three.

   If there were no common declarations go to section three. Otherwise, set the first pass flag. Using the pointers in ORGTAB to the head of each common chain we scan down the chains examining the symbol table entry for each member of the block. A zero level field implies no level declared. If a non-zero level field is found in an entry all other entries in the block with a non-zero level field must have the same non-zero value as the first one encountered. After a non-zero level is found all subsequent members of the block with no level specified are forced to the level of the declared element and an informative diagnostic issued. If two elements differ in declared level a fatal error results. If not all members were at the same level the level of the declared member is used as the level of all unspecified members. To accomplish this for unspecified members preceding a specified member a second pass flag is set and the above loop repeated.

3. If there are no variable dimensioned arrays, go to section four.

   Scan each formal parameter to determine if it is involved in variable dimensioning. This known since the RL field in word B of the symbol table will have a value of one. If a formal parameter is used for variable dimensioning and is not integer, a diagnostic is issued.

4. If there has been no equivalencing, we exit PDC. Clear out N.COM (number of common blocks) entries in the common/equivalence table (CET). Then, scan the equivalence table. If we find a member in common, set the entry in the CET to the block ordinal number (RB). If an equivalenced item has no DIM table entry, we create one and insert the pointer into word B. Next, we use CET just built to determine which common blocks are involved with equivalencing. For any such block, we must scan it and create DIM

table entries for any members not already processing them. Finally, we update the size of the DIM table and clear each new entry.

5.3 ACA - Assign Common Addresses

If there was no declared common, the routine simply exits. The body of this routine consists of a triply nested loop. The outermost loop iterates through each entry in ORGTAB. The next loop within advances through successive block name words. Finally, the innermost loop processes each member of a block group.

The outermost loop initializes a block length of zero. Next, we advance to the first block name word. For each member, we insert the current address, a dimensioned bit (if dimensioned) and a word count. The RA is placed in the DIM entry for dimensioned items. Then, the member length is added to the block length that is being accumulated. We proceed through all members and then advance to the next block name group. When all block name groups have been processed, we place the total block length in the length field of the first block member name word. In addition, bit 17 is set in the first block member name word if the block is a LEVEL 3 block.

5.4 EQV - Process Equivalence Tables

On entry, the format of the equivalence table is

```
12/2000B+n, 48/2*symbol table ordinal
3/number of subs,3/0,18/sub 3,18/sub 2,18/sub 1
```

where n = order of the name in the group.

Prescan - During the prescan, word two is reformatted as follows:

```
1/comflag,5/0,18/product of dims,18/0,18/subscript
```

where:

```
comflag   =   1 if the element is on common
SDPF      =   0 for single precision, 1 for double
subscript =   (I-1+D1*(J-1)+D1*D2*(K-1))*(SDPF+1)
```

EQV exits immediately if no equivalence statements appeared in the current routine.

a. Setup to scan the equivalence table from FWA to LWA.

b. Obtain equivalence word 1 and 2 (EW1, EW2).

c. Establish comflag bit using word A of the symbol (P.COM bit).

d. Diagnose equivalence of variables in level three.

e. Diagnose more subscripts than dimensions.

f. Extract the product of the dimensions from the DIM table for dimensional variables.

g. Compute the subscript from the DIM word two and EW2.

h. Diagnose a subscript larger than dimensional declaration as a range extension.

i. Replace word 2 of equivalence table with new form.

j. Iterate steps b-i for the entire table.

Scan 1 - During scan 1, one of the G-F tables is constructed. Its form is:

```
1/COMFLAG,5/10,18/H(I),18/0,18/R(I)
6/0,18/HP(I),18/SYMORD,18/S(I)
```

where:

H(I) = distance above the root member
HP(I) = distance below the root member
R(I) = addresss of the member I - address of root member
SYMORD = symbol table ordinal
S(I) = a link to the next entry.

a. Obtain EW1, EW2 from prescan.

b. Is this symbol table entry greater than the largest ordinal so far encountered? If so, set new largest ordinal. (This must be a first appearance in the equivalence class.) Then, go to f.

c. Search the partially constructed table to determine if this element appeared earlier in the class.

d. If it did not appear previously, go to f.

e. Compute the distance to the root member by chaining through the links and accumulating the distance. Then, go to h.

f. Construct a G-F entry for a first appearance of an ordinal.

g. H(I) is the product of the dimensions, R(I) = 0, HP(I) = 0 ORD = symbol table ordinal, S(I) = distance from head of table.

h. If this is not the first name in the group, go to j.

i. Save the current root number distance and the subscript of the root. Go to r.

j. Link this node to the root or to another node. If the distances from table base of the current root and this node are the same and the subscripts are the same, this is a redundant equivalence. If the subscripts differ, it is a contradictory equivalence relation. Go to r for a redundancy.

k. Check to see if we need to choose a new root member. If not, go to m.

l. Set the new root member distance and subscript.

m. If no previous address has been assigned to this node, go to q.

n. If the distance to the root member of the current root is zero, go to p.

o. If R (root member) does not equal R (new node) + distance from root to the new node diagnose, a contradictory equivalence relation. If the relation is valid, go to q.

p. Set R (root member) to R (current node) + distance to root.

q. Now this element is linked to the previous member R (current node) is set to distance (root current node) S (current node) is set to the root member ordinal H (root member) is set to max (H (current

node) + distance to current node, H (root)). HP (root) is set to max (HP (current node) - distance to current node, HP (root)).

r. Advance to the next entry in the prescan table. If not done, go to a.

s. Save the new length of the equivalence table.

Scan 2 - Assign a Bias to each element.

This scan will link up the elements of each class and the roots. At the end of the scan, the EQV table will be formatted as follows:

Roots: 1/COMFLAG,5/0,18/NEXTR,18/LASTE,18/BIAS
6/0,18/SPAN,18/ORD,18/S

Nodes: 1/COMFLAG,5/0,18/H,18/NEXTE,18/BIAS
6/0,18/HP,18/ORD,18/S

where:

NEXTR = ordinal of the next root
LASTE = ordinal of the last element in the class
NEXTE = ordinal of the next element in the class

Note: The fact that the first node of a class must immediately follow its root enables the linking of the equivalence classes without a pointer in the root entry.

a. Initialize the number of roots to zero and LASTR to minus one.

b. If this is not a root member, go to e.

c. Increment the number of roots. Change HP to span (HP+H). Change R field to BIAS (R+HP) using the old HP field. Clear the H field. Set LASTR to this ordinal. If this is the first time LASTR has been set, go to h.

d. Link previous root to this root by setting the NEXTR field. Go to h.

e. Update R field (bias) to be R (current node) + R (link of current node).

f. Locate the root entry for this class by following the successor vector.

g. Update the previous last element pointer in the root and link the previous node to this node.

h. Iterate through the table until done.

Scan 3 Build a binary equivalence class table from the list. Check and mark common/equivalence interaction. Locate the base member of each class.

a. Allocate space for the local address table equal to the number of equivalence classes. Allocate space for the equivalence class table.

b. Fetch a root member (EW1, EW2). Construct the ECT entry. Extract the link to the next root.

c. Obtain EW1, EW2 of the first member. Combine comflag, bias and symord into an ECT entry.

d. If the current bias is less than the lowest bias so far, make this the new minimum and potential base member. If this current bias member of the equivalence class is in a different storage level than the one with the lowest bias, issue a fatal error (LEVEL EQUIVALENCE ERROR).

e. If this member is in common, check that not more than one member of the class is in common. If so, we have a common/equivalence error.

g. Link to the next node in the class. If more members exist, go to c.

h. If the root is the base, go to j.

i. Swap the root and base members of the class.

j. Store the number of members into the base entry in the ECT. Set up the local address table entry for the class.

k. If there are more classes, go to b.

Scan 4 Set bits and fields in symbol table and dimension table. Form the LAT entries for local classes.

a. Pick up an ECT entry. If it is in common, go to k.

b. Set LAT entry to the ECT entry.

c. Update the DATA.. block with the length of the span.

d. Set word one of DIM entry to the form:

1/1,23/0,18/SPAN,18/RA

e. Establish DEF and EQV bits for word A.

f. Call DEI to distribute equivalence information.

g. If more classes exist, go to a.

h. Set the new size of the LAT.

i. If R is not three, exit from EQV.

j. Move the ECT into the EQC table area and exit from EQU.

k. If the class in common has an error, issue the message COMMON/EQUIVALENCE ERROR and/or LEVEL EQUIVALENCE ERROR.

l. If the base is not in common, swap the base and common member.

m. Extract word B bits from the common element for use by DEI.

n. Detect and diagnose an illegal extension of a common block.

o. Place the FWA and bias of class in LAT entry.

p. Update the length of the common block in the block prefix word of the common block word. Update the length of the class in the ECT.

q. Set bit 59 in the DIM table word 1 so DATA processor can tell the name is equivalenced since the EQU bit is not set in symbol table for the base member.

r. If only one member in the block, go to u.

s. Set the equivalence bit in word A for a common block member, set word one of the DIM entry to base-bias form.

t. If more members of the block exist, go to s.

u. If common headers remain to be processed, link to the next header and go to s.

v. Load up the DIM skeleton, number of members, and DEF, COM, EQU bits for a DEI call.

w. Modify the bias of each ECT entry due to class in common.

x. Go to g.

DEI - Distribute Equivalence Information

Set the bits passed in into word A of the symbol table entry for the current ECT entry. Update word A of the DIM entry to the form:

6/0,18/SYMORD,18/BIAS,18/RA

Place the word count from the DIM table into the ECT entry. Then iterate the process for all members of the class.

5.5 ALA - Assign Local Addresses

a. If the E or C option is set plus code so that no COMPS storage is issued.

b. Now allocate nearly all storage to the LAT.

c. Fetch a symbol table entry. Increment to next entry. Go to k if end of symbol table.

d. Go to c if the symbol is in common or equivalenced.

e. If it is a LEVEL 2 or 3 symbol and not in common, set an error flag. Go to c.

f. If the symbol is not dimensioned, go to c.

g. If it is a label, go to c.

h. Place the address of the DATA.. value in word one of the DIM entry. Increment the block length.

i. Save the word count from the DIM entry and the symbol table ordinal in the LAT.

j. If scratch storage is exhausted, diagnose a phase one memory overflow.

k. Update the length of the DATA.. block and set the new LAT length.

l. Diagnose all ECS variables not in common if the error flag is set. Exit from ALA.

5.6 SCA - Save Common Addresses

a. Exit if no common blocks.

b. Get the block length from the block prefix header and place it in ORGTAB.

c. Iterate b for all common blocks.

d. For R=3, modify the FWA of tables.

e. Now allocate almost all scratch space to the SCA (saved common address table).

f. For debug mode, plug the code to build the SCA table.

g. Setup to scan the common block starting with the prefix word.

h. If the common block member has a DIM entry, go to j.

i. Place RA and RL fields from the common table into word B of the symbol table entry. For debug mode, they are placed in the saved common address table.

j. If more block members remain, go to h and examine the next one.

k. Advance to the next block prefix word. If more of the common table remain, go to g.

l. Store the new length of the saved common address table and exit.

5.7 ICS - Issue Storage to COMPS

a. If E or C are not selected, exit unless LEVEL statements have appeared.

b. If no common blocks, go to k.

c. Issue a USE/block name/ or a USELCM/blockname/ to COMPS depending on the storage level.

d. If the block is a LEVEL 3 block extract the block length from the base member and issue a storage allocation to COMPS in the form nnn BSS mmmB.

e. If the block is equivalenced, output storage in the form of a BSS for the base member only. Go to g.

f. Output a name BSS nB for each member of the common block.

g. If there are more common blocks, go to c.

h. Switch to the DATA.. block.

i. Exit if no local arrays (L.LAT=0).

j. Output storage for the local arrays.

k. Exit from ISC.

6.0 Table Formats

6.1 CBT - Common Block Table.

This holds a copy of ORGTAB during phase one for use by REFMAP.

VFD 42/block name,18/index to block prefix word.

6.2 COM - Common Table.

Maintained after DPCLOSE time only for R=3.

```
VFD  1/DIM,5/0,18/WC,18/SYMORD,18/RA
```

is format for a member of a common block where:

DIM	=	1 if dimensioned
WC	=	word count of item
SYMORD	=	symbol table ordinal
RA	=	block relative address.

Each group of member words is preceded by a block header word of the form:

```
VFD  1/EQU,5/0,18/length,8/number of names
     18/link to next prefix word
```

where:

EQU	=	1 if equivalent.
length	=	length of the common block (only in the first header word).
number of names	=	number of name words following the header.

6.3 DIM - Dimension Table.

This table holds basically all information regarding declaratives subsequent to phase one. All entries are two words each.

```
VFD  1/EQU,5/0,18/SYMORD,18/BIAS,18/RA
VFD  3/NSUBS,3/PABC,18/SUBC,18/SUBB,18/SUBA
```

where:

EQU	=	1 if the entry is equivalenced.
SYMORD	=	Symbol table ordinal of the base member is equivalenced. Otherwise, the original symbol ordinal.
BIAS	=	Offset from the base member if equivalenced.

RA	=	Block relative address.
NSUBS	=	Number of subscripts.
PABC	=	3 bit field specifying variable or constant subscripts.
SUBC	=	Product of the dimensions or the third variable subscript ordinal.
SUBB	=	2nd dimension or variable subscript ordinal.
SUBA	=	1st dimension or variable subscript ordinal.

6.4 ECT - Equivalence Class Table.

Saved for REFMAP if R=3 is selected. The format is:

Base member VFD 1/COM,5/0,18/SPAN,18/SYMORD,18/number of members-1.

Other members VFD 6/0,18/word count,18/SYMORD,18/BIAS

6.5 LAT - Local Address Table.

Holds each local name for which storage must be issued to COMPS. The form is:

VFD 6/0,18/word count,18/SYMORD,18/0

6.6 SCA - Saved Common Address Table.

Holds the relocation information for each name in COMTAB which has no DIMTAB entry. This is needed only in DEBUG mode since otherwise the information can be placed in word B of the symbol table entry immediately. The information from the SCA will go into word B at the end of pass one. The format is:

VFD 6/0,18/word count,18/symord,18/RA

DECPRO

1.0 General Description

DECPRO contains processing for all Phase 1 declarative statements. Information accumulated is placed in tables containing dimension, common, and equivalence structures. These tables are processed by DPCLOSE.

2.0 Entry Points

2.1 DPCOM

DPCOM processes COMMON statements and enters information into ORGTAB and the common table (COM).

2.2 SCF - Set Common Flags

Resets flags associated with common if an error is found while processing a common block.

2.3 DPEQU

Processes the EQUIVALENCE statement and adds entries to the equivalence list for items mentioned in the statement.

2.4 SEF - Set equivalence Flag

Save the length of the equivalence table in case an error occurs.

2.5 DPTYP

DPTYP processes all FORTRAN TYPE statement. This includes LOGICAL, INTEGER, REAL, DOUBLE, COMPLEX, ECS. The specified type is placed in the symbol table.

2.6 DPDIM

Processes all DIMENSION statements.

2.7 DPLEV

Processes all LEVEL statements and assigns the variables to the declared storage level.

2.8 DPIMP

Processes the IMPLICIT statement and sets the specified implicit typings.

3.0 Diagnostics And Messages

FORMAL PARAMETER IN COMMON OR EQUIVALENCE STATEMENT

BAD SYNTAX IN TYPE STATEMENT

PREVIOUSLY TYPED VARIABLE, FIRST ENCOUNTERED TYPE IS RETAINED

SUBPROGRAM NAME MAY NOT BE REFERENCED IN A DECLARATIVE STATEMENT

COMMA MISSING BEFORE VARIABLE INDICATED

ILLEGAL SEPARATOR ENCOUNTERED

BAD SYNTAX ENCOUNTERED

ILLEGAL SEPARATOR BETWEEN VARIABLES

COMMON BLOCK NAME NOT ENCLOSED IN SLASHES

COMMON VARIABLE IS FORMAL PARAMETER OR PREVIOUSLY DECLARED IN COMMON OR ILLEGAL NAME

ILLEGAL BLOCK NAME

TOO MANY LABELED COMMON BLOCKS, ONLY 61 BLOCKS ARE ALLOWED

ECS VARIABLE MAY NOT APPEAR IN EQUIV STMT

BAD SUBSCRIPT IN EQUIV STMT

ONLY ONE SYMBOLIC NAME IN EQUIVALENCE GROUP

SYNTAX ERROR IN EQUIVALENCE STATEMENT

VARIABLE HAS MORE THAN THREE SUBSCRIPTS

VARIABLE WITH ILLEGAL SUBSCRIPTS

PREVIOUSLY DIMENSIONED VARIABLE, FIRST DIMENSTIONS WILL BE RETAINED

A VARIABLE DIMENSION OR THE ARRAY NAME WITH A VARIABLE DIMENSION IS NOT A FORMAL PARAMETER

RETURNS OR EXTERNAL NAMES MAY NOT APPEAR IN DECLARATIVE STATEMENTS

INVALID LEVEL NUMBER SPECIFIED

LEVEL CONFLICTS WITH PREVIOUS DECLARATION, ORIGINAL LEVEL RETAINED

THIS STATEMENT FORM IS OBSOLETE. USE A LEVEL 3 STATEMENT

CHARACTER BOUNDS REVERSED IN IMPLICIT STATEMENT

ILLEGAL CHARACTER BOUND IN IMPLICIT STATEMENT

ILLEGAL TYPE SPECIFIED IN IMPLICIT STATEMENT

ILLEGAL SYNTAX IN IMPLICIT STATEMENT

DECLARATIVE STATEMENT OUT OF SEQUENCE

IMPLICIT STATEMENT IS NON-ANSI

4.0 Environment

4.1 Common Blocks

/MACBUF/ contains miscellaneous scratch cells used during statement processing.

4.2 Externals.

ERPRO and ERPROI	Fatal and informative message routine
ASAER	Non-ANSI message routine
N.COM	Number of common blocks

VARDIM	Non-zero if variable dimensioning appears
CONVERT	Constant conversion routine
RSELECT	Non-zero if R=2 or 3 selected
PH1SCAN	Return point for phase one processing
ORGTAB	Table in LSTPRO holding common block information
IMPTYPE	Natural type table
N.FP	Number of formal parameters
NRB	Natural real bits (bits 58-33 corresponding to A-Z on if natural type is real
LEVEL	Non-zero if a LEVEL statement has appeared
DBLPREC	Non-zero if double precision type statement has appeared
VALUE	Ordinal of symbol holding value returned by a function
DFLAG	Non-zero if a DEBUG mode
O.COM	Origin of the common table
L.COM	Length of the common table
O.EQV	Origin of the equivalence table
L.EQV	Length of the equivalence table
Z.EQV	Number of the equivalence table
O.DIM	Origin of the dimension table
L.DIM	Length of the dimension table
Z.DIM	Number of the dimension table
NTYPE	Returns the natural type of a variable (in LSTPRO)

PSYM	Prepare a symbol for addition to an error message
CFO	Check first occurrence for debug variables
ADDWD	Add a word to a managed table
ADDREF	Collect a reference
SYMBOL	Enter a symbol in the symbol table

5.0 Processing

5.1 CDN - Check Declared Name

On entry, B6 = error number in case the E-list item is not a name. FPFLAG set by the processors and A4, A5 holding E-list pointers. On exit, the registers hold the exit condition from SYMBOL except that X6=0 for previous appearances or X6 = natural type shifted by P.TYP. SAVEB1 holds the symbol table ordinal and SNAME the E-list for the name. Processing performed is:

a. Produce an error if the item is not a name.

b. On first occurrence for debug variables, a CFO call is made to validate the context (unless this is a type statement where the check will be made later).

c. Diagnose the use of the routine name in a declarative.

d. Diagnose use of an FP conditional on the value of FPFLAG.

e. Diagnose use of symbol types other than the standard ones (RETURN names, etc.).

f. Exit if processing a type statement.

g. Exit if not external, else issue on error.

5.2 DPCOM

Initially, FPFLAG is set so that formal parameters will not be permitted in COMMON statements. The present length of common tables is saved using SCF. If the first

E-list item is a name, this must be a blank common statement so a name of blanks is generated. If it is not a name or a slash, an error is produced. Should the item following the first slash be a slash, the block is set to blank common. For a name, a check is made to guarantee seven or fewer digits and type integer, then the constant becomes the name. Any $ is removed at this point from the name (such as A2$).

ORGTAB is now searched to determine if this is the first appearance of the block. For first appearance, a check is made to ensure that more than 125 different common blocks have not been declared. If this is still within bounds, the number of common blocks (N.COM) is updated, E.ORG set (error recovery flag), and the ORGTAB entry established.

On a previous occurrence, a scan is made down the linked common table until the the last occurrence of the block is found. The RB field (common block ordinal) and var bit are formatted for use in setting word B of symbol table entries. Then, a zero word is appended to the common table. This will later contain the number of elements in this COMMON statement and a link field.

Next, the list of items in the common statement will be processed. Operations performed are:

a. Use CDN (CNAME) to check the name.

b. Set the type on first occurrence.

c. Check for previous appearance in a COMMON statement.

d. Save the symbol table ordinal in COM table.

e. Collect a reference if necessary.

f. Use DIMEN (DIMCHK), if needed, to process a dimension declaration on a COMMON statement.

g. Increment GNC (number of items in this COMMON statement).

h. Repeat a - g if the next element is a comma.

i. Install the number of items in the block header word.

j. If this was not the first appearance of the block name, set a link in the header word of the last appearance (link value is the location of this header-location of the last header).

k. Clear error recovery flags.

l. Set the common bit, and word B flags (var bit and RB field) into each symbol table entry for this block. Here, a check is made to diagnose the same variable appearing twice in one block declaration.

m. If the next E-list item is a slash, restart at the beginning of common processing.

n. Exit for an end of statement to the phase controller.

5.3 DPEQU

DPEQU performs a syntax check of the EQUIVALENCE statement and makes entries into the equivalence list. Processing is as follows:

a. Disallow formal parameters in EQUIVALENCE statements.

b. Save the current equivalence table length for error recovery.

c. Validate the next item. It must be a left parenthesis.

d. Clear the group name count.

e. Use CDN to validate the name (CNAME).

f. Set the var bit and the type flagging an error for level three items.

g. Increment the group name count.

h. Add the first word to the equivalence list and a second word of zero.

i. Collect references if needed.

j. If the item after the name is a left parenthesis, the subscript must be processed.

   (1) The next element must be an integer or octal constant that is non-zero and less than 2**17.

   (2) Increment the subscript counter and issue an error for more than three subscripts.

   (3) Save the subscript value in DIMTAB + subscript number.

   (4) Repeat a j1 to j3 if the next item is a comma.

   (5) Produce an error if it is not a left parenthesis.

   (6) Pack the number of subscripts plus the three subscript values into the record word of the equivalence list entry.

k. Repeat step e to j if the next item is a comma.

l. Issue an error if the item is not a right parenthesis.

m. Produce an error message if the group contained only one name.

n. Exit if the next item is an end of statement.

o. Repeat steps b to n if the next item is a comma; otherwise, produce an error for bad syntax.

5.4 DPTYP - Process Type Statements

Processing for all type statements is handled as follows:

a. Set FPFLAG zero so that formal parameters are allowed. If the statement is an ECS statement issue an informative error suggesting the use of the LEVEL statement.

b. For double word items, the DBLEPREC flag is set. This is interrogated in DPCLOSE.

c. Validate the name using CDN (CNAME macro).

d. If this is not the first occurrence of the name and the natural type does not match the present type, an informative message is produced indicating an attempt to retype the variable. If the statement type is ECS set the level field in the symbol table entry to three and go to g.

e. If this is a pseudo first occurrence (i.e. the symbol appeared in a debug context), move the declared type to saved natural type field in word B.

f. For a true first occurrence, the type given in the declaration (contained in ATYPE) is placed in the type field.

g. Collect references if necessary.

h. If the next element is a left parenthesis:

   (1) Check to make sure the name is not external.

   (2) For an unused debug variable, call CFO to validate the context and move the saved natural type field to the type field.

   (3) Call DIMEN to process the dimension declarations.

i. If the next item is a comma, repeat steps c to h.

j. Exit to the phase controller if the next item is an end-of-statement.

k. Otherwise, produce an error diagnosing a bad separator.

5.5 DPDIM - Process DIMENSION Statement

Processing is performed as follows:

a. Set FPFLAG so that formal parameters are permitted.

b. Use CDN to validate the name.

c. Set the natural type into word B.

d. Collect references if necessary.

e. Produce an error message if the next item is not a left parenthesis.

f. Call DIMEN to process the dimension declaration.

g. Repeat steps b to f if the next item is a comma.

h. Exit to the phase controller on an end of statement.

i. Otherwise, issue a diagnostic for bad separator.

5.6 DIMEN - Process Array Declaration

On entry, the E-list pointer denotes the item after the left parenthesis. DIMEN assumes a previous call to CDN. On exit, the E-list pointer is set for the item after the closing parenthesis. Processing is handled as follows:

a. Clear subscript counters and flag.

b. Extract the subscript element.

c. Bump the number of dimensions.

d. Flag more than three dimensions as an error.

e. For a constant:

(1) Verify that the constant is integer or octal.

(2) Convert it to binary using CONVERT.

(3) Diagnose ≤ zero or larger than machine size dimension declarations.

f. For a variable name:

(1) Get the symbol table ordinal.

(2) Diagnose the not found return since it must be a formal parameter.

(3) Test to be sure it is a formal parameter.

(4) Set the var bit and set RL=1, this plus the FP bit uniquely denotes a variable dimension value.

(5) Set a variable subscript flag.

g. Position the variable subscript flag and put the cumulative flags back in VARSUB. Save the dimension information in DIMTAB to be packed up later.

h. If necessary, set VARDIM non-zero to mark a variable dimension appearance for DPCLOSE.

i. Collect any references needed for REFMAP.

j. Repeat steps b to i if the next E-list item is a comma.

k. Issue an error if the item is not a right parenthesis.

l. Issue an error if the item was previously dimensioned stating that the previous declaration will be retained.

m. In case of variable dimensioning, check to verify that the array name is also a formal parameter.

n. Add the DIM bit and DIMP ordinal, plus VAR bit to words A and B of the symbol table.

o. Form the second word of the DIM table entry containing number of subscripts, variable or constant subscript indicator (A B C), and the A, B, and C fields.

p. Add words one and two to the DIM table and exit.

5.7 DPLEV

a. Set FPFLAG to allow formal parameters.

b. Set LEVEL to indicate occurrence of LEVEL statement.

c. Check specified level for constant value.

1. If not a simple integer constant, issue diagnostic and return.

2. Call CONVERT to get the binary value of the level number.

d. For level zero or level number greater than the maximum level allowed, issue diagnostic and return.

e. For a valid level number, save the value in LVL.

f. If expected comma is missing, issue diagnostic and return.

g. Validate variable usage using CDN (CNAME).

h. Set type or natural type as returned by CDN.

i. Compare previously declared level, if any, with current level.

j. Issue informative diagnostic if levels differ and go to stop M.

k. Store level in word B of symbol table entry.

m. Collect reference for REFMAP if needed.

n. If the next E-list element is a comma go to g.

o. If end of statement, return; otherwise, issue bad separator error.

5.8 DPIMP

a. First a check is made for the following fatal errors:

1. IMPLICIT statement after anything but a program header card or an internal debug deck. LASTTYP will hold the type of the last statement processed or the last statement processed before the internal debug deck.

2. Identifier picture length not equal to string length before left parenthesis. SCANNERs search routine is such that an IMPLICIT statement may be correctly typed even though the type specified is syntactically incorrect. If one or both of these fatal errors occurs, the diagnostics are issued, the non-ansi diagnostic is issued, and control returns to PHICTL.

b. Before processing begins the natural type table is cleared so that the implicit types can be built directly into the table.

c. The first type to be implicited is stored in ATYPE when SCANNER types the statement. Subsequent types are determined using the ELIST representation and a table of allowable types. The type currectly being worked with is kept in B1.

(0=logical, 1=integer, 2=real, 3=double, 4=complex)

d. For a given type, the individual character or range of characters are indicated by bits set in X7. Bits 58-33 correspond to A-Z. These characters are then put into the appropriate position in the IMPTYP table, depending on B1, the type. X0 holds all the bits which have been implicited so far, so that conflicts can be ddetected before adding to the table.

e. After all the types have been processed, the natural integer characters (I-N) and the natural real characters (A-H, 0-Z) are added to the table if they have not been implicited any other type (determined from X0).

f. For subroutines and functions the formal parameters must now be given a natural type based on the new natural type table. If the function name was not explicitly typed, it too must be typed with the new table. For unused debug variable the natural type is saved in the Save Natural Type field (type field holds Unused Debug Variable type).

g. Informative errors are indicated in ERRORWD. For character bounds reversed, bit 59 is set; for previously typed character, bit 58 is set. Upon completion this word is checked, and informative diagnostics, if any, are issued.

h. In case of fatal error, other than the two checked at the beginning, the diagnostic is issued. The natural type table must then be restored to its original status.

i. Before exit in any case, the non-Ansi diagnostic is issued.

6.0 Tables

6.1 COMMON Table (COM)

6.1.1 ORGTAB

Each entry in ORGTAB reflects a common block and common block ordinals are relative to the position of the block in ORGTAB. The format of an ORGTAB word is:

```
VFD  42/7L block name, 18/d
```

where d is the distance of the first block header word from O.COM.

6.1.2 COM - Common Table

The main portion of the common table is a series of linked blocks. Each block reflects the mention of a block name at the FORTRAN level followed by a list of items. The format produced by such a group is the group header word followed by variable words. For the header word, the format is:

```
VFD  24/0, 18/number of names, 18/link
```

link = the number of words to the next block header or zero if this is the last one

number of names = number of individual variable words to follow

Individual variable words are formatted as follows:

```
VFD  24/0, 18/symbol table ordinal, 18/0
```

6.2 EQV - Equivalence Table

Each item in an equivalence list has a two word entry in the table. The format of this entry is:

```
VFD  12/2000B + ord, 48/2*symbol table ordinal
VFD   3/# of subscripts, 3/0, 18/subC, 18/subB,
      18/subA
```

where ord is ordinal of the name in the group (1,2,...).

6.3 DIM - Dimension Table

Each dimension table entry contains two words per item. The format of word one is:

```
VFD  6/0, 18/symbol table ordinal, 36/0
```

and word two:

```
VFD  3/# of subscripts, 3/ABC, 18/C, 18/B, 18/A
```

where the ABC field has bit 2 set if dimension 1 is variable and clear for a constant dimension, bit 1 on for dimension 2 variable and clear for constant, bit 0 on for dimension 3 variable and off for a constant. A contains the dimension value for the first dimension, if it is constant, or the symbol table ordinal of the variable dimension. B is the same as A except for dimension two. C holds the total array size for a constant array or the symbol table ordinal of the third dimension if it is variable.

PH1CTL

1.0 General Information

Task Description

The phase 1 controller processes the header card and controls processing of all declarative statements.

2.0 Entry Points

2.1 PH1CTL

This is the main entry point of PH1CTL. It is entered from the loader.

2.2 PH1SCAN

This entry point is from a declarative routine, such as DPCOM, to look for the next statement.

3.0 Diagnostics and Messages

3.1 Fatal to Compilation

None

3.2 Fatal to Execution

a. HEADER CARD SYNTAX ERROR

1. No '(' after program name with parameters.
2. A name or constant does not follow the '=' after a file name.
3. Specified buffer length is not an integer.
4. ')' missing after file names or parameters.
5. No EOS after the ')'.
6. No program name on header card.

7. Formal parameter is not a variable name.

8. RETURNS, expected after a ',' after subroutine parameter list, is missing.

b. NUMBER OF FILES EXCEEDS MAXIMUM.

c. DUPLICATE FILE NAME.

d. EQUIVALENCE ERROR, FILE IS NOT IN SYMBOL TABLE OR A FILE IS EQUIVALENCED TO ITSELF.

e. CONFLICTING USE OF NAME IS EXTERNAL STATEMENT.

f. SYNTAX ERROR ON EXTERNAL STATEMENT.

g. RETURNS LIST ERROR.

1. No '(' after RETURNS.

2. RETURNS parameter not a variable name.

3. No ')' after RETURNS parameters.

3.3 Informative

a. No program card.

b. Declared buffer size exceeds maximum length.

4.0 Environment

4.1 Low core cells

SYM1 (12B)	Starting address of symbol table
SYMEND (13B)	Address of last word of symbol table
TYPE (24B)	Type code of current statement
SELIST (32B)	Address of next E-list element
ATYPE (51B)	Type of function
PROGRAM (56B)	Type of program - main, subroutine, function, block data

4.2 Common Blocks

//	One word block containing base address for referencing DEBUG tables
DBGBLK2	Used by DEBUG
DBGBLK1	Used by DEBUG to denote DEBUG options
NONFTNX	Used by DEBUG
MACBUF	Buffer used by SVARG to save macro calls and FMAC to format them

4.3 Externals

ADDREF	Code block in PS1CTL to collect a reference for REFMAP.
ASAER	Code block in ERPRO called to issue NON-ANSI usage diagnostic
BTOCT	Code block in ENDPRO to convert binary to octal
CAFLAG	Flag which is set to zero if FAX is to be used
CFO	Code block in DBGPHCT called to check DEBUG usage of variable names with actual program usage
CODE.	Cell in LOSTPRO containg length of the code block
COMPMSG	Cell in FTN, address of the message area
CONVERT	Code block called to convert a constant and/or to place a constant in the CON. table
DBGEPKT	Code block to process the external DEBUG packet
DBGINT	Code block to process interspersed DEBUG statements
DBGINTX	Alternate entry point to DBGINT to process an interspersed DEBUG statement when it follows an unrecognized standard FORTRAN statement
DBGIPKT	Code Block to process the internal DEBUG packet

DFLAG	Non-zero if in debug mode
DPCLOSE	Code block to which control is transferred when an executable FORTRAN statement is encountered after the header card and all declaratives have been processed
DPCOM	Code block to process the FORTRAN COMMON statement
DPDIM	Code block to process the FORTRAN DIMENSION statement
DPEQU	Code block to process the FORTRAN EQUIVALENCE statement
DPIMP	Code block in DECPRO to process IMPLICIT statement
DPLEV	Code block in DECPRO to process LEVEL statement
DPTYP	Code block to process the FORTRAN TYPE statement
ECGS	Code block to enter a symbol in the SYMBOL table and set the define bit
ENTRY.	Cell in LSTPO containing the symbol table ordinal for ENTRY.
ENTRY.D	Cell in ERPRO containing the base and relocation address for entry point of program
ERPRO	Code block called to issue fatal error diagnostics
ERPROI	Code block in ERPRO called to issue informative diagnostics
FATALER	Code block in ERPRO to issue fatal to compilation errors
FMAC	Code block in STMTP to create macro calls to the COMPS file
FORMAT	Code block to process the FORTRAN format statement

FP.	Cell in LSTPRO containing the symbol table ordinal for FP.
FTNEND	Code block which is called if an EOR is encountered on the first card and N.ERROR is zero
FWAWORK	Cell in LSTPRO containing the first word address of working storage
F1AMAC	Code block in STMTP to perform macro calls to the COMPS file with one argument
F.LFN	Cell in FTN whose bits gives information about the type file
INFORM	Cell in ERPRO which contains the address in ERPRO of the routine to issue NON-ANSI diagnostics
INITBL	Code block which initializes tables for phase1
LASTTYP	Cell to save last statement TYPE and ATYPE
LFER	Cell in LSTPRO which contains a jump to FATALER in ERPRO
MACFLAG	Cell in FTN, if zero doesn't send macros to COMPS
MSG=	Code block to issue message to the dayfile
NASAFLG	Cell in FTN, if non-zero flag on ANSI usages
N.ERROR	Cell in LSTPRO which contains the number of errors
N.FILES	Cell in LSTPRO which contains the number of files for a main program
N.FP	Cell in LSTPRO which contains the number of formal parameters in an argument list
OLIST	Cell in FTN, if non-zero, then produce an object listing
OUTUSE	Code block in LSTPRO to output a USE name to the COMPS file

O.LCC	Cell in SCANNER which contains starting address of LCC directives
PLIMIT	Cell in FTN to hold print line limit
PNORD	Cell in LSTPRO which contains the ordinal of the name used as the entry point
RSELECT	Cell in FTN, which indicates R=2 or 3 selected
SCANNER	Code block to obtain the type of the next statement
START.	Cell in LSTPRO containing the length of START. block
ST.	Cell in LSTPRO containing the symbol table ordinal for ST.
SUPIDFL	Cell in FTN, if zero, informative diagnostics are printed
SVARG	Code block in STMTP which saves the argument used in constructing a macro to the COMPS file
SYMBOL	Code block in LSTPRO called to make a new entry or search for an exisiting entry in the symbol table
TEMPAO	Cell in LSTPRO for temporary storage of A0
TRACE.	Cell in LSTPRO which contains the symbol table ordinal for TRACE.
TYPFLAG	Cell in SCANNER indicating if a bad DEBUG statement was found
UCODE.	Cell in LSTPRO containing the name CODE. in shifted form
UDATA.	Cell in LSTPRO containing the name DATA. in shifted form
UFLAG	Cell in FTN, non zero if E option is selected
USTART.	Cell in LSTPRO containing the name START. in shifted form

VALUE. Cell in LSTPRO containing the symbol table ordinal for VALUE.

WB.ESS Word B of symbol table for special SYMBOLS

WB.LFN Word B of symbol table for file names

WORDY Cell in SCANNER containing the number of words of LCC directives

WRWDS Code block in FTN called to perform the writing of R-list macros to the R-list file

XFRNAME Cell in LSTPRO containing the transfer name

5.0 Processing

5.1 DPEXT

a. Get next E-list element. If it is not a name, issue a diagnostic and return.

b. If it is a name, enter it in the symbol table if not already in.

c. If the symbol was referenced before, check that it is not the routine name (ordinal 1), return, namelist, entry point, file name or a local variable.

d. If this is the first occurrence check for use as a debug variable and enter type in the symbol table.

e. Set the external bit if there is no conflict in the name usage.

f. Collect a reference for REFMAP if necessary.

g. Exit for EOS; repeat processing, steps a-g, for comma; for any other ELIST element, issue diagnostic and return.

5.2 PHISCAN - Main loop for declarative processing

a. Set up LASTTYP for DPIMP based on the last non-debug statement type saved in TEMPB7.

b. Terminate line of references for R=1, R=2, R=3.

c. If in debug mode and the next statement is a debug statement, call DBGINT to process it. Return to step d when it is sensed that the next statement in non-debug. If not in debug mode or next statement non-debug, go to d.

d. Type the next statement via SCANNER. Save ATYPE in bits 37-20 and TYPE in bits 17-0 or TEMPB7.

e. If the statement type returned is debug (only if debug card after unrecognized statement) call DBGINT to process it. DBGINT is entered through DBGINTX because SCANNER has already processed the statement.

f. Skip over a bad debug card (TYPE=0, TYPEFLAG 0). Go to step g for a program card (TYPE=0, TYPFLAG=0). Reset TYPEFLAG to 0.

g. If the statement is a header card issue a diagnostic. If it is an executable statement go to DPCLOSE to end Phase 1 processing. Otherwise, go to the appropriate routine to process the statement.

5.3 PH1CTL

The routine is entered at this point from the SCOPE loader. If control is returned from COMPASS, the card image is placed one character per word in a 80 word buffer in SCANNER. Flags are set in ERPRO if non ANSI usages are to be flagged and/or informative diagnostics are to be produced.

If the program is in DEBUG mode, a call to DBGEPKT to process the debug package and return the program card. If not, in DEBUG mode SCANNER is called to get the first card.

If the header card is either a program, block, data, subroutine or function card, control is transferred to the appropriate routine. If an EOR is encountered on the first card and no errors have been encountered, control is transfered to FTNEND, else a dummy program card, 'PROGRAM START. (INPUT,OUTPUT)' is inserted and control is transfered to process the program card.

5.4 Program Card

The name is entered in the symbol table, if the SYSEDIT=IDENT option is used then a $ will be appended to the name entered in the symbol table. The file names are entered in the symbol table and at the same time they are placed in scratch table at O.LFN format of scratch table.

REGULAR ENTRY 24/0, 18/ORDINAL, 18/BUFFER +FET SIZE

EQUIVALENCE ENTRY 1/1,23/0,18/ORD(LFN1),18/ORD(LFN2)

If the buffer size is declared, then this value will be entered in the table instead of the default of 2000(octal) words.

After all file names have been initially processed, the RA field is computed and stored in the symbol table for each file. 'FILE' or 'FEQU' macro calls are placed on the COMPS file for each file. The TRACE macro, PENTRY macro, and 'RJ Q8NTRY'. are placed on the COMPS file.

5.4.1 SUBROUTINE - DPSUB

a. Call PPN to enter subroutine name in the symbol table.

b. Set PROGRAM to 2001BS48+ no. of arguments.

c. Call PPL to process the parameter list.

5.4.2 FUNCTION - DPFUN

a. Call PPN to enter function name in the symbol table.

b. Save type of function plus one in VALUE.

c. Set PROGRAM to 2002BS48+ no. of arguments.

d. Call PPL to process the parameter list.

5.4.3 BLOCK DATA - DPBDA

a. Set PROGRAM to zero.

b. Set name to BLKDAT. if no name is given.

c. Call PPN to enter the name in the symbol table with type T.CGS.

5.5 PROGRTN

a. Enter ST. in the symbol table and save its ordinal and, if a function, set type and variable bit in word B of symbol table for the entry of VALUE.

b. If a program, add Q8NTRY. to symbol table.

c. Call INITBL to initialize tables for phase 1.

d. If formal parameters, enter FP. in the symbol table.

e. If anything was saved, restore statement type and pointer, then, go to PH1S1 to process the statement.

f. If in DEBUG mode, call DBGIPKT to process internal debug packet.

g. Go to PH1SCAN.

5.6 PROCESS PROGRAM NAME - PPN

a. If next element in E-list is not a name, pass control to ERPRO.

b. Enter name in symbol table and set type.

c. Store name in IDENT+1 and COMPMSG+1.

d. If deck option is set, write * DECK card to COMPS file.

e. Send 'COMPILING program name' to B display.

f. Write IDENT and XTEXT cards to the COMPS file.

g. If FAX is not being used and no object listing is requested, write a LIST -L, -R card to COMPS.

h. Send any LCC directives to the COMPS file.

i. Set relocation base.

j. Collect reference to routine name if R>0.

k. Return to caller.

5.7 PROCESS PARAMETER LIST - PPL

a. Enter parameter in symbol table. If the name is already in symbol table, flag name as being doubly defined.

b. If number of formal parameters has exceeded maximum number, call ERPRO.

c. If R=2 or 3, add parameter to references for map; continue a, b, and c for all formal parameters.

d. Call PRP to process return parameters if present.

e. Call ESF to enter special symbols.

f. Output traceback and entry point information to the COMPS file.

g. If formal parameters exist, output FORPAR macro to establish the order of the F.P. blocks.

5.8 PROCESS RETURNS PARAMETER LIST - PRP

a. Check for incorrect syntax and for function subprogram. In either case issue a diagnostic.

b. Enter returns parameter in the symbol table and set type to RETURNS.

c. Flag duplicate parameters.

d. Collect references for reference map, if needed.

e. Repeat steps B-D for all parameters until ") terminates the list.

6.0 Tables

6.1 Scratch table for files on program card

6.1.1 Initial entries

```
VFD  24/0,18/Program Ordinal,18/Buffer Length
```

6.1.2 Equivalence entries

```
VFD  6/0,1/1,17/0,18/ORD(LFN1),18/ORD(LFN2)
```

CODE GENERATION TECHNIQUE

The method used for code selection in the FORTRAN Extended compiler can best be explained with an example. Consider the following FORTRAN source statements:

```
      IF  (ATAG  .LT. B*B)  CALL COMET
      PSI=(B+ACT  (N)) **2+RHO*SIGMA (N)
      K=N+K
      QTAB (N)=XTAB (I)/RHO+PSI
4     TAB2 (2,2*K)=PSI+ATAN (RHO)
```

Statements within the bracket constitute a flow block or sequence. Initially, these statements are analyzed and converted to a register free notation called R-list which would appear as:

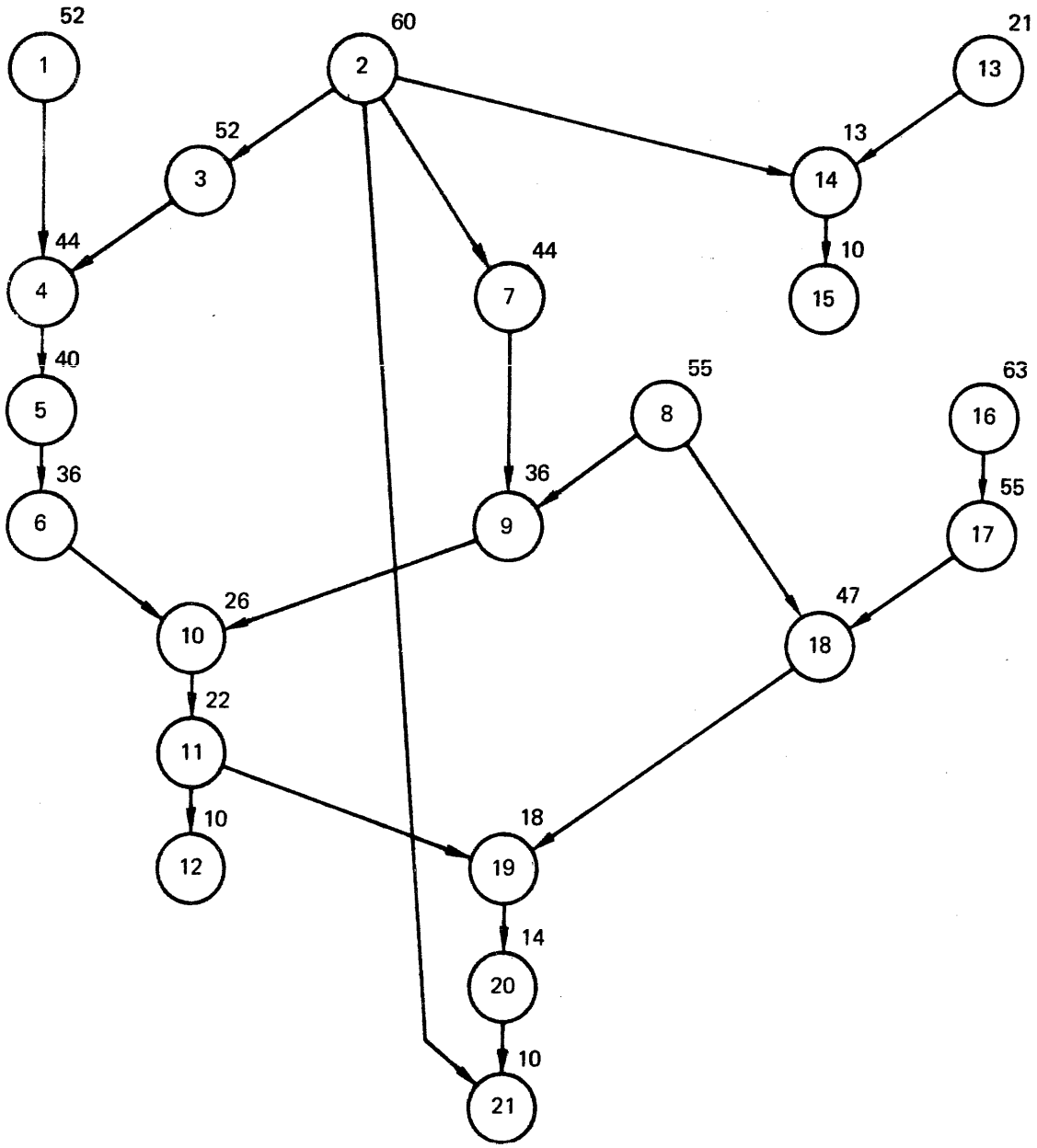
```
R1◄B              R11 = R6+R10          R19 = R17/R18
R2◄N              R12 = N (R11)         R20◄PSI
R3◄ACT-1,R2       R12►PSI               R21 = R20+R7
R4 = R1+R3        R13◄N                 R22 = N(R21)
R5 = N(R4)        R14◄K                 R23◄N
R6 = R5*R5        R15 = R13+R14         R22►QTAB-1,R23
R7◄N              R15►K
R8◄SIGMA-1,R7     R16◄I
R9◄RHO            R17◄XTAB-1,R16
R10 = R8*R9       R18◄RHO
```

(N(Ri) indicates the normalization of the result of a floating add or subtract, left arrows are loads and right arrows are stores.)

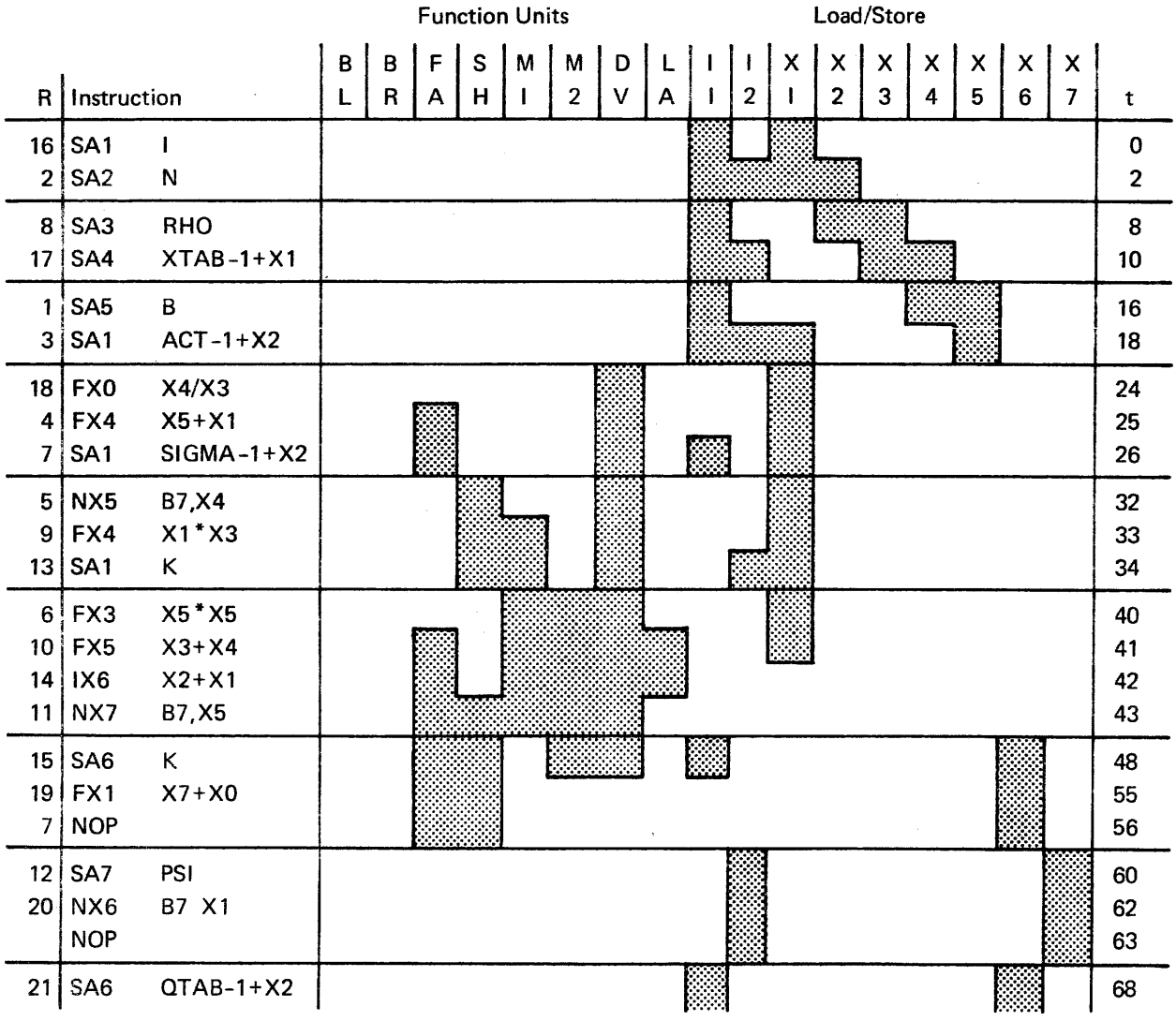
The generated R-list is then scanned and common suboperations are eliminated resulting in the squeezed R-list.

```
R1←B              R10 = R8*R9          R17←XTAB-1,R16
R2←N              R11 = R6 + R10       R19 = R17/R9
R3←ACT-1,R2       R12 = N(R11)         R21 = R12 +R19
R4 = R1 + R3      R12→PSI              R22 = N(R21)
R5 = N(R4)        R14←K                R22→QTAB-1,R2
R6 = R5*R5        R15 = R2 + R14
R8←SIGMA-1,R2     R15→K
R9←RHO            R16←I
```

From the squeezed list a PERT-like network, the following dependency tree, is formed showing the precedence of operations.


52
1
60
2
21
13
52
3
13
14
44
4
44
7
10
15
40
5
55
8
63
16
36
6
36
9
55
17
26
10
47
18
22
11
10
12
18
19
14
20
10
21

The numbers within the circles at the nodes are keyed to the squeezed R-list. The time in machine cycles required for each operation is known. From this information, the latest time at which each operation must begin in order to finish executing the network in the minimum amount of time is calculated. This is done assuming no conflicts of any kind and parallel instruction issue as well as execution. These times called priorities, are the numbers shown next to each circle; in a PERT sense they correspond to negative late start times with the network being completed at time zero. Code is generated beginning with the highest priority entry in the squeezed R-list noting which function unit is used and for how long. For all later instructions, it is required that the preceding operations have been issued and there are no function unit or register conflicts; for this purpose, a picture of the status of all registers and function units must be maintained. Using this approach, the code shown on the next page is produced resulting in the indicated overlap of operations.


Function Units
Load/Store
B L
B R
F A
S H
M I
M 2
D V
L A
I I
I 2
X I
X 2
X 3
X 4
X 5
X 6
X 7
t
16 SA1 I
2 SA2 N
8 SA3 RHO
17 SA4 XTAB-1+X1
1 SA5 B
3 SA1 ACT-1+X2
18 FX0 X4/X3
4 FX4 X5+X1
7 SA1 SIGMA-1+X2
5 NX5 B7,X4
9 FX4 X1*X3
13 SA1 K
6 FX3 X5*X5
10 FX5 X3+X4
14 IX6 X2+X1
11 NX7 B7,X5
15 SA6 K
19 FX1 X7+X0
7 NOP
12 SA7 PSI
20 NX6 B7 X1
NOP
21 SA6 QTAB-1+X2
0
2
8
10
16
18
24
25
26
32
33
34
40
41
42
43
48
55
56
60
62
63
68

CLOSE2

1.0 General Information

Task Description

CLOSE2 is the end of pass 2 processor. It closes out the COMPS file by issuing storage for statement, DO, optimizing and intermediate temporaries. It also issues code to terminate the formal parameter substitution lists. The processor also adjusts the variable dimension table and pointers for the assembler, calls REFMAP to produce the reference map, and calls FTNXAS to assemble the COMPS file. Exit is made to FTNEND if the input file is empty, or to LDRPH1 if there are more program units to be compiled.

2.0 Entry Points

2.1 BTDIS

This entry point is called to convert a binary number to display code.

2.2 CLOSE2

CLOSE2 is entered by a jump from APLIST to close out pass 2 processing.

2.3 IEM

IEM is entered by a return jump to issue the error message to the dayfile.

2.4 INIT

INIT is entered by a jump from the overlay loader after the pass 2 overlay has been loaded.

2.5 NFPUNT

NFPUNT is entered by a return jump to force outputting an informative optimization diagnostic.

2.6 PUNT

PUNT is entered by a return jump to force outputting of a fatal memory overflow diagnostic.

2.7 SDATA.

SDATA. contains the saved length of the DATA. block.

3.0 Diagnostics

3.1 Fatal to Execution

If compilation cannot be continued due to insufficient memory, the diagnostic message "PASS 2 MEMORY OVERFLOW IN - XXXXXX" is written to the output file, where the X's represent the routine in which the overflow occurred.

3.2 Informative

If better code optimization could have been performed if more memory had been provided, the diagnostic message "MORE MEMORY WOULD HAVE RESULTED IN BETTER OPTIMIZATION" is written to the output file.

4.0 Environment

CLOSE2 resides as the first routine of the (1,2) overlay.

4.1 Low core cells

N.LRB (7B) number of local relocation blocks
SYM1 (12B) first word address of symbol table
PROGRAM (56B) program/subroutine indicator

4.2 Common Blocks

/TABLES/

BLKCOM	address of BLKCOM in ORGTAB
L.PROG	program length
O.LRB	first word address of local relocation base

5.0 Structure

INIT is entered from the overlay loader after the (1,2) overlay has been loaded. If no fatal errors were detected during pass 1 processing, a jump is made to PRE to begin processing the R-list file. Otherwise, an error message is issued to the dayfile indicating the number of fatal errors encountered. If the debug mode of compilation has been selected and the NOGO option was not turned on, processing continues with a jump to PRE. Else pass 2 processing is closed out, the reference map is issued, and control is transferred to FTNEND or LDRPH1 depending upon whether the input files were empty or not.

When CLOSE2 is entered at the end of pass 2 processing, storage is issued for compiler generated temporaries, parameter substitution lists are terminated, the VDTAB table is adjusted, and the COMPS file closed out. The reference map is issued to the list file. If LEVEL statements appeared in the source program the symbol table must be searched and the RA, RL and RB fields modified so that they point to the CM pointer word for the LCM/ECS item instead of the LCM/ECS address. The FTNXAS assembler is called to assemble the COMPS file if the COMPASS assembly option has not been selected. Program control is then transferred eiher to FTNEND or LDRPH1 to terminal or continues the compilation process.

FORTRAN EXTENDED ASSEMBLER

1.0 General Information

Task Description

The FORTRAN Extended assembler FTNXAS replaces COMPASS as the assembly pass of FORTRAN Extended Version 3.0. It is a one pass assembler designed specifically to increase compilation speed. It accepts a formatted subset of the COMPASS assembly language and produces binary relocatable subprograms. All information required to facilitate a one pass assembly is gathered during the previous two passes of the compiler.

2.0 Usage

Entry Points

FTNXAS

The assembler is entered by a return jump from CLOSE2.

3.0 Diagnostics

3.1 Fatal to Execution

3.1.1 ILL

The word ILL appearing to the left of the source line on the assembly listing means the assembler could not recognize the statement or encountered an ERR pseudo-op. When this occurs, assembly is abandoned, the fatal to execution flag is set, the source line is printed regardless of the "O" option, and assembly proceeds with the next statement. The fatal to execution flag causes the message "FTNX ERRORS" to be written on LGO in place of the normal relocatable subprogram. Note that since the code produced by the first two passes is assumed to be correct, minimum error checking is designed into the assembler. The usual response to an error in the source string will be this diagnostic. Abnormal termination of the job may occur during compilation because various

unused jump vectors in the assembler are used for storage of unrelated code.

3.1.2 SYMBOL ERR

This message appears to the left of the source line if the assembler was unable to find a symbol in the two word symbol table. This error causes the same action as described for ILL.

3.1.3 STORAGE OVERFLOW

This message is printed on a separate line when the assembler has expended all available working storage for chained common and external reference information. Assembly will proceed to count the increase in field length required, but no relocatable binary deck will be produced. The message "INCREASE FIELD LENGTH BY XXXXXX" is printed after the END statement of the subprogram.

3.2 Informative

None.

4.0 Environment

The assembler may reside after CLOSE2 in the (1,2) overlay, as a separate overlay.

4.1 Input String

The input string consists of COMPASS language card images constructed using the following rules.

Executable Instructions

1. The label field must be blank, except the forcing characters + (upper) and - (lower) may be used in column 1.

2. The operation code must begin in column 3.

3. There must be only one space between the operation and address fields.

4. Following is a list of executable instruction mnemonics and address fields that FTNXAS can recognize, followed by the binary produced.

Opcode	Address	Binary in Octal	Other Action
RJ	Symbol	0100000000	External relocation noted; force upper next instruction.
JP	B1,Symbol	0210XXXXXX	Program relocation noted; force upper next instruction.
JP	B1	0210000000	
ZR	Xj,Symbol	030jXXXXXX	Program relocation noted.
NZ	Xj,Symbol	031jXXXXXX	Program relocation noted.
PL	Xj,Symbol	032jXXXXXX	Program relocation noted.
NG	Xj,Symbol	033jXXXXXX	Program relocation noted.
EQ	Symbol	0400XXXXXX	Program relocation noted; force upper next instruction
EQ	Bi,Bj,Symbol	04ijXXXXXX	Program relocation noted.
NE	Bi,Bj,Symbol	05ijXXXXXX	Program relocation noted.
GE	Bi,Bj,Symbol	06ijXXXXXX	Program relocation noted.

Opcode	Address	Binary in Octal	Other Action
LT	Bi,Bj,Symbol	07ijXXXXXX	Program relocation noted.
BXi	Xj	10ijj	
BXi	Xj+Xk	11ijk	
BXi	Xj-Xk	12ijk	
BXi	Xj*Xk	13ijk	
BXi	-Xk	14ikk	
BXi	-Xk+Xj	15ijk	
BXi	-Xk-Xj	16ijk	
BXi	-Xk*Xj	17ijk	
LXi	jkB	20ijk	
LXi	kB	20i0k	
AXi	jkB	21ijk	
AXi	kB	21i0k	
LXi	Bi,Xk	22ijk	
AXi	Bj,Xk	23ijk	
NXi	Bj,Xk	24ijk	
UXi	Bj,Xk	26ijk	
PXi	Bj,Xk	27ijk	
Fxi	Xj+Xk	30ijk	
Fxi	Xj-Xk	31ijk	
DXi	Xi+Xj	32ijk	
DXi	Xi-Xj	33ijk	
IXi	Xj+Xk	36ijk	

Opcode	Address	Binary in Octal	Other Action
IXi	Xj-Xk	37ijk	
FXi	Xj*Xk	40ijk	
DXi	Xj*Xk	42ijk	
MXi	jkB	43ijk	
MXi	kB	43i0k	
FXi	Xj/Xk	44ijk	
NO		46000	
Sri	Aj+A.E.	g0ijXXXXXX	Relocation noted.
Sri	Bj+A.E.	g1ijXXXXXX	Relocation noted.
Sri	A.E.	g1i0XXXXXX	Relocation noted.
Sri	Xj+A.E.	g2ijXXXXXX	Relocation noted.
Sri	Xj+Bk	g3ijk	
Sri	Xj	g3ij0	
Sri	Aj+Bk	g4ijk	
Sri	Aj	g4ij0	
Sri	Aj-Bk	g5ijk	
Sri	Bj+Bk	g6ijk	
Sri	Bj	g6ij0	
Sri	Bj-Bk	g7ijk	

where:

1) If r=A, then g=5; if r=B, then g=6; if r=X, then g=7.

2) A.E. is an address expression consisting of octal constants and/or symbols in the same relocation base separated by plus (+) or minus (-) symbols.

Pseudo-Ops

1. Any pseudo-op may be labeled.

2. Pseudo-ops are free field except labels, if present, must begin in column 1.

3. Following is a list of permissible pseudo-ops, their forms, and the results.

Label	Opcode	dress	Action
Required	BSS	Octal constant	a) Force upper. b) Define the label. c) Increment the origin counter by the value of the octal constant.
Optional	DATA	Octal constant	a) Convert the octal constant to binary and write it out. b) Define the label, if present.
Optional	DIS	n, character string	a) Define the label, if present. b) Write n words with blank fill.
Optional	EQU	Anything	a) Ignored.
None	USE	name or /name/	a) Change the origin counter to that of the block indicated.

Label	Opcode	dress	Action
Optional	VFD	1) n/octal constant. 2) n/address expression, 18≤n≤60. 3) (n*6)/(M)(C,H, L or R). Character string, 1≤n≤10,0≤M≤10. 4) Any combination of the above separated by commas. 5) The total bits specified must be 15, 30, or 60. 6) Any relocatable quantity must be in the lower 18 bits of the generated field.	a) The specified data is converted to binary and written. Relocation is noted.
	END	(must start in Cols 3-8) Symbol or blank	a) Terminate assembly and return.
	IDENT	Anything	a) Causes assembler initialization to take place and assembly to begin.
	LIST	Anything	a) Ignored.

Macro Calls

1. Macro calls for SUB, DELAY, FILE, and ENTR must have exactly one space between the call and the parameters. NAME and ADDSUB are free field.

2. Below is a list of macro names the assembler will recognize. See Section 5 of this document for the macro prototype description and the generated code.

Label	Macro Name	Parameters	Comments
None	ADDSUB	FP	
None	SUB	FP, CON	
None	DELAY	FP	
None	FILE	LFN,NAME,NQENT	
None	FORPAR	X	FORPAR calls are ignored by FTNXAS.
None	NAME	N,T,BASE,BIAS, FP,D1,D2,D3	
None	ENTR	NAME	
None	FMT	label	
None	TRACE	name,address,nargs	
None	PENTRY	name,lname	
None	FEQU	lfn1,lfn2,no ent	
None	GNAME	name	
None	ORG	name,bias,f	
None	REPI	DLEN,RC,INC,DESTIN	
None	HOL	string	
None	APL	name,bias	
None	EIO	value	
None	IOM	base,bias,type,count,B59,B57,base2,	

4.? Two Word Symbol Table

Active statement labels, entry points, and the labels DO., ST., and OT. are defined by a block relative address and a relocation base indicator. The length of the relocation base associated with each formal parameter is also held in this table.

4.3 Actual Parameter, Variable Dimension and Generated Label Tables

Three tables contain respectively actual parameter, variable dimension, and generated label definitions relative to the CODE. relocation base.

4.4 ORGTAB

A table of one word entries which is used to pass to the assembler the names and lengths of the common relocation bases and the lengths of the local relocation bases, START., ENTRY., VARDIM., CODE., DATA.., and HOL..

4.5 APTAB, VDTAB, GLTAB

Set in ORGTAB to the first word address of the AP, VD, and GL tables.

4.6 Object Listing Option

OLIST to indicate the "O" option has been selected.

4.7 SYM1, SYMEND

RA+12B and RA+13B point to the first and last entries of the two word symbol table respectively.

4.8 U.LGO

Name used in referencing the binary file.

4.9 ILLFLAG

Set to non-zero if a fatal to execution or compilation error has occurred before entry to the assembler.

4.10 COMPS File Structure

Due to the specialized nature of the FTN internal assembler, the structure of line images on the COMPASS file must conform to certain restrictions as to the columns the location, opcode and address fields are placed in and the order of the cards. The order of the cards on the COMPS file is expected to follow these conventions:

IDENT NAME

```
FTNMAC    XTEXT          - ignored
          LCC       XXX(YYY,m,m) - optional
          USE START.          -
```

PROGRAM

```
FILE and FEQU       macro calls one per file name
FILES.    BSS 0B
          FLINK     macro calls for each file
          DATA 0B
          TRACE     macro call
          PENTRY    macro call
          USE CODE.
          SA1 FILES.
          RJ Q8NTRY.
          rest of program
          END XFRNAME
```

SUBROUTINE or FUNCTION

After the 'USE START.' card the following will appear:

```
          TRACE     macro call
          PENTRY    macro call
          FORPAR    macro calls, one for each formal
                    parameter
          rest of program
          END
```

FAX assumes that the above mentioned cards are precisely in the order specified.

5.0 Major Subroutines and Logical Sections

1. PIDENT - initializes the assembler for each program.
2. BUILDOT creates a 22 word ORGTAB entry.
3. MR.CLEAN - removes blank fill from a symbolic name.
4. INITL(NOBINPO) - moves the next source image to ILINE from the COMPS buffer. Controls reading the COMPS file.
5. START - controls analysis of the label field of each source statement.
6. COL1VEC determines the contents of the label field.
7. BLANK - processes a blank label field.
8. PFC - processes forcing characters + and - in the label field.
9. PLABEL - processes a symbolic name in the label field.
10. OCSCAN - controls opcode field analysis.
11. Opcode transfer vector decodes the contents of the opcode field.
12. L3.RJ - produces a binary instruction for RJ.
13. L3.JP - produces a binary instruciton for JP.
14. L3.XJP - produces binary instructions for ZR, NZ, PL and NG.
15. L9.EQ - produces a binary instruction for EQ.
16. L3.BJP - produces binary instructions for NE, GE, and LT.
17. L3.BOOL - produces binary instructions for all BXi.
18. L3.MX, L3.SH - produces binary instructions for shift unit instructions MXi, LXi, and AXi.

19. L3.PUN - produces binary instructions for NXi, UXi, and PXi.

20. L3.ARIT - produces binary instructions for FXi, DXi, IXi.

21. L3.VFD - translates the address field of VFD pseudo-op to binary.

22. PDIS - processes the DIS pseudo-op.

23. PDATA - converts the address field of the DATA pseudo-op to binary.

24. PBSS - processes the BSS pseudo-op.

25. PUSE - processes the USE instruction.

26. USENXT - changes relocation bases.

27. USESTAR - returns to the previous relocation base.

28. PSET - processes the SET pseudo-op.

29. PORG - processes the ORG macro.

30. PREPI - processes the REPI macro.

31. PADDSUB - processes an ADDSUB macro call.

32. PDELAY - processes a DELAY macro call.

33. PSUB - processes a SUB macro call.

34. PFILE - processes a FILE macro call.

35. PNAME - processes a NAME macro call.

36. PENTR. - processes an ENTR macro call.

37. EVAL - evaluates an address expression.

38. REF - obtains the value of a symbolic name.

39. CONVERT - converts display coded octal constants to binary.

40. PACKID - separates a symbolic name from the input string.

41. WRSEQ - writes a sequence of words on the LGO file.

42. WRTEXT - maintains ORGC and POSC, creates and writes TEXT tables for the loader.

43. FOTEXT - forces out the current text table.

44. WRLIST - prints each source line, generated binary instruction and the ORGC.

45. L4.CKL,DEF - defines statement labels.

46. L4.CKRB - does address relocation bookkeeping.

47. ILL,SILL,STOVER - processes error conditions.

48. PEND - terminates the assembly process.

49. PFMT - processes a FMT macro call.

50. PTRACE - processes a TRACE macro call.

51. PENTRY - processes a PENTRY macro call.

52. PGNAME - processes GNAME macro call.

53. PHOL - processes a HOL macro call.

5.1 PIDENT

This routine initializes the assembler. It is entered from the opcode vectors when the IDENT pseudo-op is encountered. The following tasks are performed:

1. Allocate on LGO buffer and setup the FET. The buffer is allocated from MEMSTRT to MEMSTRT+L.LGO. Allocate space for the IO Aplist table.

2. MEMEND, the upper limit of the assembler working storage, is set to the end of IOTAB1.

3. If there were compilation errors, write a prefix table and the text "ERRORS IN FTN COMPILATION" or the binary file.

4. If the object listing option "O" has been specified, set the switches at NOBINPO and WRTSW with return jumps to WRLIST. Set LCNT, and entry point in LIST, to zero to cause a page eject and set LINE to LINE+4 to blanks.

5. Print the IDENT and USEBLK card.

6. If LCC cards are present, place each on the file LGO file with an end of record write. Skip over any LDSET directives.

7. Put a 15 word prefix table with the program name and compilation time, date and options into the LGO buffer. Add a LDSET USE table for the libraries required by FTN object code.

8. Put the ID word and program name word for the PIDL table in the LGO buffer.

9. Scan the COMMON portion of ORGTAB until a zero word is encountered moving the COMMON block names and lengths to the PIDL table and using the subroutine BUILDOT make a 22 word CORGTAB entry for each. The 22 word CORGTAB is built upwards in memory from MEMSTRT and is terminated by a zero word. If the block is an ECS/LCM block the length is rounded up and divided by eight at this point.

10. LORGTAB entries of 22 words each are built for the seven local relocation bases that are always present using the subroutine BUILDOT. The lengths of the bases are stored in the locations START , VARDIM , ...,HOL which are entry points in ORGTAB. The names of the relocation bases are taken from the assembler table LOCNAM.

11. If the current subprogram is a subroutine or function and there are formal parameters, there must be a 22 word LORGTAB entry setup and initialized for each. The names and lengths come from the two word symbol table. The formal parameters are contiguous from ordinal two each having the FP bit set to one. The length of the relocation base is placed by Pass 2 in the RA field and is changed to a program relative address at this time while the RB are set beginning at seven in increments of one and RL is set to 1.

12. The LORGTAB is terminated with a zero word and the program length which was accumulated during LORGTAB construction is printed then stored in the PIDL table. MEMSTRT is adjusted to the next location after the terminating zero word.

13. Replace special symbol cell contents (VALUE.,ST.,etc.) with their program relative addresses.

14. Map the entry point table constructed in pass one into the ENTR table in the LGO buffer.

15. Process the EXT table constructed in pass one and produce the loader table of externals as well as the LINKTAB.

16. The next step is to print the external names if the list code option was selected.

17. If this is a block data subprogram exit to INITL. The actual parameter, variable dimension and generated label definition tables are scanned. The tables lie immediately below the two word symbol table and are terminated by a zero word. Each entry contains in the lower 18 bits definitions of the AP,VD and GL labels relative to the CODE. relocation base. Any or all of these tables may be empty. The base address of CODE. is added to make the definitions program relative. The entries reformatted to look like word B of the two word symbol table.

18. For a main program, the following processing occurs:

    a. Read in and process N.FILES number of cards (these will be FILE or FEQU cards).

    b. Read and list the next card which must be FILES. BSS 0B.

    c. Read and process N.FILE number of FLINK cards.

    d. Read the DATA -nnnB card and place the print limit value on LGO.

    e. Call PTRACE to process the TRACE macro.

f. Read and list the USE CODE. card.

g. Read and list the PENTRY macro call.

h. Exit to INITL.

19. For a subprogram, the following processing occurs:

a. Process the TRACE macro via PTRACE.

b. Read and process the PENTRY macro.

c. Exit to INITL.

5.2 BUILDOT

This subroutine is used during assembler initialization to allocate and initialize a 22 word ORGTAB entry corresponding to each relocation base in the program.

The calling sequence is:

B2 = Origin counter (base address) for this block. This will be zero for common blocks and the sum of the lengths of previous local blocks for program relocation base.

B3 = Relocation base code. This is the LCT ordinal passed out to LGO and used by the loader. It starts at three for common blocks and for local blocks is always one.

B6 = First word address of the 22 word entry.

X1 = The relocation base name in bits 18-59 with blank fill. Bits 0-17 must be zero.

X2 = Block length, a running sum of block lengths is maintained in B2 to provide the current origin counter for local blocks and the program length at the end of ORGTAB initialization.

5.3 INITL (NOBINPO)

INITL with its alternate entry point NOBINPO is the first routine in the main assembly loop. Its function is to read the next source image into ILINE from the COMPS file. While doing this, INITL also manages the COMPS FET

and issues read requests whenever there is room for one PRU in the buffer. The number of words moved to ILINE is stored in location SWC. The alternate entry NOBINPO is used whenever the last line processed did not produce any binary output. It is one word located at INITL-1 and filled with NOPS unless the "O" option is specified. In this case, it is plugged with a call to WRLIST during initialization. INITL exits to START to begin processing the line.

5.4 START

This routine sets the following registers for the character pickup macros GCH and CWD:

B6 = 6
B7 = 54
A5 = ILINE
X5 = (ILINE)
X0 = 54 bit mask; left justified

START exits by jumping into COL1VEC using the first character of the line as an index.

5.5 BLANK, OCSCAN, PFC, PLABEL, PNL, OPCODE VECTORS

5.5.1 COL1VEC: A jump vector used to determine the contents of the label field. It partitions the first 50 display characters into the following sets and branches to the indicated routines for further processing.

(A, B, C,...,Z,),.,=,(,)) branch to PLABEL

(+,-) branch to PFC

(blank) branch to BLANK

(Zero Byte,0,1,2,3,4,5,6,7,8,9,+,1,(,$,=,comma)
branch to ILL

5.5.2 BLANK

FFLAG is set to NFLAG to cause a force upper if the last instruction was an unconditional jump. BLANK exits to OCSCAN.

5.5.3 PFC

FFLAG is set to +1 or -1 for plus or minus in column one respectively. This indicates force upper or force lower to WRTEXT. PFC skips to the opcode field beginning in column three, places the first 2 characters in B2 and B3 and transfers to FLVEC-1+B2 to interpret the opcode.

5.5.4 PLABEL

The label beginning in column 1 is separated from the line using the subroutine PACKID and saved in the location ALABEL for later definition. FFLAG is set to +1 to indicate forcing upper is required and PLABEL exits to OCSCAN. If the label is of the form [IOn make an entry in the IO Aplist table defining the address of the IO aplist number n.

5.5.5 OCSCAN

Blanks between the label field and the opcode are skipped and the first two characters of the mnemonic are placed in B2 and B3. OCSCAN exits by transferring to FLVEC-1+B2.

5.5.6 Opcode Recognition Transfer Vectors

Opcode fields are decoded by indexed jumps using succeeding characters of the mnemonic. Most instructions can be identified by examining the first two characters of the opcode field but for those that require further scanning, a vector for each of the third, fourth and fifth letters has been included along with the routine PNL which picks the next character from the input string and jumps back to the vectors. The vector exits that result from an executable instruction mnemonic will set X4 to an instruction prototype that is completed as the address field is processed. Other jumps that result from pseudo-ops and macro calls simply transfer to specific routines, although in some cases the spare 30 bits in the last vector position is used.

5.6 L3.RJ

This routine processes the RJ instruction. The address for the symbol is fetched from the two word symbol table using the REF subroutine and NFLAG to set to one indicating the next instruction is to be forced upper.

5.7 L3.JP

The JP may appear with or without a symbol in the address field but is always indexed by B1. If there is a symbol, the subroutine REF is used to obtain its address. NFLAG is set to one and the routine exits to WRTEXT.

5.8 L3.XJP

This routine processes the X register conditional jumps ZR, NZ, PL, and NG. The register number is selected from the input string and REF is called to obtain the jump address before exiting to WRTEXT.

5.9 L3.EQ

The EQ jump may be either conditional or unconditional. If unconditional, no B register is specified or Bi will be the same as Bj. In this case, REF is called to obtain the jump address, NFLAG is set to one and exit is made to WRTEXT. When the two B registers present are different, i and j are selected from the input string, added to the prototype, and REF is called for the symbol definition before the exiting to WRTEXT.

5.10 L3.BOOL

For boolean instructions, the 15-bit opcode is determined by examination of the address field. This is done by observing the first and second operators to determine the g and h fields and setting i, j and k from the input string. The exit is made to L4.15.

5.11 L3.MX, L3.SH

This routine processes the mask instruction and the four shift instructions. Here, it is required to determine whether the shifts are nominal or constant, reset the opcode in the former case and set the i, j and k fields in both cases. The exit is to L4.15.

5.12 L3.PUN

The assembler must select and set the i, j and k fields for the pack, normalize and unpack instruction. The exit is to L4.15.

5.13 L3.ARIT

The instruction for double and single precision floating point operations and the 60 bit integer operations require only the selection of a mask from ARITAB using the operator character code as an index and or it with the prototype in X4. This routine exits to L3.PUN to set the i, j and k fields.

5.14 L3.SET

The increment unit instructions have the largest variety of address fields of any class of statements that are encountered by the assembler. This routine must determine the second digit of the opcode and the remainder of the 15 or 30 bit instruction. The analysis begins by transferring into L3.JVEC using the first character of the address field as an index. This will cause a transfer to L3.S1 if the first character is an A, B, or X to L3.S11M if the first character is a minus sign, to L3.S7 if the first character indicates a symbol or constant follows and to ILL otherwise. At L3.S1, a character-by-character scan of the input string is used to determine if the first item in the address field is a register name or a variable. For variables, a transfer is made to L3.S7 to evaluate the address expression, while for registers, the h field of the opcode is adjusted, j is selected and included in the instruction, and the scan continues after setting X7 to remember the sign unless the next character is a zero byte. If the next item in the string is a register name, the opcode is further adjusted, k is selected and set and control is transferred by L4.15. Otherwise, a branch is made to L3.S7 to continue the address expression analysis. At L3.S11M, X7 is set to indicate the preceeding minus sign and control goes to L3.S7 where the subroutine EVAL is called to evaluate the symbolic address field. The routine exits to WRTEXT with a 30 bit instruction.

5.15 L3.VFD

The VFD pseudo-op routine translates data subfields one by one, packing the information into the item being constructed. Numeric and symbolic fields are converted using the subroutine EVAL while character string data is packed and formatted by the VFD routine itself. The data fields that can be successfully assembled are limited as follows:

1. The total of the bits that are specified in any one appearance of a VFD must be 15, 30 or 60.

2. Relocatable fields must appear as the lower 18 bits of the specified field.

3. Character data must be C, H, L or R specification.

5.16 PDIS

The DIS routine moves the number of words specified from the line buffer to the current text table by calling WRTEXT once for each word.

5.17 PDATA

The only type of DATA fields the assembler will encounter are single octal constants. They are converted to binary by the subroutine CONVERT and added to the current text table by WRTEXT.

5.18 PBSS

The BSS routine first forces upper by setting FFLAG to 1 and calling WRTEXT. The argument is converted to binary by the subroutine CONVERT and added to the ORG counter, then FOTEXT is called to write out the current text table and start a new one with the updated origin counter. This routine exits to NOBINPO to print the line.

5.19 PUSE, USENXT, USESTAR

The USE processor separates the relocation base name from the input string using PACKID and after determining whether it is common or local and selecting the correct portion of the 22 word ORGTAB, it calls the subroutine USENXT to change TEXT.ADD to this relocation base entry. USENXT makes a linear search of either the CORGTAB or the LORGTAB until it finds the relocation base name. It then saves TEXT.ADD in the location USEBB and resets it to the address of the new ORGTAB entry and exchanges the contents of NFLAG with the contents of the second word in the table entry. USENXT is also called by some of the pseudo macro routines. USESTAR is called to change back to the previous block by resetting TEST.ADD from USEBB and exchanging NFLAGS.

5.20 L3.ORG

The ORG instruction requires, after evaluation of the address field by EVAL, the relocation base to be changed and the origin counter in the new base to be reset. The subroutine FOTEXT is used to force out the old text table and start a new one with the correct origin counter.

5.21 PADDSUB

5.21.1 PADDSUB processes the address substitution macro, ADDSUB. This macro will be called by the program being assembled only once, in the event that it is a subprogram having parameters. After calling USENXT to change to the VARDIM. relocation block, its prototype at ADDSC, except for the last word, is written on the LGO file by WRSEQ. The last word will be 30 bits if MACHINE ≠ 6600B, and 60 bits if MACHINE=6600B; it is written on the LGO file by WRTEXT. The correct relocation byte is added, and USESTAR is called to change back to the previous relocation block. Exit is to INITL.

The prototype is set to execute at a location specified by the LOC pseudo-op which just precedes it; its address field should be set to the address of the first word of the VARDIM. relocation block, which will be the same for all subprograms.

5.21.2 The macro prototype is:

```
ADDSUB MACRO FP
       USE VARDIM.
       SB4 1
       SA3 FP-1
       MX0 42
       SB6 60
       NO
B)     SA3 B4+A3
       SB7 X1
       SA1 A1+B4
       SA2 X3
A)     UX4 X3,B2
       SB3 A2
       LX4 42
       SB5 B6-B2
       SA3 A3+B4
       LX2 B2,X2
       SX5 X4+B7
       BX4 X0*X2
       SA2 X3
```

```
BX6 -X0*X5
IX4 X6+X4
LX6 B5,X4
SA6 B3
NZ X3,A)
NO
NZ X1,B)
JP *+1
USE *
ENDM
```

5.22 PSUB

5.22.1 PSUB processes the SUB macro. This macro is called by the program being assembled after it encounters a reference to a formal parameter. The form of the call is:

```
SUB   FP,K
```

where the comma and K may be missing and the form of the resulting entry is:

```
VFD  3/2,9/POSC,30/K,18/ORGC
```

Entry is made into the FP relocation block.

5.22.2 PACKID is called to strip the block name FP, which is left in position for the subsequent call to USENXT. The position counter (POSC) and the origin counter (ORGC) are extracted from the current block and saved in B registers. USENXT is called to change to the FP block. Returning, FFLAG is set to 1 to force upper and the position counter is converted from FTNXAS form to COMPASS form. Now the sublist entry will be formed, starting with the origin counter. B1 contains the delimiter left by PACKID, and if non-zero, the constant is converted by CONVERT, shifted, and ORed into the entry. Lastly, the position counter is packed into the entry and WRTEXT is called to write the entry on the LGO file. The correct relocation byte is added, and USESTAR is called to change back to the previous block. Exit is to INITL.

5.22.3 The macro prototype is:

```
SUB MACRO FP,CON
 .POS SET 59-$
 .ORG SET *-$/59
```

```
USE FP
VFD 3/2,9/.POS,30/CON,18/.ORG
USE *
ENDM
```

5.23 PDELAY

5.23.1 PDELAY processes the DELAY macro. This macro is called by the program being assembled if it is necessary to call SUB twice in the same word for the same formal parameter. The form of the call is:

```
DELAY  FP
```

and the form of the resulting entry is:

```
VFD  3/2,9/30,48/ST.
```

where ST. is the address of the start of the sublist table. Entry is made into the FP block.

5.23.2 PACKID is called to strip the block name FP, and USENXT is called to change to the FP block. SYMBOL is then called to obtain the address of ST., which is shifted, marked off, and packed with the 30 field into X4. WRTEXT is called to write the entry on the LGO file. The correct relocation byte is added, and USESTAR is called to change back to the previous block. Exit is to INITL.

5.23.3 The macro prototype is:

```
DELAY MACRO FP
     USE FP
     VFD 3/2,9/30.30/0,18/ST.
     USE *
     ENDM
```

5.24 PFILE

5.24.1 PFILE processes the FILE macro. This macro is called when a main program being assembled wishes to set up a FIT/FET and buffer for a file. It uses two prototypes, which are formatted as follows:

1) FIT prototype:

```
VFD  60/0
```

```
VFD  42/0,18/FET pointer
VFD  30/0,2/2,28/0
VFD  60/0
VFD  60/0
VFD  42/0,18/buffer address
VFD  60/0
```

2) REPI prototype:

```
VFD  6/43B,18/2,36/1
VFD  42/0,18/FWA of zero's
VFD  18/23B,42/0
```

5.24.2 CONVERT is called to convert the buffer length to binary. The origin counter is obtained, placed into the FIRST, IN, and OUT fields of the FET prototype at FILEC, and the buffer length + origin counter is placed into the LIMIT field. Since the zero word which is the sixth word of the FET must be repeated 13B times, its address is placed into the second word of the REPI prototype. The new origin counter is saved in PFILEC, and WRSEQ is called to write the prototype on the LGO file. FOTEXT is then called to force out the current text table so that the REPI table can duplicate the zero word. Then the origin counter is reset to the value saved in PFILEC, and WRWDS2 is called to write the REPI table on the LGO file. Exit is to INITL.

5.24.3 The macro prototype is:

```
FILE MACRO LN,NAME
     ENTRY NAME→ .
NAME→. BSSZ 1B
IN$ SET NAME→.
LG$ SET LN+1
     VFD 16/1,26/12/18/IN$+17
     VFD 60/IN$+17/60/IN$+◆17
     VFD 60/IN$+17+GL$
     BSSZ 14B
     BSS LG$
     ENDM
```

5.25 PNAME

5.25.1 PNAME processes the NAME macro. This macro is called by the program being assembled when a NAMELIST string is to be defined. Processing takes place in two phases: 1) stripping, partially processing, and saving the actual

parameters, and 2) forming the required binary output and writing it onto the LGO file. The format of the call is:

```
NAME N,T,BASE,BIAS,FP,NDIM,D1,D2,D3
```

where N and T are always present, and the rest of the string may be entirely missing. In addition, BASE and BIAS may be concurrently missing, and the following combinations of the D fields may be missing: D1, D2, D3; D2, D3; D3. If NDIM is missing then D1, D2, D3 will not be present.

5.25.2 Phase 1 begins by setting the locations from NNAME to Z3N, which will contain information derived from Phase 1 processing of the actual parameters, to zero. REP is called to obtain the NAMELIST name (N), which is saved in NNAME, and its address, which is stored in VNAME. CONVERT is called, which converts the NAMELIST type (T), and this is stored in TNAME. At this point, a test is made to see if the next character is a zero byte. If it is, then there are no more parameters and transfer is made to phase 2 at PNAME4. Otherwise, a check is made to see if the BASE field is present, and, if it is, then the base is stripped by REF and stored in BASN and the bias is stripped by CONVERT and stored in BIASN. The CONVERT strips the FP field, if present, and it is stored in FPN. If the NDIM field is not present we go to phase 2 at PNAMEX. At this point, CONVERT is called to strip D1, D2, and D3 until one is found missing or all three are stripped, and they are saved in Z1N, Z2N, and Z3N, respectively. If all three fields are missing, ZZN is left at its initial value of zero, otherwise, it is set to one. Phase 2 at PNAME4 begins by getting the NAMELIST name from NNAME, changing the trailing blanks to zero characters, and calling WRTEXT to write onto the LGO file a word in the following format:

```
VFD  42/NAME,12/NDIM,6/T-T*8/8
```

If the BASE and BIAS fields were present, they are added together to form the address for the next word; otherwise, the NAMELIST address from VNAME is used. This address and the NAMELIST type from TNAME are written on the LGO file in the following format

```
VFD  30/D1,1/T/8,1/1,28/FP     if FP is present
```

or

```
VFD  30/D1,1/T/8,29/BASE+BIAS if FP is null and
                              base is nonnull
```

or

```
VFD  30/D1,1/T/8,29/N         if base is null.
```

If NDIM is greater than 1 a third word is generated. The form is

```
VFD  30/D3,30/D2
```

Exit is to INITL.

5.25.3 The macro prototype is:

```
NAME    MACRO   N,T,BASE,BIAS,FP,NDIM,D1,D2,D3
        LOCAL   Z
Z       MICRO   1,,$N$
        VFD     42/0L=Z=,12/NDIM,6/T-T/8*8
        IFC     NE,/FP//,2
        VFD     30/D1,1/T/8,1/1,28/FP
        ELSE    5
        VFD     30/D1
        IFC     NE,/BASE//,2
        VFD     1/T/8,29/BASE+BIAS
        ELSE    1
        VFD     1/T/8,29/N
        IFGT    NDIM,1,1
        VFD     30/D3,30/D2
        ENDM
```

5.26 PENTR

5.26.1 PENTR processes the ENTR. macro. This macro will be called by the subprogram being assembled when an entry point other than the main entry point is to be provided. Although there are two types of ENTR. macro expansions possible, depending upon whether a subprogram has formal parameters or not, the expansion throughout a particular subprogram will be consistent, and a jump to the particular processor required will be stored over a word of NOP's located at the beginning of the initialization through PENTR1 and execute the initialization phase; subsequent calls will be directed to the appropriate processor at PENTR1. The format of the call is:

ENTR. NAME

ENTR. macro prototype with arguments:

```
VFD  60/0
VFD  30/SA2(*+2),15/BX6 X2,15/NO
VFD  30/SA6(FTNNOP.),30/EQ(ENTRY.+1)
VFD  30/EQ(*+1),15/NO,15/NO
VFD  30/SA1(NOPS.),30/SA2(NAME)
VFD  15/BX6 X1,15/LX7 X2,30/SA6(FTNNOP.)
VFD  30/SA7(ENTR.),15/NO,15/NO
```

ENTR. macro prototype without arguments:

```
VFD  60/0
VFD  30/SA1(NAME),15/BX6 X1,15/NO
VFD  30/SA6(ENTR.),15/NO,15/NO
```

Parentheses indicate values to be substituted.

5.26.2 REF is called to get the address of NAME, and it is stored in ENAME. If this is not the first time through, PENTR1 transfers to PENTR6 if the subprogram has arguments, or to PENTR8 if it does not. Otherwise, initialization begins by testing word A of the second symbol table entry to see if the FP bit is set; if it is not, then there are no formal parameters and after setting a jump to PENTR8 into PENTR1, control is transferred to PENTR5 to complete the initialization for this case. Otherwise, there are parameters, and the jump at PENTR1 is set to a jump to PENTR6. SYMBOL is called to get the address of NOPS., which is saved in ANOPS; FTNNOP., which is saved in AFTNN; and ENTRY., which is saved in AENTR. Control is then transferred to PENTR1 to jump to the proper processor. At PENTR5, initialization for the no parameters case continues as above with the calling of SYMBOL to save the address of ENTRY. in AENTR. PENTR6 is the start of the processing for a call to ENTR. in a subroutine with arguments. The origin counter is obtained to form a base for the self-relative substitutions in the macro prototype, and a jump around the entire macro expansion is formed and left in X4. Then the following substitutions are made:

1. ORGC+4 upper address of word 2

2. ORGC+5 upper address of word 4

3. FTNNOP. lower address of word 6

4. FTNNOP. lower address of word 3

5. ENTR.+1 lower address of word 3

6. NOPS. upper address of word 5

7. NAME lower address of word 5

8. ENTR. upper address of word 7

Now the jump in X4 is sent to the LGO file by WRTEXT, and WRSEQ is called to write out the entire prototype onto the LGO file. Exit is to INITL. PENTR8 is the start of the processing for a call to ENTR. in a subroutine with no arguments. The origin counter is obtained and a jump around the entire macro expansion is formed and left in X4. Then the following substitutions are made:

1. NAME upper address of word 2

2. ENTR. upper address of word 3

Now the jump in X4 is sent to the LGO file by WRTEXT, and WRSEQ is called to write out the entire macro expansion on the LGO file. Exit is to INITL.

5.26.3 The macro prototypes are:

```
ENTR. MACRO NAME
      LOCAL X,Z,T
      EQ T
NAME  BSS 1
      ENTRY NAME
      SA2 X
      BX6 X2
      SA6 FTNNOP.
      EQ ENTRY.+1
X     EQ Z
Z     SA1 NOPS.
      SA2 NAME
      BX6 X1
      LX7 B0,X2
      SA6 FTNNOP.
      SA7 ENTRY.
T     BSS 0
      ENDM
```

when formal parameters appear or:

```
ENTR. MACRO NAME
      LOCAL T
      EQ T
NAME  BSS 1
      ENTRY NAME
      SA1 NAME
      BX6 X1
      SA6 ENTRY.
T     BSS 0
      ENDM
```

when formal parameters do not appear.

5.27 EVAL

This subroutine is used to evaluate address expressions consisting of octal constants, symbols and the operators + and -. It must be entered by a return jump to EVAL with:

B3 = X1 = the first character of the expression,

X6 = plus or minus zero indicating a preceding minus (or implied plus),

X7 = any previous address sum,

B2 = bit count of the character in X1.

EVAL returns with the expression value in X7, and it has been added to X4.

5.28 REF

REF is used to obtain the equivalent address of any symbol or label encountered during the assembly process. This routine is entered with a return jump to REF with the symbol name in X1, the first character in B3 and the bit count of X1 in B2. If the name begins with either of the special characters [ ], then its equivalent address is obtained from the generated label, actual parameter, IO aplist or variable dimension label definition table. Otherwise, the subroutine PACKID is used to separate the label from the input string and format it left justified with blank fill in the lower 48 bits of X1 in preparation

for calling SYMBOL or LABEL. If the symbol is an external name, X3 is set to zero (relative address) and the exit is made. If not external, word B of the symbol table entry is stored in RBTEMP unless it is already non-zero, in which case, it is cleared to zero, the RA field is separated to Z3, and the exit is made.

5.29 CONVERT

A display coded octal number which may be preceded by a minus sign is converted to binary. The first character of the constant must be placed in the lower 6 bits of X1 before the return jump entry. On return, the converted value is in X1.

5.30 PACKID

This subroutine is used to separate identifier names from the input string. It will pack up to eight characters until a zero byte, +, -, /, blank or comma is encountered. The character string is left justified and blank filled in the lower 48 bits of X1. The character that served as the delimiter is preserved in B1 upon exit.

5.31 WRSEQ

WRSEQ writes up to 15 words of data with relocation information into the current text table. The calling sequence is:

```
A4 = first word address of text block,
X4 = first word of text block,
X7 = left justified 4 bit relocation bytes,
RJ   WRSEQ.
```

5.32 L4.15, WRTEXT

L3.15 is an alternate entry to WRTEXT used when a 15 bit quantity is to be written. WRTEXT builds and writes text tables for the loader from the instructions and data the assembler produces. The calling sequence is:

```
X4 = data to be written,
B1 = bit count of data,
B6 = return address,
EQ   WRTEXT.
```

WRTEXT forces upper when it is required by data size or the FFLAG is greater than zero. It also maintains the position counter and the origin counter for each relocation base and adjusts the relocation byte word in the text table. If the switch at WRTSW has been set, WRLIST will be called to print the line.

5.33 FOTEXT

This subroutine is entered by a return jump. It terminates the current text table by forcing upper, installing the word count in word 1 and left justifying the data bytes in word 2. The text table is then written on the binary output file. Before returning, a new text table is initialized with the origin counter in word 1.

5.34 WRLIST

This subroutine is responsible for producing the assembly listing. It is given the binary to print, if any, and the current source statement in ILINE. If the position counter is 60, the origin counter and the current relocation base name are also printed. If RBTEMP is non-zero, the relocation base name of the address field will also be printed.

5.35 L4. CKL, DEF

After a statement has been decoded and the binary added to the text table, these two subroutines are used to define the label that appeared on the statement if any. The pseudo-ops DATA and VFD result in transfer of control to L4.CKL, but BSS calls DEF itself. At L4.CKL, DEF is called if the contents of ALABEL are non-zero and control is passed to L4.CKRB. DEF looks up the symbol whose name is in X1 on entry in the two word symbol table using LSTPROC. For the label, it then sets the RA field to the origin counter, the RL field to 1 if the current block is local, for common the RL field is set to 2 and the RB field to the ORGTAB ordinal for the block the symbol is defined in. Before the return location, ALABEL is cleared to zero. DEF is disabled by the USE processor which stores an "EQ DEF" in DEF.1 after the second "USE CODE".

5.36 L4.CKRB

Any 30 or 60 bit quantity that has been generated may have a relocatable address in the lower 18 bits. In this case, RBTEMP will reflect this by having been set to word B of the two word symbol table entry, otherwise, it will be zero. If RBTEMP is non-zero, the RL field is examined to determine the type of relocation necessary. Program relocation requires the relocation byte word in the current text table have a 2 added to it and RBTEMP be set to zero. A common or external reference requires a data byte for the loader be created and linked into the reference chain corresponding to the variable. For common, the starting address of this chain is located in the lower 18 bits of the first CORGTAB word for the entry that corresponds to the common block in which the variable is defined. This word is located at (CORGTAB) + RB*22. The external reference chain begins in the lower 18 bits of the LINKTAB entry that contains the variable name. The RA field of RBTEMP contains this address. The new one word link is taken at the address contained in FREEMEM, which is incremented by one and compared to the contents of MEMEND and replaced. If FREEMEM is greater than MEMEND, working storage has been exhausted, and the error routine STOVER is called. Otherwise, RBTEMP is cleared and L4.CKL exits to INITL to prepare for the next line.

5.37 ILL, SILL, STOVER

5.37.1 ILL picks up the message "ILL", transfers it to location ILL.1 which stores the message in LINE + 3, sets ILLFLAG to non-zero, calls WRLIST to print the message, and the source image then exits to INITL to prepare for the next statement.

5.37.2 SILL, where control is transferred in the event LSTPRO cannot find a symbol name in the two word symbol table, picks up the message "SYMBOL ERR" and transfers to ILL.1.

5.37.3 STOVER is called when all available working storage has been used. The first time this occurs the message "STORAGE OVERFLOW, NO OBJECT PROGRAM WILL BE PRODUCED" is printed. The size of the overflow is added to the contents of location STOVSIZE and FREEMEM is reset to the contents of MEMSTART. The exit is to INITL.

5.38 PEND

Control is transferred to PEND when the END pseudo-op is encountered to clean up the assembly process and terminate. The following tasks are performed to accomplish this:

1. If the contents of STOVSIZE is not equal to zero, the size of the overflow is calculated and printed with the message "INCREASE FIELD LENGTH BY XXXXXX" and exit is made as if ILLFLAG was non-zero.

2. If the contents of ILLFLAG are non-zero, the following steps are taken before exiting:

   a. An End-of-record is written on LGO,

   b. LGO is backspaced one logical record,

   c. A prefix table with the program name is written in LGO,

   d. Control is transferred to EX.90 to do another end-of-record write on LGO, print the END pseudo-op and return.

3. All partially filled text tables are forced out on LGO from CORGTAB and LORGTAB by calling FOTEXT once for each relocation base.

4. The accumulated common and external reference information is formed into FILL and LINK tables respectively and written on LGO. The contents of the fill chains for LCM common blocks are formed into XFILL tables and placed on the binary file. This is accomplished by following the chain that begins at the CORGTAB and the LINKTAB entries for common block and external symbols and packing the data bytes which are the upper 30 bits in each link into the correct table formats. The COMPS buffer is used as scratch memory for this purpose.

5. An XFER table containing the date and the transfer symbol, if present, is written on LGO.

6. The LGO buffer is cleared out and the relocatable deck terminated by doing an end-of-record write on LGO.

7. The last statement of the program (the END Pseudo-op) is printed by calling WRLIST.

9. The assembler returns through its entry point to CLOSE2.

5.39 PTRACE

5.39.1 Process the TRACE macro call and generate the traceback word on the LGO file.

5.39.2 The macro prototype is:

```
TRACE     MACRO     NAME,ADDRESS,NARGS
X         MICRO     1,,$NAME$
TRACE.    VFD       42/7L≠X≠
          VFD       18/ADDRESS
ARGS.     EQU       NARGS 0
          IFC       NE,/NARGS//,1
TEMPAO.   BSS       1
          ENDM
```

5.40 PAPL

5.40.1 Process the APL macro to generate a pointer word in SCM to the LCM block address.

5.40.2 The macro prototype is:

```
APL       MACRO     SYM,BIAS
          VFD       1/1,59/SYM+BIAS
          ENDM
```

The top bit on signifies an LCM address in the lower part.

5.41 ALR

This routine appends the fill table information for an LCM address to the fill chain. On entry, X2 holds the RB ordinal of the LCM block - 1. Bit 59 is set in word 3 of the common block ORGTAB entry to denote an LCM block with a fill chain. The 30 bit fill entry is flagged as LCM by setting the tsp bit.

5.42 PIOM

5.42.1 Process the IOM macro to generate an IO parameter list entry.

5.42.2 The macro prototype is:

```
IOM     MACRO   BASE1,BIAS,TYPE,COUNT,B59,B57,BASE2
        VFD     1/B59
        IFC     EQ,/BASE1//,2
        VFD     1/1
        ELSE    1
        VFD     1/0
        VFD     1/B57
        VFD     9/TYPE
        IFC     EQ,/B57//,7
        IFC     EQ,/BASE2//,3
        VFD     24/COUNT
        VFD     24/BASE1+BIAS
        SKIP    5
        VFD     24/COUNT
        VFD     18/BASE2,6/BIAS
        SKIP    2
        VFD     18/BASE2+COUNT,610
        VFD     24/BASE1+BIAS
        ENDM
```

5.43 EIO

5.43.1 Process the end of IO list macro.

5.43.2 The macro prototype is:

```
EIO     MACRO   P
        VFD     60/P
        ENDM
```

6.0 Data Formats

Working storage during the assembly process.

6.1 22 Word ORGTAB (LORGTAB and CORGTAB)

These two tables have the same format. They contain a 22 word entry for each relocation base that will be encountered by the assembler. COMMON relocation bases are entered in the CORGTAB and local relocation bases are entered in LORGTAB. Each 22 word entry is formatted as follows:

WORD1: Bits 18-59/Relocation base name, left adjusted with blank fill.

Bits 0-17/Pointer to Fill chain for COMMON blocks; unused for locals. Initialized to zero in both cases.

WORD2: Bits 0-59/NFLAG (force upper, next instruction), initialized to zero.

WORD3: Bit 59 = 1 if LCM common block, Bits 58-24/Unused.

Bits 18-23/Relocation base code. This is 1 for all local blocks and may be 3 to 77B for common blocks. It serves as the LCT ordinal for the loader.

Bits 0-17/Origin counter, initialized to the first word address of the block.

WORD4: Bits 18-59/Unused,

Bits 0-17/Position counter, initialized to 60.

WORD5: Bits 18-59/Unused,

Bits 0-17/Test table ordinal indicates the current word being filled, initialized to 2.

WORD6: Bits 54-59/Text table code number, 40B,

Bits 36-47/Word count, initialized to zero,

Bits 18-23/Relocation code,

Bits 0-17/Load address, initialized to ORGC.

WORD7: Bits 0-59/4 bit relocation fields, initialized to zero.

WORD8: Bits 0-59/Initialized to zero.

WORD9-22: Bits 0-59 Uninitialized.

6.2 LINKTAB

The LINKTAB consists of one-word entry for each external symbol used in the program being assembled. The last entry is a zero word. Each word is formatted:

```
VFD  42/NAME,18/PTR
```

where NAME is the external symbol name, left adjusted with zero fill and PTR is the start of the reference chain that describes the usage of the symbol. PTR is initialized to zero.

6.3 LINK and FILL chains (external and common reference information).

One chain exists for each common relocation base and external symbol. They are linear linked lists of one word elements containing in upper 30 bits a data byte for the loader and in the lower 18 bits a pointer to the next link. Each list is terminated by a zero pointer. The elements of these lists are taken from working storage beginning above the LINKTAB. FREEMEM always points to the next available word in this area. Following is the format of each link:

```
VFD  1/1,2/POS,9/RL,18/LOC,12/0,18/PTR
```

where LOC specified the relative address of the reference, RL specifies the relocation of this address, POS the position in the word and PTR points to the next link in the chain.

6.4 Actual Parameter, Variable Dimension, and Generated Label Definition Tables.

Upon entry, these tables contain the lower 18 bits the address relative to the origin of the CODE. relocation base AP, VD or GL labels. They are reformatted during initialization and take the following form during assembly.

```
VFD  22/0,2/1,18/RA,6/3,12/0
```

where RA is the program relative address of the label.

6.5 Two Word Symbol Table

During initialization of the assembler and during assembly before the second "USE CODE.", word B of the two

word symbol table entry for each symbol is reformatted as follows:

```
VFD  22/0,2/RL,18/RA,6/RB,12/0
```

where RA is a program relative or common block relative address or a pointer to the LINKTAB entry that corresponds to the external symbol, RL is the type of relocation (01=Program, 02=Common, 03=External) and RB is an ordinal to either the CORGTAB or LORGTAB entry which corresponds to the relocation base in which the symbol was defined.

7.0 Modification Facilities

7.1 Assembly Options

7.1.1 Options File

1. MACHINE. This option, which is set to either 6400B or 6600B, causes the ADDSUB Macro to be expanded with a JP *+1 in the last two parcels if set to 6600B.

7.1.2 DEBUG ETC.

This option, and several assembly flags which are normally equated to it, control debugging aids that have been left in the assembler. The normal value of DEBUG is zero, however, setting it to 1 and reassembling will cause (1) SNAP calls to be inserted at strategic points, (2) ORG and MOVE macros not to be expanded, and (3) the last few words of L3.JVEC to be assembled in. The SNAP routines must be available when DEBUG is non-zero.

7.2 Opcode Vectors

Opcode fields are decoded by the assembler by placing the first two characters of the field in index registers B2 and B3 and jumping to B2+FLVEC-1. If the opcode can be uniquely recognized from the first two letters, an exit is made to the correct routine for processing the address field. Otherwise, the address of a further vector is placed in B2 and the subroutine PNL is called to separate the next character from the input string, add it to B2 and jump to the contents of B2. New entries and new vectors can be placed anywhere in the existing general

scheme, but care must be taken to insure that a valid vector entry is not within the range of an ORGSTART, ORGEND pair. Also, each unused word in new vectors should contain an "EQ ILL" and each block of two or more unused words should be surrounded by ORGSTART, ORGEND macro calls.

7.3 MIC, ORG, MOVE Macros

MIC creates a micro with the name equal to the second parameter. It will be a character string representing the value of the first parameter. Thus, after

```
N     SET     423

      MIC     N,Z
```

the catenation A B C ≠ Z ≠ would equal ABC423

The purpose of the move macros are to allow the definition of unused areas in the assembler, of areas of code which can be moved, and the moving of the code into the unused areas at assembly time.

The unused areas are bracketed by ORGSTART and ORGEND macro calls, which build tables of lengths and first word addresses of unused areas. After the last ORGEND macro call, the areas of code which can be moved are bracketed by MOVSTART and MOVEND macro calls. Each pair of calls result in the relocation of the associated code into the smallest area in which it will fit. The tables are adjusted to reflect the usage. If all unused areas are too small, no action is taken. Changes in the size of a code block bracketed by MOVSTART, MOVEND must be reflected in the MOVSTART call parameter or an ERR pseudo-op will be produced.

REFERENCE MAP

1.0 General Information

Various modifications were made to passes 1 and 2 of the compiler for version 3 so it could produce a source keyed reference map, with relative addresses for the local variables, statement labels and common variables. In most cases, the routines were entirely reorganized and recoded (FTN, LSTPRO, PS1CTL, PH2CTL, DECPRO, GOTO, DOPROC, etc.), while in others minor surgery was performed (DATA, SCANNER, ERPRO, POST, PROSEQ, FAX, etc.)

The R option is keyed off the presence or absence of the P option on the control card, flags are set by FTN which are interrogated by pass 1 of the compiler. For the short map, the reference map processor derives all its information from the symbol table and the common block name and length table, ORGTAB. For the R = 2 long map option, additional information is collected during passes 1 and 2. This includes references to symbols appearing in source statements, loop information and the table of usage defined variables as built by ENDPFO. During the code generation phase of pass 2, PRE builds a loop information table. For the R = 3 option, the common block member and equivalence information tables are saved by DPCLOSE.

All this information is then printed out by REFMAP at the end of the code generation phase of pass 2.

The routines involved in reference map production and their functions are:

FTN--Set flags if R option specified on control card and allocate a buffer (REFMAP) for collecting the reference if R≥2.

Flags set are R=FLAG (0,1,2 or 3), and RSELECT is set to a non-zero value if R≥2.

LSTPRO--Rewind refmap file at the end of pass 2 if R≥2.

All pass 1 statement processors -- Call ADDREF to collect references for symbols as they are encountered in a statement (R≥2 only).

PS1CTL--Contains the routine ADDREF which formats the references into "lines" and writes them out to the REFMAP file. Also, terminates the REFMAP file at the end of pass 1 (LDPS2) and writes the saved common and euqivalence information to it.

DPCLOSE--saves the common and equivalence class tables if R = 3.

PRE--Allocates and builds the loop information table from information saved in do begin macros by DOPROC in pass 1.

POST--computes the loop length and places it in the loop table when a jump backwards is generated OP Bi,Bj,)XX where OP = GE, LT, LE, GT, etc., and)XX is the loop generated label for the top of the loop.

REFMAP--Updates word B of the symbol table so that the addresses of all local symbols are changed from block relative to program relative.

Sorts the symbol table alphabetically and then by categories. If R≥2, the REFMAP file is rewound, read in and the references sorted by symbol table ordinal. At this point, the reference map for each category is produced. If R = 3, then the saved common and equivalence information is read in and used by the routines that print out the common blocks and equivalence classes.

REFMAP--exits through its entry point.

2.0 Flow of Control

Initialization--check flags, compute program length, adjust addresses of local symbols in symtab, print out any missing labels, sort symbol table alphabetically and by categories, print title line, read in REPMAP file if present and sort, search for stray names.

Output phase--print out reference map for: ENTRY POINTS, VARIABLES, FILE NAMES, EXTERNAL NAMES, INLINE FUNCTIONS, NAMELIST GROUP NAMES, STATEMENT LABELS.

If (R≥2 and no compilation errors) print out loop map. Print out common blocks and members, equivalence classes and members.

Print program statistics.

Pre-exit code--restore changed names in symbol table and name of blank common if changed. Note that refmap sets the function bit for stray names and does not turn it off. It may in the future be necessary to do so.

Entry Points

REFMAP	main entry point
LWA.R	LWA+1 of REFMAP
DSORT	a simple switch sort routine

3.0 Diagnostics

The following messages are informative and pertain to a lack of storage:

CANT SORT THE SYMBOL TABLE

REFERENCES AFTER LINE NNNN LOST

INCREASE FL BY NNNNB

Refmap also checks the program length, and if it is greater than 377 777B, it issues a FE diagnostic to the output file.

4.0 Environment

Refmap expects that the symbol table is intact and properly formatted, i.e., all symbols have been assigned block relative addresses.

The local and common block lengths have been saved in ORGTAB. Pass 1 has collected references for symbols,

saved the Common/Equivalence tables, the UDV table, etc. If loops were present and R≥2, Pre and Post have constructed the loop table. LWAWORK is the LWA of working storage.

5.0 Structure

Print subroutines

Z8	converts binary number to octal with trailing B
OCTC	converts binary number to octal with trailing B
PSTITLE	print subtitle line
LISTV	list all references for a symbol
LISTR	lists all references in a category (Refs, defs) for a name
FMT	formats data into a coded line
DLL	dump the last line
PBNB	prints bias, base (name) for COMMON/EQV printouts

Sorts

SORTSYM	sorts the symbol table alphabetically
SORTC	sorts list of names into categories
SORTR	sort REFMAP file by symbol table ordinals
SORTRL	sorts a single reference list into reference, definition and FREF lists
DSORT	simple interchange sort
LINKUP	subroutine of SORTSYM
CNAME	subroutine of SORTSYM

Initialize

CPL	computes program length, relocates symbol table

ILW Issues SCM address pointer words for all LEVEL 2 or 3 symbols.

6.0 Modification Facilities

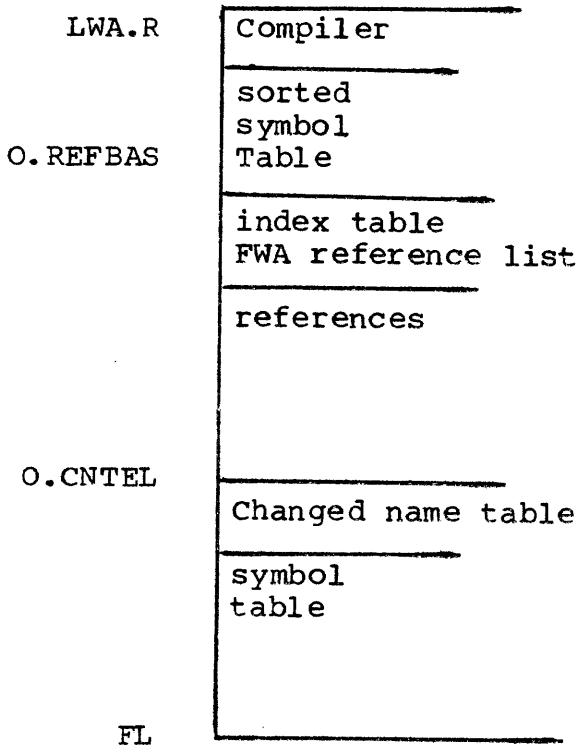
REFMAP calls the COMDFCK OPTIONS and the length of a coded line is determined by the local symbol WPL.

7.0 Methods

The print routines are word oriented for speed. The symbol table sort is a radix sort on the characters.

On the print loops for the various categories, X1 and X2 hold words A and B of the symbol table entry to be printed and the use of other registers is controlled by the macros EFIELD, GSYM, etc.

Before attempting to modify one of the print loops, one should study the macro GSYM which sets up the registers.



POST

1.0 General

POST resides in Pass 2 and performs the following operations.

Converts R-list entries into COMPASS line images.

Inserts SUB and DELAY macro references into the COMPASS string when a location field contains a formal parameter.

Forms traceback information for 60 bit return jumps.

Maintains the number of SUB and DELAY references issued for each formal parameter.

Maintains the length of generated code for both the CODE. and VARDIM. relocation blocks.

Defines statement labels (prefixed with a decimal point) and generated labels (prefixed with equivalence symbol).

2.0 Entry Points.

2.1 OPNPOST

Initializes the POST routine and is entered via a return jump each time a new sequence is initiated. It sets the COMPASS string buffer limits and initializes to zero the number of SUB's and DELAY's issued for each formal parameter.

2.2 POST

Entered via a return jump to translate one R-list entry into a COMPASS line image. Also uses the instruction size (15 or 30 bits) and the label field to maintain the length of the sequence and total length of code issued.

Prior to entering POST via a return jump, the following cells within POST must be set:

POSTIFO+0 Contains 1st R-list entry. The R-list op code must have been changed to the

appropriate machine code if the R-list code has a choice of machine codes (i.e., R-list code 10 can be machine code 10 or 22).

POSTIFO+1 Contains the second R-list word (in case of type III R-list).

POSTIFO+2 Contains the descriptor for the R-list entry.

POSTIFO+3 (or I) These three words contain a code for the actual register type.

POSTIFO+4 (or J and number that is to be used.) I is the destination register.

POSTIFO+5 (or K. J and K are the two source registers if there are two).

The following is a list of the register codes:

X0=1	A0=9	B0=17
X1=2	A1=10	B1=18
X2=3	A2=11	B2=19
X3=4	A3=12	B3=20
X4=5	A4=13	B4=21
X5=6	A5=14	B5=22
X6=7	A6=15	B6=23
X7=8	A7=16	B7=24

Generally, each instruction is placed in a string buffer, starting at LWAWORK and working towards FWAWORK. Each word will contain a right justified character, and the string is terminated by a zero word. After each R-list entry is formed, the information is packed and added to the list of line images to be transferred to the COMPASS file. Any extra information that needs to be added, such as trace back information or SUB macro references, is added at this time. If it seems as though there is insufficient room (40 words) to POST the next instruction, POST exits with X6 negative. This looking ahead is done so that OPT does not release needed flags if POST happens to fail on the last entry in a sequence.

2.3 CLSPOST

Entered via a return jump to transfer the COMPASS line images for a sequence to the COMPS file via WRWDS2. Also, the number of SUB's and DELAY's issued for each formal parameter in this sequence is added to the total for the parameter. This sum and subsum are maintained in word 2 of the 2 word symbol table entry for the parameter.

2.4 POSTIFO

POSTIFO is declared an entry point so that the routine calling POST can preset the information POST needs and is not to be entered as it is a data area.

2.5 FCHAR

FCHAR is declared an entry point in POST so that OPT can force instructions upper as it wishes.

FCHAR is set by OPT and is either a blank or a plus in display code.

POST always sets FCHAR blank before it exits.

2.6 PARCEL

Contains an instruction parcel count 0, 1, 2, 3 or 4. It is an entry point because OPTB will add or subtract 1 parcel when it adds or deletes a NO instruction from the code.

2.7 OVERS

Contains a parcel count 0, 1, 2 or 3 of the instruction word from the immediately prior sequence. This is necessary to maintain the correct length of issued code.

2.8 COVD

Contains the address of the location that holds the length of code issued and is either the address of CODE for CODE. block or VARDIM for the VARDIM. block.

3.0 Diagnostics

POST produces no diagnostics. It will translate the information it is given. If the information is bad, it

will go into the COMPASS file improperly and will promptly be diagnosed by the assembler.

4.0 POST is called by OPT. It expects the information described under 2.2 to be preset.

5.0 Structure

5.1 Initialization

POST starts by converting the contents of the I, J, K cells, which OPTB has set to codes for specific registers, to display code. The initial values to these cells I, J, K are given on the preceding page. After they are converted to display code, the register type will be in the lower 30 bits of the word, the register number in the upper 30 bits.

FCHAR is transferred to LWAWORK using X7, as the store register. A7 is therefore initialized and it is used as the store register and pointer for the strung out COMPASS line. A blank is added to the string and this will eventually appear in column 2. Thus, we start with a '+b' or a "bb".

The R-list descriptor is read up to determine the type of instruction to be converted and each type is processed in the following several manners:

5.1.1 TYPE1 Only a maximum of three registers. The op code is closely examined to find the proper instruction and the letters for the instruction and the register assignments are added to the string.

5.1.2 TYPE2 Only a mask or set instruction needed. Which one is determined and the letters, register and constant are added to the string.

5.1.3 TYPE3 The proper letters are added according to the op code. The IH field is then processed which takes into account the necessity for issuing a SUB macro reference.

5.1.4 TYPE4 Only jump can be compiled in this case. The proper letters are added and the symbol is processed.

5.1.5 PACK This is the routine that the above four routines, go to when they have finished placing a card image backwards in the string. The string is packed to look like a card that comes through the card reader and thus it is terminated either be a zero word or by the last word having 12 bits of zero in the lower 12 bits.

After the instruction has been added to the temporary COMPASS file, the substitution macro reference or delay substitution macro reference, is placed in the COMPASS file if necessary. The substitution macro is referenced any time a 30 bit instruction has a formal parameter as part of its address field. The delay macro is referenced only when the same formal parameter is used in both the upper and lower parts of the same word. Traceback information is also output at this time if a 60 bit return jump was just processed. POST then exits with X6 positive unless there are not 40 more words of working storage left (the maximum amount of storage POST could need to process one R-list entry.) If not enough room is left, X6 is negative.

6.0 Formats

6.1 The code at PACK8 may be changed during execution to facilitate the necessity of putting out a delay substitution macro reference for the following situation:

Code desired:

```
SA1  FP1
SA2  FP2
```

where FP1, FP2 are formal parameters.

Due to the way the address substitution takes place, the following COMPASS output must be produced:

```
SA1   FP1
SUB   FP1
SA2   FP1+1
DELAY FP1
SUB   FP1,1
```

6.2	APCA	Contains the CA field of the instruction presently being sent to COMPASS.
6.3	APLST	Contains zero or the name of the formal parameter just formed in a K field.
6.4	APLAST	Contains the name of formal parameter last used in a K field.
6.5	CALAST	Contains the last CA field put into a sub macro reference.
6.6	CMPSPR	Contains the starting address of the line images for the COMPASS file (OPNPOST sets it to FWAWORK).
6.7	CMPSPS	Contains the address in which COMPASS line images stored.
6.8	TRACEP	Set non-zero on a 60 bit return jump. Indicates traceback information should be added. This is done after the RJ instruction has been converted to a COMPASS line image.
6.9	COMPASS	Line string each line for COMPASS is strung out, one character per central memory word, backwards starting at LWAWORK. When the entire instruction has been translated, it is packed 10 characters per word and added to the temporary COMPASS file.
6.10	Temporary	COMPASS file starts at FWAWORK and grows forward toward LWAWORK. Contains COMPASS card images for all instructions translated between a call to OPNPOST and one to CLSPOST.
6.11	Sequence	SUB and DELAY count is held in bits 41-52 of word 2 of the 2 word symbol table. The total SUB and DELAY count is held in bits 19-36 of the word 2 symbol table entry for the formal parameter.

APLISTE

1.0 APLISTE - Aplist Expander is a part of Pass 2.

Converts the APLIST to card images and places them into the COMPASS file.

Outputs a SUB macro reference for any formal parameters that were not referenced in the subprogram and maintains the length of the relocation base associated with each formal parameter.

2.0 APLISTP

APLISTP processes the accumulated APLIST and outputs it to the COMPASS file.

APLISTP is entered by a return jump from CLOSE2 and exits through its entry point. If there is insufficient room to process the APLIST, at least as much working storage as there is APLIST, it exits to PUNT.

The specified APLIST is rebuilt and grouped so that each list is contiguous in memory and in the proper order in which they should appear. These grouped lists are then examined and any that can be eliminated and combined into another are. The list is then converted to COMPASS line images and placed in the COMPASS file.

3.0 No diagnostics are produced.

4.0 The following cells are referenced and are expected to contain the indicated values.

AP1. The address of the first entry in the APLIST.

APLAST. The address of the last entry in the APLIST.

HIGHORD. The APLIST ordinal of the largest APLIST.

FWAWORK. First word address of the working storage.

SYM1. Start of the symbol table.

PGM. The program/subprogram indicator.

SUBREF. Flag that is non-zero if the ADDSUB code has been put out.

S.NEC. Flag set if it is necessary to issue at least one word of ST. storage.

AOSUBO
CMPSPR
CMPSPS
ADDSUBN
CLOSPOST

All referenced for the ADDSUB coding.

WRWDS2 is called to transfer the converted APLIST to the COMPASS file.

5.0 The APLIST processing is divided into three phases. The format of the table built in the processing is under 6.0.

5.1 First, the jumbled APLIST is grouped so members of each group appear in contiguous memory cells in the proper order. This is done in the following manner.

The present APLIST number that is being grouped is set to one.

If the present APLIST number is greater than the highest APLIST number (HIGHORD) the grouping of the APLIST is complete.

Otherwise, the entire APLIST as it appears is searched and each entry that has the same number as the present APLIST number is combined into one word and placed in the grouped APLIST. The grouped lists start at FWAWORK and grow toward APLAST.

After all entries in a group have been extracted from the jumbled APlIST, an entry is made in the GAPL giving the FWA and LWA of this group. This table starts at AP1 and grows toward APLAST.

The present APLIST number is bumped by one and processing continues at 5.1.2.

5.2 The grouped APLISTs are then examined to see if it is possible to eliminate any of them. This is done by

comparing two lists (a primary and a secondary) at a time. If it is found that we hit the end of either list before finding a non-equal entry, the group that we hit the end of can be eliminated. In this case, the following is done:

The GAPL entry for the group which is to be eliminated is set to minus 1.

In the last examined entry of the non-eliminated group, a GN (Group Number) is placed in the upper 12 bits. If it has a GN already, it is used rather than generating a new one.

The APLIST number of the eliminated group is combined with this GN and added to the OUT table (starts at end of GAPL - grows towards FWAWORK). This table is used when translating the APLIST to COMPASS line images.

If the eliminated group was considered the first or prime group, its grouped APLIST is searched for any non-zero GN fields. If any are found, the OUT table is searched for each such GN and this GN in the OUT table is then replaced by the new GN. When translating to COMPASS, a non-zero entry in the GN field of an APLIST entry says, "Before this entry is translated to a COMPASS line image, the APLIST numbers that are linked to this GN in the OUT table must be placed in the temporary COMPASS file first".

This is continued until all APLIST groups have been compared.

5.3 The list is then translated into COMPASS line images.

An entry is picked from the GAPL to find parameters for a list that is to be translated to COMPASS line images. Negative entries are ignored and a zero entry indicates the end of the GAPL.

When a usable GAPL entry is found, the number of this list is placed in the temporary COMPASS file as [APn. BSS 0 where n is the APLIST number.

The actual list is then output to the temporary COMPASS file. At selected times during this transfer, the available core is checked. If it is running short, the temporary COMPASS lines that have been formed thus far

are transferred and the pointers are reset. Each member of an APLIST is transformed in the following manner:

If there is a non-zero GN field, the OUT table is searched. Each entry in the OUT table that has this GN also contains an APLIST number of an eliminated APLIST that should be output to the temporary COMPASS file before this APLIST entry is processed.

Then the I field is examined. It can indicate either a statement temporary, a variable, or it can be an indication of non-standard return. Default goes to b.

a) Non-standard returns, (I=7) places a VFD 60/0 in the temporary COMPASS file. This is done so that substitution of actual parameter addresses at execution time terminates before any non-standard return addresses are substituted.

b) Symbols, (I=0) the symbol is read from the symbol table. If it is a formal parameter, a flag is set saying it is necessary to output a substitution macro reference. The symbol along with any constant add in (Cfield) is placed in the temporary COMPASS file followed by a SUB macro reference if necessary.

5.4 Example:

Consider a subprogram as follows:

```
      SUBROUTINE X (A,B,C)

      DIMENSION A (50)

      CALL T (A(40), B,C+1,24)

      CALL S (X11, Y, A)

      CALL G (Y, A)

      END
```

APLIST would be placed in the COMPASS FILE IN THE FOLLOWING format:

```
      [AP1. BSS 0

      VFD       60/A+47B
```

```
VFD       60/B
SUB       B
VFD       60/ST1.          (Statement Temporary)
VFD       60/CON.+nB
VFD       60/0
[AP2. BSS 0
VFD       60/X11
[AP3. BSS 0
VFD       60/Y
VFD       60/A
SUB       A
VFD       60/0
```

6.0 Table Formats

6.1 APLIST INPUT

APLAST

```
VFD  30/0,12/I,18/H1                    word 2
```

Ap1.

```
VFD  12/2nnn,18/CA,14/0,16/NO           word 1
```

nnn is the number of cells back from the present location in the jumbled APLIST that contains the next member of this APLIST (nnn=0 indicates the end of a particular APLIST).

NO, the APLIST group that this entry belongs to.

CA, I, H1, have the usual meaning.

6.2 GROUPED APLIST

```
FWAWORK     VFD  12/GN,18/CA,12/I,18/H1
```

(GN initially zero)

```
            APLAST
```

6.3 GAPL, Grouped APLIST parameters list.

```
            APLAST

AP1.        VFD  24/0,18/FWA,18/LWA
```

FWA first word address of this group.

LWA last word address of this group.

6.4 OUT Out Table

```
            VFD  24/0,18/ON,18/GN
                 GAPL

AP1.
```

GN is a group number

ON is the number (output number) of an APLIST group that has been eliminated.

6.5 CMPSTR, contains the first word address of the temporary COMPASS file.

6.6 APNAME, contains name of last symbol if it was a formal parameter.

6.7 APCA, CA field of last symbol output if it was a formal parameter.

PRE

1.0 General Information

PRE is the main controlling routine for PASS 2.

PRE Functions.

a. Call the macro expanders (MACROE, PRODB, PRODE, PROIXFN).

b. Expand all R-list into three-word form and define the sequence.

c. Put out inactive labels, formal parameter names, variable dimension storage and the END line to COMPASS.

d. Call the optimizers (COPY, SQUEEZE, PURGE, BUILDDT, OPTA).

e. Accumulate APLIST entries and call APLISTP.

f. Issue BSS storage for statement, DO, and optimizing temporary storage.

2.0 The only entry point for PRE is PRE.

2.1 A jump to PRE causes code generation for all R-list on the file.

2.1.1 Calling Sequence:

```
        JP          PRE
```

a. AP1
   APLAST — address of end of symbol table (SYMEND)

b. VAR1
   VARLAST — SYMEND - 100B

c. MACORG — EQU to lowest significant macro number in MACROX.

d. FFLAG — =0 for standard mode

e. PROGRAM — contains transfer address, if main program

2.1.2 Processing Flow

After reading in a fixed number of words at a time from the R-list file, each positive entry with a positive opcode is expanded into 3-word form. The descriptor ST field of each is examined to detect one of the following cases:

a. If ST = 7, a macro ordinal switch determines which of the macro expanders to call. A PRODE call terminates the sequence after the DO end jump.

b. If ST = 0, the entry is considered "normal R-list" and is added to the sequence at PS2CTL by a call to ADDTOSQ.

c. If ST = 1-6, the entry is directed to a jump table, where the following are detected:

1. Sequence terminators are: DO end jump, entry statement, active label, and unconditional jumps.

   They cause generation of an end of sequence entry (100). Normally, PROSEQ is called for code generation. However, in a well-behaved DO, sequences are allowed to accumulate before this call.

2. APLIST entries are added to the list by ADDTOAP.

3. The end of R-list entry triggers PASS 2 closing (CLOSE2). If VARDIM is present, it is coded at this time.

3.0 Diagnostics Produced

3.1 Fatal to compilation

INSUFFICIENT MEMORY.

3.2 Fatal to execution: none.

3.3 Informative

MORE MEMORY WOULD HAVE RESULTED IN BETTER OPTIMIZATION.

4.0 Environment

MACWRDS - Local to READRL. Set by macro expanders to the number of words placed in MACBUF.

FWAWORK - Local to PRE. Must contain current lowest unused work storage address.

LWAWORK - EQU VARLAST. Must contain current highest unused work storage address.

WELLBE Set by DO processor to 1 if a well-behaved DO loop is in process, else zero.

5.0 Processing

5.1 Setup

The loop table, APLIST table, and VARDIM tables are allocated. If OPT=2 is selected, the OPT file is opened in order to retrieve the index. Then FWAWORK is advanced over the index area. At this point, an initial end of statement marker is placed at FWAWORK.

5.2 Main loop

READRL is called to obtain the next item from the R-list file. For a zero statement type field, we add the R-list1, R-list2 and descriptor to the sequence under construction. Then the process is iterated. If the ST value is 7, we must expand the macro returned by READRL. ST values from one to six are processed differently depending on their location. Within a well behaved DO loop, end of statement/sequence markers (ST=1) and unconditional jumps are treated differently.

5.2.1 ST=1 (End of Statement/End of Sequence)

Within a well behaved DO, end of sequences are appended to the current sequence and sequence accumulation continues. End of statements are appended and the card

count cell is updated. However, if OPT=0, sequence processing is forced since a statement is a sequence in fast compile mode.

In non-well behaved regions, an end of sequence terminates the sequence and calls PROSEQ to process it. An end of statement is appended and CARDCT updated. Sequence processing will be forced when twenty statements are accumulated or the accumulated SRLIST exceeds one-quarter of working storage.

5.2.2 ST=2 (APLIST)

The two words (RLIST 1 and 2) are added to the APTAB and the VARDIM table plus F table (if OPT=0) are moved down to provide more room.

5.2.3 ST=3 (Unconditional jump)

In a well behaved DO, the jump is added to the sequence, and the sequence is terminated here. However, sequence accumulation continues. For a non-well-behaved DO region, a jump is added, the sequence terminated and PROSEQ called.

5.2.4 ST=4 (Label)

Inactive labels are issued to the COMPS file as label EQU * lines. For active labels, processing is the same as for unconditional jumps.

5.2.5 ST=5 (End of R-list)

Terminate the current sequence, call PROSEQ and initiate cleanup procedures.

5.2.6 ST=6 (Entry)

Forces sequence termination and processing after it has been appended.

5.2.7 ST=7 (Macro Reference)

If the macro belongs to the group of special macros (ordinals less than 100B), processing is split off. These macros are DO begin and end, index functions and loop begin and end (in OPT=2). For a DO begin, PRODB is called. DO ends result in a call to PRODE, and index

functions are processed by PROIXFN. All normal macros are expanded by a call to MACROE. Then the resulting expansion will be read in the main loop by calls to READRL.

5.3 Loop Begin Processing

WELLBE is set to force sequence accumulation end a loop flag indicator is turned on. Contained in the loop begin macro is a list of register candidates. This list is moved above the APLIST and VARDIM tables where it is held until loop end is found; then control reverts to the main loop.

5.4 Loop End Processing

If not in loop mode, we immediately discard the macro and revert to the main loop. Then SELECT is called to perform candidate selection. The candidate table is sorted by frequency of use. Then the min (4,number of candidates) are chosen for register allocation. If a candidate has no uses, it will be discarded. Next, the body of the loop is moved up to make room for register definition and register lock R-list in the prologue. Then the candidate registers are made unavailable to OPT as scratch registers.

The BIND flag is then set on. With this set, OPT will not call POST to issue COMPASS lines. Then PROSEQ is called. Upon return the number of registers assigned is printed. Now that it is certain that the sequence can be coded using the available registers, OPT is reset, POST is allowed to process code lines and PROSEQ is called again on the same sequences. Upon return, R-list to post store necessary, values is generated and PROSEQ called. Finally, the locked registers are released and control reverts to the main loop.

5.5 OPT=2 Failure

When PROSEQ fails on a sequence with locked registers, an attempt is made to reduce the number allocated. For each failure, another register is released. When all locked registers have been freed, processing reverts to the normal fail paths.

6.0 Formats

6.1 The R-list Descriptor

VFD 1/LD,1/SR,1/JP,4/F1,4/F2,2/TY-1,5/FT,1/K,1/IG,
10/USES,1/JK,1/RS,20/DO,1/FE,1/SZ,3/ST,1/KL,1/SQ,1/CM

LD set on long and short loads.

SR set on long and short stores, including register store.

JP set on all jumps.

F1 function unit used.

F2 second possible function unit, = F1 if only one possible.

TY-1 type (I-IV).

FT 6600 function time, = 10B for loads, 12B for stores.

Unit	Indicator
Branch	1
Boolean	2
Shift	3
Add (Integer)	4
Add (Floating)	5
Multiply #1	6
Multiply #2	7
Divide	10
Increment #1	11
Increment #2	12

K set if hardware instruction includes no k field.

IG used by DOPRE.

USES set by BUILDDT, used by OPT.

JK set if hardware instruction includes jk field.

RS subsequent register store bit. Used in OPT.

DO this area is used by DOPRE.

FE set for jumps and stores.

SZ — set for 30 bit instruction.

ST — EOST = 1, APLIST = 2, unconditional jump = 3, label = 4, end of R-list = 5, entry = 6.

KL — initially = 0, set by SQUEEZE and SQZVARD when instruciton killed.

SQ — set for squeezable instructions.

CM — set if operands commutative.

6.2 Flag words

6.2.1 In PRE (for PROSEQ)

VARGLAG = 0, set to 1 at VARDIM time.

6.2.2 In READRL

MACWRDS = 0, set to number or words placed in MACBUF.

6.2.3 In ADDTOSQ

MACREF (EQU R-list) LWA+1 of parameter words.

6.2.4 In ADDTOAP (for APLISTP)

APLAST LWA, APLIST+1.

6.2.5 In PROSEQ

NORLIST number of entries to process.

FWASEQ FWA of R-list to process external to PROSEQ.

LASTR first word of last single entry sequence encountered.

LASLBL if LASTR = label, LASLBL = LASTR.

6.2.6 In SQZVARD (for PROSEQ)

LENGTH number of cells in redundant VARDIM store code list for COMPASS.

STORBUF FWA or redundant store list.

READRL

1.0 General Description

READRL is called to return the next R-list instruction from the R-list file. Either a macro or an R-list1, R-list2, and descriptor will be returned.

2.0 Entry Points

2.1 MACBUF

Fixed size area used for macro expansion.

2.2 NORNXT

Address of the next R-list word to be read.

2.3 NOREND

Address plus one of the end of R-list words in the buffer.

2.4 MACNXT

Address of the next R-list word in the macro expansion buffer.

2.5 MACWRDS

Holds the number of words in the macro.

2.6 TWA1

Three word area holding the descriptor, R-list1 and R-list2 on exit from READRL.

2.7 DESCR

Address of the start of the descriptor table.

2.8 USETAB

Address of a table used in OPT=0 for accumulating user counts.

2.9 READRL

Primary entry to return an R-list item.

3.0 Diagnostics and Messages

MEMORY OVERLFLOW IN READRL

4.0 Environment

Externals

MACREF	Same cell as the external R-list
RDWDS	Routine to read information from a file
IXLOC	Current index location for OPT=2
OPTLVL	Holds the OPT=n value from the FTN card
F.OPT	Fet for the OPT file
USEDESC	Table holding the intra-macro user count
BASE	Base R number of the F-table for OPT=0
DOMACK	Origin of the DO macros
USEBUF	Table in MACROX holding uses counts
MACORG	Base of the normal macro numbers
RDRLF	Flag to ADDTOSQ to make an F-table entry
FWAWORK	Holds the first word address of working storage
RLIST	Next, address to store an R-list entry triplet
LOOPFLG	Set in OPT=2 if inside a well behaved loop and register allocation is specified
CNDTAB	Candidate table for OPT=2 frequency counts
PUNT	Error exit for memory overflow

5.0 Processing

5.1 Main Loop

a. Call READAW to read a word.

b. If the word returned is negative, go to a.

c. Extract the opcode of the R-list word. If it is positive, this is an R-list instruction. Go to o.

d. Bias the opcode by the macro origin and check to see if this is a DO macro. If so, bias it by the origin of the DO macros.

e. Store the first word of the macro at FWAWORK.

f. Increment FWAWORK by the number of words in the macro and check for memory overflow.

g. Copy the remainder of the macro to working storage. This includes manipulation of the NORNXT pointer and a possible need to replenish the working storage buffer if the macro text is not all present there.

h. Exit if OPT=1 or 2.

i. Save the number of intermediate R numbers as the first entry in USE TAB. If no intermediates, go to l.

j. Extract a six bit uses count. Decrement number of intermediate R numbers. Store the count in the next cell in USETAB.

k. If more intermediate uses, go to j.

l. If no formal R numbers, exit from READRL.

m. Extract a six bit uses count. Decrement number of formal R's. Store the count in the next cell in USETAB.

n. If more formal R numbers, go to m. Else exit READRL.

o. If this R-list instruction did not come from a macro expansion, go to p (zero uses at this point).

p. Fetch the corresponding uses count from USEBUF.

q. Or the uses count with the descriptor and store the descriptor in TWA1.

r. Isolate the R-list type from the descriptor.

s. If OPT=1 or 2 or this R-list is not from a macro, go to aa.

t. If this instruction is not a store, go to aa.

u. Set preceding a store indicator.

v. If this instruction is not a register store, go to z.

w. Set preceding a register store indicator.

x. If the register store is on a formal R number other than A0, go to z.

y. Insert the precedes a register store indicator into the descriptor of the last previous R-list instruction. Go to aa.

z. Using the F-table and the R number being stored, locate the descriptor of the defining R-list and set the precedes a store indicator.

aa. If this is not a type 3 R-list, set TWA3 to zero and exit READRL. For type 3, call READAW to get the second word in TWA2.

ab. Exit if READRL is not operating in loop mode for OPT=2 (LOOPFLG is non-zero).

ac. If the opcode is not a load or a store, exit READRL.

ad. Extract the IH field from R-list2 and search the candidate table for a match. If no match is found, exit READRL.

ae. Increment the total number of uses. Then exit if it is a load instruction.

af. Increment the number of uses as a store and exit READRL.

5.2 READAW - Read a Word

a. If the macro buffer is not empty, return a word from it. (If the word is negative, complement it and set RDRIF (OPT=0 processing) to cause ADDTOSQ to make an F-table entry). Set R-list from macro flag. Go to d.

b. If OPT=2, check the number of words left. If this count is zero, call READRAN to set up the next record from the file FTNOPT, update the words left and go to g.

c. Normally, try to extract a word from the NOR1 buffer. If it is empty, go to g.

d. If this is OPT = 0, go to f.

e. Set R-list not from a macro flag.

f. Return the word read in TWA3 and X7. Exit READRL.

g. Call RDWDS to replenish the NOR1 buffer. Update the pointers and go to c.

5.3 READRAN

a. Using IXLOC pick up the next entry from the index.

b. Place the PRU number in the OPT FET plus six word.

c. X1 holds the number of words in the record.

d. Wait for file activity to cease and clear the end of record condition.

e. Set the file on which to do input to the OPT file. Exit READRAN.

6.0 Table Formats

6.1 Descriptor Table

Bit Number	Field
59	Load Instruction
58	Store Instruction

57	Jump Instruction
56-53	Functional Unit 1
52-49	Functional Unit 2
48-47	Type of R-list
46-42	6000 Function Time
41	Set if instruction has no k field (set for jk types)
40	
39-30	Uses count
29	Set for instructions with a jk field
28	Set if the instruction may not be killed
27	Object of a store
26	Object of a register store
25-8	Field used by DOPRE
7	Set for stores and jumps
6	Set for 30 bit instructions
5-3	Special type field 1=end of statement, 2=APLIST, 3=unconditional jump, 4=label, 5=end of R-list, 6=entry
2	Kill bit. Set for dead instructions
1	Set if the instruction is squeezable
0	Set if the operands are commutative

DOPRE

1.0 General Information

Task Description

The second pass DO processor examines DO begin and DO end macro references, standard index function macro references and all R-list instructions generated within the innermost loop of a DO nest provided the loop is well behaved (see section 8.2). R-list instructions are generated to count DO loops, reference standard index functions, and to materialize the control variable when necessary. The R-list is generated by considering the optimum use of B registers and all code in the DO loop is altered to take advantage of B register assignments where possible.

2.0 Entry Points

2.1 PRODB

2.1.1 PRODB is referenced by PRE when a DO begin R-list macro is encountered in the R-list buffer. For OPT=1, 2 and a well behaved DO, begin macro flags are set for ensuing calls to PROIXFN and PRODE. For OPT=0 and non well-behaved DO's PRODB generates R-list instruction necessary to initialize the loop.

2.1.2 The calling sequence to PRODB is a return jump to PRODB. Flags and addresses needed are:

OPTLVL - holds the value specified by OPT=m on the FTN card.

MACREF (RLIST) - address of the DO begin R-list macro in memory.

FWAWORK - address of the working buffer where R-list items may be stored. Overflow is checked against VARLAST.

2.2 PROIXFN

2.2.1 PROIXFN is referenced by PRE when pseudo R-list for a standard index function is encountered in the R-list buffer. PROIXFN breaks the index function into a table of terms and either preserves this table for PRODE by replacing part of the pseudo R-list (standard) or generates index function (first level optimization).

2.2.2 The calling sequence to PROIXFN is a return jump to PROIXFN. It is expected that the following variables in the communications region will be properly set:

MACREF (RLIST) - Address of the DO begin pseudo R-list macro memory

VARLAST - Address of the next available call in the VARDIM buffer.

FWAWORK - Address of the working buffer where R-list items may be stored. Overflow is checked against VARLAST.

2.3 PRODE

2.3.1 PRODE is referenced by PRE when a DO end R-list macro is encountered in the R-list buffer. For an ill-behaved loop (no optimizing attempted), MACROE is called to generate the instructions for the bottom of a DO loop given the DO end R-list macro. If the compiler is in standard mode and the well-behaved flag is set, then a series of ten scans or phases of optimization is executed. These scans and their functions are:

I. Scan all R-list between the DO begin and the DO end R-list macros selecting candidates for available execution time B registers from among the load, store and set R-list instructions and the pseudo R-list macros for standard index functions. Scan I compiles candidate information in two word entries and places them in the A table (see section 6.0 - FORMATS). A table items are linked together on the basis of type: constant, address, index function or variable increment (see section 8.0).

II. Scan the A table computing the cost in instruction parcels if a B register is not assigned this candidate. Assuming that the lowest number of parcels used generates the most efficient object code, B registers are assigned to the candidates

having the largest cost. Scan II computes this cost for all constant, address and increment entries in the A table, marking each as to type and as a candidate. In each index function chain, the member which appears in the largest number of instruction sequences is given a cost and marked as a candidate; the differences between the candidate and other group members are filed as candidates.

III. Scan the A table and assign the B registers by generating a B table of up to seven entries. Registers are assigned by largest cost until the A table is exhausted or the B table is full. If a B register is assigned to a constant, the constant chain is searched and wherever possible constants are marked to use the sum or difference of presently assigned B registers. If a B register is assigned to an address, the address chain is scanned and the difference between the candidate and other addresses are filed as candidates.

IV. Scan the B table and decide from register assignments which one of 12 methods should be employed to count the DO loop. The decision is based primarily upon the contents of the B registers. (See section 8.0 for loop counting methods.)

V. Scan the B table and mark the candidates in A with the register (or registers) that have been assigned to them.

VI. Scan the B table for A assignments. If the A table entry is a constant, address or difference type candidate set up the R-list instructions to load this register at the top of the DO loop.

VII. Scan the A table for increments and index functions generating the initializing R-list instructions to pre-compute variable increments and to initialize and increment index functions.

VIII. Generate the DO loop counting code selected by Scan IV. R-list instructions are generated to initialize and increment the loop count and, if necessary, the control variable. The 12 methods of counting are shown in section 8.0.

IX. Scan all R-list between the DO begin and DO end R-list macros creating R-list references to B registers when applicable. The general R-list given by pass I will be tailored to the B register assignments made by previous scans and will replace the general R-list passed on by PRE. The pseudo R-list index function macros will be expanded into the proper sequence of instructions generated for double precision and complex arrays.

X. Separates top of the loop R-list from end of the loop R-list and inserts the body of the loop. This is done by moving the TOP-END buffer just below VARDIM, putting the referencing R-list just below this. Top of the loop instructions are then extracted and moved to the beginning of the R-list buffer where the DO begin macro was. The body of the loop follows, then the end of the loop instructions. "Ignore" op codes are squeezed out and any remaining negative op codes cause generation of normal R-list to replace them.

2.3.2 The calling sequence to PRODE is a return jump to PRODE. Addresses needed are:

MACREF (R-list)	Address of the DO begin R-list macro in memory.
VARLAST	Address of the next available cell in the VARDIM buffer.
FWAWORK	Address of the working buffer where R-list items may be stored. Overflow is checked against VARLAST.

3.0 Diagnostics

None

4.0 Environment

Not applicable

5.0 Structure

5.1 CANON

CANON generates and orders the table of index function terms (T) given the standard index function pseudo R-list macro produced by ARITH in pass one of the compiler. Each index function may have as many as five terms dependent upon the combination of variables and constants used in the subscript. CANON uses TERM and FILET.

5.2 TERM

TERM unpacks a single subscript. If the subscript contains a variable, exit is made to the second word following the return jump to TERM, otherwise the first.

5.3 FILET

FILET makes the actual entry of a term in T, combines terms with like variables, orders the table in decreasing order using all 60 bits as key for the sort and leaves the address of the next available location in the cell TN.

5.4 IXFN

IXFN is called whenever a standard index function is encountered in the R-list. IXFN then searches the integer definitions following the DO end macro to see if the subscript uses any of the variables that are redefined within the loop. If so, and it includes the control variable, the control variable is marked for materialization. If not, and the index function has not appeared before, it is filed as a candidate in the A table.

5.5 LINKA-NSRTA

5.5.1 LINKA files candidates for B registers in the A table, if not already there and links them to other candidates of the same type (address, constant, etc.). If a candidate has already been filed, then the sequence in which it appears is noted for use in determining the value of having a B register assigned.

5.5.2 NSRTA files a candidate in the A table without linking and without checking for prior entry.

5.6 REF

Given the OC, CA, SO, RI, H2, RF, I, HI fields, REF files a three word type three R-list item in the R-list buffer area of memory. This R-list item will generate a reference to an array element at object time. If the array is double precision, a second type three R-list item will be placed in the buffer.

5.7 MRKIXFN

MRKIXFN chains through all index function entries in the A table by group, selecting the most popular member of each group (appearance in most sequences) and marking that member as a candidate for a B register and as a member of a group.

MRKIXFN then calls FORMDIF.

5.8 FORMDIF

FORMDIF inspects address and index function groups and if a difference is found between the inspected item and the head of a group does the following:

1) If the difference is constant, the difference is filed in the constant chain.

2) If the difference is symbolic, then an item is filed containing both symbols and the constant difference (if any).

5.9 EVALCON

EVALCON evaluates a constant entry in the A table placing the number of parcels saved by using a B register in the value field of the A entry. It also marks the entry as a candidate and as a constant.

5.10 MRKCON

MRKCON scans the linked constant candidates in the A table marking each as a candidate and constant and computing the cost in parcels if a B register is not assigned the candidate.

5.11 MRKADR

MRKADR searches the address chain of the A table for the most popular candidate (used in most sequences) and then

marks that entry as a candidate having a group and constant. The cost of not assigning a B register is stored in the value field of the candidate.

5.12 MRKINC

MRKINC scans the linked increment entries in the A table marking each as a candidate and computing the cost of not having a B register.

5.13 SRCHC

SRCHC is called to determine if a constant entry in the A table may be formed using constants already assigned to B registers. Given a constant which is the sum or difference of assigned constants, SRCHC chains through the constant chain for a candidate equal to it. If found, the two are related by setting the appropriate bits in the REG, REG2, and NEG fields in the A table entry found.

5.14 TOPB

Given a one or two word R-list item, TOPB adds a descriptor word and files a three word R-list item in the TOPEND buffer after insuring that the buffer won't overflow.

5.15 ENDB

Given a one or two word R-list item, ENDB adds a descriptor word and an end flag and files a three word R-list item in the TOPEND buffer after insuring that overflow won't occur.

5.16 VARB1

Given a one word R-list item, VARB1 stores it in the VARDIM buffer.

5.17 VARB2

Given a two word R-list item, VARB2 stores it in the VARDIM buffer.

5.18 TDOWN

TDOWN displaces the T table lower by one storage address.

5.19 MOVETR

MOVETR moves the T table (through and including the first entry of all zeroes) to the last six words of the 12-word standard index function pseudo R-list, (see section 6.2 - the second table on the page).

5.20 MOVERT

MOVERT moves the last six words of a standard index function pseudo R-list item to the first six words of the T table. (See section 6.2.)

5.21 GENT

GENT generates the R-list to compute and reference a standard index function. This code replaces the standard index function pseudo R-list produced by ARITH in pass one of the compiler. GENT calls GENVAR.

5.22 GENVAR

GENVAR generates the R-list to compute the variable part of an index function. This code appears at the loop top or point of reference depending on the index function. Non-variable computations appear at the program top. GENVAR calls SUMC.

5.23 SUMC

SUMC generates the R-list to add up the coefficients of a single written variable and places this code in the VARDIM buffer. SUMC calls COEFF.

5.24 COEFF

COEFF generates the R-list to multiply the factors of a coefficient and stores the generated R-list in the VARDIM buffer.

5.25 DOONE

DOONE generates a reference to a subscript quantity based on the contents of the B registers. The generated R-list replaces the standard index function pseudo R-list.

5.26 GENALL

GENALL moves R-list down in memory and generates the R-list to compute and reference redefined index functions. GENALL calls GENT.

5.27 MATC

When the control variable must be materialized (updated in memory) during the course of a DO loop, MATC makes entries in the A table to allow the loop parameters to compete for B registers.

5.28 MCOST

MCOST determines the minimum cost for DO loop counting considering all possible situations available methods.

5.29 CCOST

CCOST determines the cost of counting a DO loop by using the upper limit.

5.30 LOADX

Given an A table item, LOADX generates R-list instructions to load an X register with the item.

5.31 LOADY

Given a loop parameter, LOADY generates R-list instructions to load a register R with it.

5.32 STOREX

When counting a DO loop in memory and the count has been updated in an X register, STOREX generates R-list instructions to store the X register in memory.

5.33 LOADBMC

Generates R-list to load the first limit and subtract the second limit of the DO statement. LOADBMC calls LOADY.

5.34 COUNTR

Generates the R-list to compute (C-B)/D for counting the DO loop. Calls CONCNT.

5.35

Generates the R-list to compute (B-C)/D for counting the DO loop in the negative direction. Call CONCNT.

5.36 CONTOP

Generates R-list instructions to multiply R by an integer constant in the most efficient manner using shift and add combinations. R-list is stored in the TOP-END buffer.

Generates R-list instructions to multiply R by an integer constant in the most efficient manner using shift and add combinations where R is a variable division or product of variable dimensions. R-list is stored in the VARDIM buffer.

5.37 CONCNT

Determines whether the DO may be counted by a constant or not, i.e., whether (B-C)/D is constant.

5.38 DIVCON

Generates R-list to compute (B-C)/D optimally by checking constants (if any) and shifting or no division at all where possible.

5.39 LMCHK

LNCHK checks DO loop limits to recognize one trip loops and sets flags for section VIII to bypass the generation of loop counting code.

5.40 REDINC

REDINC checks for index functions with same increment having these conditions:

1. Not involved in counting the loop.
2. Not requiring two B registers to form index function.
3. Both index functions in B registers.
4. One incrementing instruction has already been generated.

When conditions are met, REDINC generates code to compute difference of index functions which will replace index function in higher numbered B register. All references to index functions, other than the base, must then be generated as a sum of the base index function and the difference register. At least one parcel per use is saved as the incrementing instruction is eliminated.

Example:

GIVEN CODE			ALTERED CODE	
	SB1 A		SB1	A
	SB2 B		SB2	B
	SB7 A+5		SB2	B2-B1
)AA	SA1 B1		SB7	A+5
	BX7 X1	)AA	SA1	B1
	SA7 B2		BX7	X1
	SB1 B1+1		SA7	B1+B2
	SB2 B2+1		SB1	B1+1
	GE B7,B1,)AA		GE	B7,B1,)AA

5.41 INIDV

INIDV generates the R-list to set an X register to the initial value to be given the induction variable (lower limit of the loop) and then generates the store instruction to materialize the variable in memory.

6.0 Formats

DOPRE is concerned with two levels of optimization:

1) First level optimization where PROIXFN generates R-list to process the standard index functions in a DO loop without benefit of B register optimization. PRODE calls upon MACROE to generate R-list for the bottom of the DO loop.

2) Second level optimization results in PRODB and PROIXFN setting flags and addresses for PRODE to generate candidates for B registers at execution time. In addition, instructions generated within the loop during pass 1 by other processors are changed to take advantage of B register assignment at execution time.

In first level optimization, only the table of index function terms (T) is generated. In second level optimization, a table of candidates (A) and a table of B register assignments (B) is generated in addition to the table of index function terms (T). The A table is a variable length table with two words used for each entry. Candidates may come from constants, increments, addresses, address differences, or index functions. The T table is one word per entry and a maximum of six entries in the table. The last entry must always be zero. The B table is a fixed table of seven entries, one word per entry, and each entry contains information related to the assignment of the corresponding B register. Thus, the third word of the B table designates how B3 was assigned to be used at execution time. Certain fields of both A and B have double usage and these fields will be noted.

In addition to these tables, section 6.0 will be concerned with the standard index function pseudo macro before and after the call to PROIXFN and the TOP-END buffer created by PRODE in the second level of optimization.

The tables and formats follow.

6.1 T TABLE (INDEX FUNCTION TERMS)

```
VFD  4/L,32/V1,1/V2,1/V3,4/V4,18/C
```

L	Set of 1 if V1 if the loop control variable
V1	Base-bias of variable involved in the term
V2	Set to 1 if the first dimension of the array V1 is adjustable
V3	Set to 1 if the second dimension of the array V1 is adjustable

V4 The I part (table number) where C(T) is variable and the H(BIAS) is found in the C field

C Constant multiplier for the variable part of the term

6.2 B TABLE (B REGISTER ASSIGNMENTS)

```
VFD  30/VALUE/R,30/ALOC
```

VALUE Savings in parcels made by the assignment of this B register

R After Scan V of PRODE, R is register number defined as that B register

ALOC Address of the A table entry for the candidate given this register

6.3 STANDARD INDEX FUNCTION PSEUDO R-list MACRO

Before Proixfn

```
VFD  12/OC,18/NWF,11/0,3/TYPE,16/RI
VFD  12/0,18/CA,30/IH of the array
VFD  3/NS,3/P(A,B,C),18/C,18/B,18/A
VFD  60/MC of 1st subscript
VFD  6/0,24/CA,30/IH of variable in 1st subscript
VFD  60/AC of 1st subscript
VFD  60/MC of 2nd subscript
VFD  6/0,24/CA,30/IH of variable in 2nd subscript
VFD  60/AC of 2nd subscript
VFD  60/MC of 3rd subscript
VFD  6/0,24/CA,30/IH of variable in 3rd subscript
VFD  60/AC of 3rd subscript
```

After Proixfn

```
VFD  12/OC,18/NWF,12/0,1/RDIXFN,1/DP,16/RI
VFD  18/CA,12/0,30/IH of the array
VFD  6/0,18/ADPSUB,18/ACHAIN,18/LSUB
VFD  6/0,18/ASUB,18/AINC,18/LINC
VFD  30/0,30/1st dimension of the array
VFD  30/0,30/2nd dimension of the array
     T Table, Entries To-Tc, See T Table Format
```

OC, RI, CA, IH are described in R-list literature.

TYPE — 3 bit type field as used in SYMTAB

NS — number of subscripts

P(A,B,C) — indicates adjustable dimensions for subscript 1, 2 or 3 respectively

C,B, A — constants or IH of variables for dimensions of subscripts 3, 2 or 1 respectively

MC — multiplicative constant in subscript

AC — additive constant in subscript

ADPSUB — if array is double length points to second A table item for this index function

ACHAIN — points to the A table item which heads the chain for this unique index function

LSUB — points to previous unique index function in R-list

ASUB — points to A item for this index function

AINC — points to A table item for this variable increment

LINC — points to the previous index function with unique local terms

NWF — number of words following as part of this macro

6.4 A TABLE (CANDIDATES FOR B REGISTERS)

Word 1 is the same for all entries.

```
Word 1    VFD  1/CAND,1/GRP,1/IXAD,3/REG,3/REG2,
               1/ONE,1/NEG,1/DIFF,18/VALUE/ADIFF,
               12/SEQ,18/LINK
```

For Index Functions

```
Word 2    VFD  12/H,18/CA,12/0,18/RLOC
```

For Increments

```
     Word 2    VFD  30/NAME,12/0,18/RLOC
```

For Constants

```
     Word 2    VFD  60/The 60 bit constant
```

For Addresses

```
     Word 2    VFD  12/H,18/CA,30/0
```

Address Differences

```
     Word 2    VFD  12/H,18/CA,12/0,18/H2
```

H — The ordinal in the symbol table for the variable.

CA — The constant addend or displacement of the variable if any.

SEQ — A field of flags indicating which of 12 possible sequences the candidate is used in. Sequences are indicated from right to left.

LINK — Address of next group member.

RLOC — Address in the R-list for the head of the linked entries.

CAND — Set to 1 if this entry is a candidate for a B register.

GRP — Set to 1 if this candidate is a member of a group.

IXAD — Set to 1 for address type candidates.

REG — Number of the B register assigned (if any).

REG2 — Number of the second B register assigned (if any).

ONE — Set to 1 if increment of index function is constant.

NEG — Relates B registers used to form a sum (0) or difference (1).

DIFF Set to 1 if A item is difference of addresses and cannot take advantage of SB B+B.

NAME I, H values of program temporary or loop temporary.

VALUE Cost in parcels if this candidate is not given a B register.

ADIFF The address of the A table entry created as a difference with this A table entry

7.0 Macros

7.1 ADDRR (BUF, XRJ) Generates R-list to do an integer add of two X registers (R and XRJ) IX(R+1) = X(XRJ)+X(R). R-list fields are OC, RJ, RK, RI found in IADD, XRJ, B7, B7+1 respectively. R-list is stored in an address determined from the index given as BUF. The macro uses TYPEI, OUTBUF macros and the subroutines TOPB, ENDB, VARB1, VARB2.

7.2 DEFBR (BUF, XREG) Defines the B register that will receive the information in the register specified by RI SB(XREG)=R. R-list fields are OC, SO, RI found in DEFINE, LOCKB+ZREG, B7 respectively. Type II R-list is stored in an address determined from the index given as BUF. This macro uses macro OUTBUF and the subroutines TOPB, ENDB, VARB1, VARB2.

7.3 DESBR DEXBR (BR1,XR2,XR3) Sets the NEG, REG and REG2 fields of the first word of an A table entry. Arguments are BR1, XR2, XR3 which represent a B register holding a one or zero, and X registers holding the two B registers assigned to be used as a sum or difference for generating a needed constant. The A table entry is located at L(DESBR) or X(DEXBR).

7.4 IMUL Packs two integers given in X1, X2 and does a double multiply with the result left in X6.

7.5 INCRR (BUF) sets up an 18 bit (short) add of two B registers storing the result back into one of the registers (SB(R-1) = B(R-1) + B(R)). R-list fields used to generate type I R-list are OC, RJ, RK, RI which are given by SADD, B7, B7-1, B7-1. The R-list is stored in

an address determined from BUF. Uses TYPEI, OUTBUF macros, TOPB, ENDB, VARB1, VARB2 subroutines.

7.6 LOADADR (BUF, XIH, BCA) sets up the R-list for the load address instruction SA(R) = (IH+CA) R-list fields are OC, CA, RI, I-HI, found in LOAD, BCA, B7, XIH respectively. The type III R-list is stored in an address determined from the index given as BUF. This macro uses the macro OUTBUF and subroutines TOPB, ENDB, VARB1, VARB2.

7.7 MULCON (BUF) sets up R-list to multiply a register specified by R by an integer constant. Instructions are stored at an address determined from BUF. Uses macros SHIFT, PACK, MULT, OUTBUF and subroutines TOPB, ENDB, VARB1, VARB2, CONTOP, CONVAR.

7.8 MULRR (BUF) sets up the R-list to convert the integers in registers R and R-11, place in registers R+1, R+2, and then does a double floating point multiply with the result ending in R+3.

```
PX(R+1)        R-1,B0
PX(R+2)        R  ,B0
DX(R+3)        (R+1)*(R+2)
```

R-list fields are RK, RI, OC, RK2, OC2, RJ, RK3, RI3, OC3 given by R-1, R+1, PACK, R, R+2, PACK RI1, RI2, R+3, DMUL respectively. The generated R-list is stored in an address determined from the index BUF. Calls macro OUTBUF and uses subroutines TOPB, ENDB, VARB1, VARB2.

7.9 SADRR (BUF, XRJ) sets up Type I R-list to do a short add of two registers with the result going to the third register. Fields needed are: RJ, RK, RI, OC, given by R, XRJ, R+1, and SADD respectively. R-list storage address is determined from BUF. Calls on macros TYPEI, OUTBUF, and subroutines TOPB, ENDB, VARB1, VARB2.

7.10 SETRCON (BUF, BRI, XCON) generates R-list to set a register to a constant value. Fields needed are: IN, OC, RI, given by XCON, SETII, BRI respectively. The type II R-list is stored at an address determined from BUF. SETRCON uses the macro OUTBUF and calls on subroutines TOPB, ENDB, VARB1, VARB2.

7.11 SETADR (BUF, XIH, BCA, XRI, XRF, XH2) generates R-list to set a register to an address. Fields needed are: CA, RF, IH, RI, H2, OC, given by BCA, XRF, XIH, XRI, XH2, SETIII respectively. The type III R-list is stored at an address determined from BUF. Uses the macro OUTBUF which in turn calls upon subroutines TOPB, ENDB, VARB1, VARB2.

7.12 STORADR (BUF, XIH, BCA) generates type III R-list to set A6 or A7 to an address causing a corresponding store of X6 and X7 respectively. Fields needed are IH, CA, OC given by XIH, BCA, STORE respectively. The generated R-list is stored in a buffer determined from the index BUF. STORADR uses the macro OUTBUF which uses subroutines TOPB, ENDB, VARB1, VARB2.

7.13 SRBMB (BUF, XB1, XB2) sets up the R-list for the short subtract and stores the result into the R register (SB(R)=RBTAB+XB1-RTAB+XB2), R-list fields used to generate this type of an R-list item are OC,RJ,RK,RI given by SSUB(67),RBTAB+XB2, RBTAB+XB2, R respectively. XB1, XB2 are indices for the B table which contains the designated R that is assigned the B register. The generated R-list is stored in an area determined from the argument BUF. This macro uses the OUTBUF macro and TOPB, ENDB, VARB1, VARB2 subroutines.

7.14 SRBPB (BUF, XB1, XB2) sets up the R-list for the short add and stores the result into the R register (SB(R) = RBTAB+XB1 = RBTAB=XB2). R-list fields used to generate this type one R-list item are OC, RJ, RK, RI given by SADD(46), RBTAB+RB1, RBTAB+XB2, R respectively. XB1, XB2 are indices for the B table entry that contains the R assigned to B register XB1 or XB2. The generated R-list is stored in an area determined from the argument BUF. This macro uses the OUTBUF macro and TOPB, ENDB, VARB1, VARB2 subroutines.

7.15 BPMB (BUF, XRJ, XRK) sets up R-list for a short load, store or difference with the result going into the RI register. R-list fields used to generate this type I R-list item are OC, RI, RJ, RK given by B4, OP, XRJ, XRK respectively. The generated R-list is stored in an area determined from BUF. This macro uses the OUTBUF macro and TOPB, ENDB, VARB1, VARB2 subroutines.

7.16 SUBRR (BUF, XRJ) integer subtract of two X registers (R and XRJ) IX (R+1) = X(XRJ) - X(R). R-list

fields are OC, RJ, RK, RI found in ISUB, XRJ, B7, B7+1 respectively. R-list is stored in an address determined from the index given as BUF. This macro uses TYPEI and OUTBUF macros and the subroutines TOPB, ENDB, VARB1, VARB2.

7.17 SUBXRR (BUF) same SUBRR expect IX(R+1) = X(R) - X(R-1).

7.18 DIVRR (BUF) integer division of two registers X(R+1) = X(R-1)/X(R). R-list fields are OC, RJ, RK, RI given by IDIV, B7-1, B7, B7+1 respectively. R-list is stored in an address determined from the index given as BUF. This macro uses the macros TYPEI and OUTBUF which uses subroutines TOPB, ENDB, VARB1, VARB2.

7.19 JUMP (BUF, BOC, BRI, BRF) generates the loop ending jump as type III R-list given OC, RI, RF as BOC, BRI, BRF respectively and the IH of the label for the top of the DO is found in the DO-END pseudo R-list generated by pass I. The jump instruction is stored at an address determined from BUF. JUMP uses OUTBUF which uses TOPB, ENDB, VARB1, VARB2.

7.20 TYPE1 (BUF, XRJ) generates a type I R-list instruction from the fields OC, RJ, RK, RI given by B4, XRJ, B7 and X3 respectively. The generated R-list is stored at an address determined from BUF and this macro uses the macro OUTBUF which in turn uses subroutines TOPB, ENDB, VARB1, VARB2.

7.21 SHIFT (BUF, XCOU) sets up R-list to do a shift transmit of R to R+1 and a constant shift left the number of places indicated by register XCOU for R+1. The resulting R-list is stored in memory as determined from BUF using the macro OUTBUF and subroutines TOPB, ENDB, VARB1, VARB2.

7.22 PACK (BUF) generates R-list to pack the exponent zero with the fraction R into a register R+1. Uses OUTBUF, TOPB, ENDB, VARB1, VARB2 to put the R-list at an address determined from BUF.

7.23 MULT (BUF) generates R-list to do a double precision floating multiply of R and R-2 leaving results in the register specified by R+1. Places instructions at address determined from BUF by using macro OUTBUF which uses subroutines TOPB, ENDB, VARB1, VARB2.

7.24 SADXRR (BUF, XRJ) sets up type I R-list to do a short add of two registers with the result going to one of the registers (SRi = Ri + XRJ). Fields used are RJ, RK, RI, OC given by R, XRJ, R and SADD respectively. R-list storage address is determined from BUF. Calls on MACRO's TYPEI, OUTBUF and subroutines TOPB, ENDB, VARB1, VARB2.

7.25 XMIT (BUF) sets up type I R-list to transmit from an input register to an output register. Fields used are RJ and RI given by R and R+1 respectively. R-list storage address determined from BUF. Calls on the macro OUTBUF and subroutines TOPB, ENDB, VARB1, VARB2.

7.26 ADDXRR (BUF, XRJ) generates R-list to do an integer add of two X registers (R and XRJ) and put the result back into R. R-list fields are OC, RJ, RK, RI found in IADD, XRJ, B7, B7 respectively. R-list is stored in an address determined from BUF. The macro uses TYPEI, OUTBUF macros and subroutines TOPB, ENDB, VARB1, and VARB2.

7.27 SADZRR (BUF,XRJ) generates R-list to set an X register to the value in a B register. XRJ is the R number for the B register and R is the X register number. Fields needed are RJ, RI given by XRF and R. R-list storage determined from BUF. Calls on Macro OUTBUF and subroutines TOPB, ENDB, VARB1 and VARB2.

7.28 CONLS (BUF, XCOU, XREG) generated R-list to do a constant left shift of XREG the number of places specified by register XCOU. Operation code given by KLS and BUF determines R-list storage. Uses the macro OUTBUF and subroutines TOPB, ENDB, VARB1, VARB2.

8.0 DOPRE

8.1 Principles

Sample test cases using straight forward compilation techniques indicate that 50% or more of the running time of FORTRAN benchmark programs is spent inside innermost DO loops, although they occupy under 10% of program space. Therefore, reduction of the time spent in loops particularly inner loops, is of the first importance.

However, since the time required for a short loop is greatly reduced by retaining it in the stack, it is important to reduce the space required within loops as well. To avoid special cases, DOPRE concerns itself with space reduction only, assuming that time reduction is usually a by-product.

8.2 Definition of Well-behaved Loops

Loops containing extended ranges, input-output statements, CALLS's, FUNCTION references, or calls or arithmetic statement functions which contain CALL's or FUNCTION references, or implicit subroutine calls, are not considered well-behaved and are not optimized for the following reasons:

a. The existence of an exit and return to the loop makes retaining results in registers throughout the loop impossible.

b. The existence of an exit and return to the loop makes retaining the entire loop in the stack impossible.

c. The computation outside the loop, which may be a considerable proportion of the compute-time involved, does not benefit from the loop optimization.

d. The existence of most of the above situations creates implicit definition points which complicates the analysis.

e. The existence of an extended range requires the use of program-wide temporaries.

Currently, a loop must meet three other requirements in order to be optimizable:

f. It must fit in memory.

g. It must be an innermost loop. Outer loop optimization complicates the analysis considerably, but saves less execution time and is less likely to produce in-stack operation than innermost optimization.

h. The increment must not be the control variable. This situation is rare but when it exists it makes calculation of the increments of local index functions outside the loop impossible.

8.3 Objectives of DOPRE

DOPRE generates code

a. To compute the reference all standard index functions.

b. To count DO loops.

c. To materialize control variables where necessary.

d. To provide optimization features if requested.

PRE calls DOPRE when it encounters a DO begin or DO end macro, or a standard index function pseudo-macro.

If no optimization is requested, DOPRE expands each macro and returns to PRE. Standard index functions are reordered to minimize the computations required, and computations involving adjustable dimensions are done at the program top.

The code generated for DO I=B, C, D is:

B Constant	B Variable
Set R to B	Load R with B
Store R in I	Transmit R to R
	Store R in I

The code generated for DO end is:

D Constant		D Variable
Load R1 from I		Load R1 from I
R1 + D to R3	or	Load R2 from D
Store R3 in I		R1 + R2 to R3

```
                                    Store R3 in I

C Constant                          C Variable

R3 - (C+1) to R4                    Load R4 from C
                        or
NG   R4 L                           R4-R3 to R5

                                    PL R5 L
```

If standard optimization is requested, DOPRE allows PRE to form an R-list buffer containing the above macros and pseudo-macros and the expanded R-list items for an entire well behaved loop and the first sequence outside the loop. It analyzes this information in detail, modifies it, and returns it in final form to PRE.

The analysis provides the following broad features:

1. Standard index functions which do not change in the loop (global) are computed once outside the loop.

2. Standard index functions which change in the loop only when the control variable changes (local) are initialized outside the loop and incremented within the loop.

3. Variable increments of local index functions are computed outside the loop.

4. Standard index functions containing variables changed in the loop (redefined) are computed at the point of reference.

5. The adjustable parts of all standard index functions, local, global or redefined, in a loop or out, are computed at subprogram top.

6. B registers are initialized at loop top with constants, addresses, index functions, and variable increments chosen to reduce space in the loop.

7. Instructions which reference the sums and differences of B registers are used to reduce space further.

The quantities loaded in B registers are selected to minimize the space required by the loop, in accordance with Section 1 above.

The following quantities are initially candidates for B registers:

1. Index functions none of whose variables are redefined in the loop.

2. Variable increments required for modifying index functions containing the current control variable.

3. Addresses of variables in load and store instructions.

4. Constants in Type II Set instructions.

5. Loop limits for use in materializing the control variable and/or counting the loop.

8.4 How Index Functions Are Computed

The general form of a standard index function is:

```
DIMENSION A(L,M,N)

      ...A (aI+d, bJ+e, cK+f)
```

(L,M and N are each either constant or adjustable; a, b, and c are positive integer constants; d, e, and f are signed integer constants. I, J, and K are variables.)

The above reference necessitates computing the address

A+(aI+d-1)+(bJ+e-1) *L+ (cK+f-1) * L * M

DOPRE reduces the subscript to a canonical form containing the following items:

1. The array name, e.g., A

2. The constant addend CA, e.g., d-1 in the example (assuming L and M are adjustable).

3. From zero to 5 additive terms, each containing from 1 to 3 variables.

Assuming all dimensions are adjustable and I, J, and K are different, the terms in the example would be aI, bJL, (e-1)L, cKLM, and (f-1) LM.

Terms are sorted so that

1. Index functions which look different, but require the same computations, are recognized, e.g., A(2*I+1) and D(I+1) where D is double length.

2. Non-adjustable variables appearing twice require at most one multiplication, e.g., A(2*I,1) yields A+(L+2)*I+CA, not A+2*I+L*I+CA; (CA=-L-1).

DOPRE computes the constant part of the index function. It generates code at subprogram initialization to compute the adjustable part of the index function.

Multiplication by some integer constants is simulated using shifts and adds. The special cases identified are:

1. negative constant

   An instruction B0-R to R is issued and positive multiplication proceeds.

2. constant = 1

   No generation.

3. constant = 3

   R1+R1 to R2. R1+R2 to R3

4. constant = 6

   R1+R1 to R2. Then use code for 3.

5. constant = 2n

   Shift transmit and left shift n.

6. constant = 2**n+2**m (0...010...010...0)

   Shift transmit, left shift n-m, add, shift transmit, left shift m.

7. constant =2**n-2**m (0...01...10...0)

Subtract instead of add.

8.5 Program Flow

When fast compilation is requested PRODB, PROIXFN, and PRODE merely generate R-list in line. When standard compilation is requested, most of DOPRE's work is done in 10 sections of code within PRODE. Section I scans the R-list for the entire loop and forms a table of candidates - the number of bits saved inside the loop by assigning it to a B register. Section III assigns the most promising candidate to the 1st B register, recomputes the values if they are altered, assigns the next B register, etc., until there are no more candidates or no more B registers. Section IV determines the most space-saving way of materializing the induction variable (if necessary) and counting the loop, and may alter B7 accordingly. Section V records the final assignments. Section VI, VII, and VIII generate code in another table to: load addresses and constants; compute index functions and increments; and materialize and count, respectively. Section IX re-scans the R-list for the loop, changing references to refer to B registers. Section X merges the generated code with the original R-list. Control returns to PRE.

8.6 Evaluating Candidates

As Section I forms the potential candidate list, it notes in which sequences each candidate was referenced, notes uses of B registers already present in the R-list, and groups index function and address candidates.

Where Z is a candidate

SEQ (Z) = a bit pattern containing a bit in the nth position if Z is referenced in the nth sequence.

L= the number of bits in SEQ (Z)

DOPRE cannot tell what the final code will be; it assumes that quantities in X registers are loaded only once per sequence in which used.

Section II evaluates the candidates. The "value" of a candidate is the difference between the number of bits required for instructions to load and/or reference it

within the loop if it is not assigned a B register, and the number required if it is assigned a B register.

A. Addresses in Load A and Store A Instructions

All addresses become candidates. After an address is assigned a B, the differences between it and each other required address become candidates also. That is, if Y and Y + 1 are referenced, the code will be one of the following:

	Number of Registers Assigned				
	0	1	1	2	2
set B1 to	-	Y	Y+1	Y	Y+1
set B2 to	-	-	-	1	1
reference Y	ref Y	ref B1	ref Y	ref B1	ref B1-B2
reference Y+1	ref Y+1	ref Y+1	ref B1	ref B1+B2	ref B1

Differences are used because of the likelihood that constant differences (and possibly symbolic differences) may occur several times. In the above example, the constant 1 may be referenced directly and may occur also as the difference of A and A+1, B and B+1, C+1 and C+2, etc.

As can be seen from the above chart, the value of an address is 15L, and the value of a difference is also 15L. (Symbolic differences are not formed when either symbol is a formal parameter.)

b. Constants in Set X Instructions, as increments, as differences, as loop limits. All constants are candidates (negative constants are filed in positive form). Whenever a constant is assigned a B register, Section III scans the other B registers assignments for other constants and determines which, if any, useful constants may be formed as the sum or difference of assigned constants. Suppose the constants 1, 2, 3, 4, 5, 6, 10, and 50 are required. Suppose also that B register assignments of constants are as follows:

```
B1   4

B3   1

B6   50
```

References to 5 are generated as B1+B3. References to 3 are generated as B1-B3. References to 2 are generated as B3+B3. Although the assignment of, say, 3, 2, and 50 would have been preferable (because 1, 4, 5, and 6 can be formed), no attempt is made to assign the optimum constants.

The value of a constant is 15L since "SX1 B1" is 15 bits shorter than e.g., "SX1 4."

c. Variable Increments required for modifying local index functions. With the index function in B2, the code required is:

```
        No   B                    1   B

Instruction:

     SA1  INC                     -

(SB2 X1+B2 or SX6 X1+B2)    (SB2 B2+B1 or SX6 B2+B1)

Cost:     30                         0
Value:    30
```

d. Index Function

All index functions with identical variable parts comprise a "group", e.g., X(I+5), and Y(I), are in the same group; except that each array which is a formal parameter has its own group.

Index functions are grouped because costs may be reduced by using the same memory cell or B register for more than one of them; e.g., if X(I) is in B1, Y(I) may be loaded as follows:

```
SA1  Y-X+B1
```

and this may minimize space when there aren't enough B registers to go around.

Formal parameter arrays have their own groups because references like Y-X+B1 would otherwise have to be computed at run time.

The value (in bits, within the loop) of an index function depends on how it is computed and referenced.

There are six ways to compute/reference X(I) where Y(I) is also required:

	Outside Loop	Reference
In Memory	X+I to LTEMP	SA1 LTEMP ref X1
In B	X+1 to B1	ref B1
Share Memory	Y+I to LTEMP	SA1 LTEMP ref X-Y+X1
Share B	Y+I to LTEMP	
	X-Y to B2	SA1 to LTEMP ref X1+B2
Different Memory	Y+I to LTEMP	
	X-Y to B2	SA1 to LTEMP ref X1+B2
Difference B	Y+I to B1	
	X-Y to B2	ref B1+B2

The initial value of a group of index functions is the cost of having no B register for the group minus the cost of having one B register for the group. This is the same as the cost of loading the index function in each sequence in which any member of the group is referenced, or 30* number of bits in (SEQ v...v SEQn), plus incrementation cost (for local index functions) of 45.

The differences between the candidate (the most-loaded group member) and each other member of the group also become candidates with values of 15L.

8.7 Assignment of B registers

Section II performs the initial evaluation of candidates. Section III assigns the most valuable candidate to B1, re-evaluates as necessary and assigns the next highest candidate to B2, etc., until there are no more B registers or no more candidates. The first time an address is assigned the differences with other addresses are filed. Each time a constant is assigned, combinations of it and previously assigned constants are generated.

8.8 Loop Control Code and Materialization

Subroutine MATC following Section I, and Sections IV and VIII select and generate the code to test at the bottom of the loop and to materialize the control variable if required.

MATC files the loop limits as candidates if materialization is required. Section IV chooses the best code for materialization and testing based on the assignments to B registers made in Section III. Section IV may alter assignments to produce better code.

Section VIII generates the materialization and testing code. The code produced is one of the following:

NORMAL LEVEL OF OPTIMIZATION MODE LOOP COUNTING

(where the general form of the DO is DO SN I = B,C,D)

1. Count the loop in memory

TOP		END
(C-B) D	LTEMP	SA1 LTEMP SX X1-1 SA6 LTEMP PL X6,TOP

2. Count the index function to zero

TOP	END
(-mc+mb) Bi	(increment Bi)
	GE B0, Bi, LABEL

3. Count the index function up to mc (multiplier times upper limit)

TOP	END
(index function) Bi	(increment Bi)
mc B7	GE B7, Bi, LABEL

4. Count in B7 where D is a B register

TOP	END
(B-C-1) LTEMP	SB7 B7+BD
	GE B0, B7, LABEL

5. Count in memory where D is in a B register

TOP	END
(B-C-1) LTEMP	SA1 LTEMP SX6 X1+BD SA6 LTEMP NG X6, LABEL

6. Count in B7 where there is a 1 in a B register

TOP	END
(B-C) D B7	SD7 B7+B1 GE B0, B7, LABEL

7. Count in memory where there is a 1 in a B register

TOP	END
(B-C) D -1 LTEMP	SA1 LTEMP SX6 X1+B1 SA6 LTEMP NG X6, LABEL

8. Count the loop B7

TOP	END
(C-B)	SB7 B7 -1

```
D    B7          GE B7, B0, LABEL
```

9. Test the control variable I with C loaded

```
TOP              END

                 (load C)
                 IX0 Xc-Xi
                 PL  X0,LABEL
```

10. Test the control variable I with C+1 loaded

```
TOP              END

                 (load C+1)
                 IX0 Xi=Xc-1
                 NG  X0, LABEL
```

11. Test the control variable I where count (C) is materialized

```
TOP              END

                 SX0 Xi-C-1
                 PL  X0, LABEL
```

12. Test the control variable I where the address of count (C) is in a B register

```
TOP              END

                 SAi  Bc
                 IX0  Xc-Xi
                 PL  X0, LABEL
```

8.9 Generation

Section V marks the final B register assignments in the A table. Section VI generates code at loop top to load addresses, constants, and symbolic differences.

Section VII generates code at loop top to initialize index function and variable increments to local index functions, and code at loop end to increment local index functions. Section IX re-scans the original R-list, changing the references to reflect the B register assignments and generating adjustable computations in redefined index functions. Section X merges the loop top

and loop end code with the R-list for the body of the loop, and generates code in-line to compute and reference redefined index functions.

The program then exits to PRE.

PROSEQ

1.0 General Information

PROSEQ is the controller for the processing of a sequence or group of sequences. This processing begins with a sequence as collected by PRE and results in the final production of a series of COMPASS line images on the COMPS file. A number of these routines are called to perform the actual processing. Under OPT=2, PROSEQ controls the selection and assignment of quantities to registers over the body of a loop. Necessary preload and post store code is produced within PROSEQ.

2.0 Entry Points

2.1 USEDT

Data area containing the character string USE DATA.

2.2 NORLIST

Cell containing a count of the number of R-list entries in the current sequence.

2.3 FWADESC

Cell containing the address of the start of the descriptors for the sequence as copied by PROSEQ.

2.4 FAILCT

Count of the number of times failure occurred when processing a sequence. This is only incremented by conditional code if the symbol FAILNUM is set non-zero.

2.5 FWARLIS

Cell holding the first word address of the R-list1 entries in the copied sequence.

2.6 PROSEQ

Primary entry for all processing by PROSEQ.

3.0 Messages and Diagnostics

MEMORY OVERFLOW IN PROSEQ

MORE CORE WOULD HAVE RESULTED IN BETTER OPTIMIZATION

4.0 Environment

BASE — Current base R number for this sequence. Used only for OPT=0 processing.

FWAF — Holds the first word address of the F table. Used only for OPT=0 processing.

LWAWORK — Holds the last word address of working storage.

ENDSEQ — Holds the address of the current end of sequence marker in uncopied R-list.

ENDLIST — Holds the address of the last end of statement marker within the optimum size distance from FWASEQ when a sequence fail occurs.

VARFLAG — Nonzero if a VARDIM code sequence is being processed.

LASTR — Cell in OPTB used to contain the list R-list1 entry issued. It is used to initiate register clearing by OPT.

LASLBL — Holds the last label encountered and is referenced in OPT to determine if stack timing can be used.

SQUEEZE — Routine which removes redundant calculations from a sequence by marking them as killed and adjusts the R numbers appropriately.

PURGE — Routine that physically deletes entries in the R-list sequence that were killed by SQUEEZE.

BUILDDT — Routine to construct the dependency tree used by OPT in issuing the code in the sequence.

FWAWORK — Holds the first word address of working storage.

RLIST Holds first word address of the working area used in copying the sequence. Typically the same as FWAWORK.

PUNT Called when insufficient storage is available to process the sequence.

POST Routine to produce a COMPASS line image, given an R-list entry.

SQZVARD Routine to perform squeezing on a VARDIM sequence.

OPTA Routine to issue a sequence of code in optimum functional unit usage order.

FWASEQ Holds the first word address of the uncopied sequence.

JAMMER Last report sequence processor when all recovery mechanisms in PROSEQ are unable to code a sequence into COMPASS. A call to JAMMER will always code the sequence.

NFPUNT This external is called when the lack of memory force PROSEQ into a recovery mechanism. The message MORE MEMORY WOULD HAVE RESULTED IN BETTER OPTIMIZATION is produced.

COVD. Current use block ordinal.

VARDIM Block ordinal address for the VARDIM block.

OVERS Holds the number of leftover parcels from the previous sequence.

POSTIFQ Start of an area used in passing R-list entries to POST for conversion to COMPASS.

OPNPOST Entry in POST to initialize sequence processing.

CLSPOST Entry in POST to dump the accumulated COMPS images to the COMPS file once a sequence is successfully coded.

STMTS Number of statements in the current sequence.

TREEING Holds the length of the dependency tree.

FWATREE Holds the first word address of the dependency tree.

S.FWA Holds the start of the sequence +3 on entry.

OPTLVL Hold the optimization level selected on the FTN card.

ADSUBTT Flag word indicating ADDSUB macro placed on COMPS already.

SUBREF SUB macro reference flag.

SQZFLG Holds a value indicating sequence or sequences over which register allocation (OPT=2) is taking place.

RLOCK Address of the R-list1 word for the start of the locked register prologue code under OPT=2.

RCHANGE

MEM If non-zero, indicates a sequence processing fail for lack of memory.

BIND If BIND is non-zero, no calls to POST will be made from OPT (used only by OPT=2).

FAIL Control is transferred to this external when a sequence processing fail occurs on a sequence containing locked registers.

RI(J) Contains the master R number used by REPLACE when an R-list entry is killed.

DESCR Descriptor table in READRL.

XOSCR Base of the scratch register table in OPTB.

NLOKR Holds the number of locked registers for this sequence (OPT=2 only).

REPLACE Routine to replace all references to a killed R number by a master R number.

WRWDS Writes words to a specified file.

5.0 Processing

PROSEQ is called by PRE to process an accumulated sequence. On entry, the first word address of the uncopied sequence is established (FWASEQ) and for OPT=0 mode the F table is marked non-existent so that a new one will be constructed for the next statement. An initial test is made to determine if the sequence will fit in the available working storage. If this test indicates insufficient storage, the sequence is scanned forward to the first end of statement marker and the fail logic is is employed on that statement. Normally, sufficient memory will be present. In this case, the sequence is copied. Initially, each R-list instruction occupies three consecutive words (RLIST1, R-list2 and descriptor). After the copy, all the R-list1's are followed by all the R-list2's which are followed by the descriptors. The original sequence remains untouched by the copy operation. During this copy, the number of R-list instructions (NORLIST) is established. If only one instruction is present, special processing is employed. In particular, if it is a label, POST is called to issue. For a register define standing alone, exit is made to the path for a successfully processed sequence. (A stand alone register define can occur in this case:

```
      IF(EOF(1)) 10,10

10    CALL XYZ
```

and needs no further processing other than a success exit.). For any other stand alone R-list instruction, a tree is constructed by PROSEQ and OPT is called. Generally, however, the sequence will hold more than one R-list instruction. First, let's examine OPT=1 successful sequence processing since this is the most common path.

5.1 OPT=1, Success Path

a. Call SQUEEZE to mark redundant operations in the copied sequence.

b. Call PURGE to remove redundancies marked by SQUEEZE.

c. Call BUILDDT to construct the dependency tree used by OPTB.

d. Call OPTB to issue the sequence in the most optimum fashion.

e. On a successful return, a test is made to determine if the current sequence passed to PROSEQ has been completely issued (ENDLIST +3 = ENDESEQ). If not, control is passed block to the copy process after setting FWASEQ to ENDLIST +3.

f. If the current sequence is fully issued, a check is made to determine if the group of sequences passed to PROSEQ have been completely issued (ENDSEQ+6 > R-list). If not, control reverts to the copy phase.

g. Upon completion, the sequence area is returned to working storage and PROSEQ is exited.

5.2 OPT=0 Processing

a. After copying the sequence, PROSEQ is ready to call OPTB to issue the instructions in their physically occurring order. However, if PRE determined that the present statement was more than forty RLIST instructions long, an immediate fast mode fail is assumed.

b. If OPTB returns indicating failure the sequence (consisting of one statement) will be forced down the OPT=1 path.

5.3 Failure Processing (OPT=0,1)

A failure occurs when

a. There is insufficient memory to copy a sequence or build the dependency tree. This is a memory failure and causes a MORE MEMORY WOULD HAVE RESULTED IN BETTER OPTIMIZATION MESSAGE to be issued.

b. OPTB is unable to issue the sequence because of insufficient registers.

Recovery processing from each of these failures in the same. On successive fails with the same sequence, different things are tried in order to achieve issue.

5.3.1 First Fail

a. ENDPAR is set to FWASEQ +3* optimum number of R-list entries (presenty 40 entries). This demarcates an upper limit on a portion of the sequence to be examined.

b. If the sequence contains less than the optimum number, ENDPAR is set to ENDSEQ.

c. Then the entries from FWASEQ to ENDPAR are scanned and the location of the first and last end of statement in this region are noted (FIRSTMT holds the location of the first and ENDLIST the location of the last).

d. If upon scanning the region FIRSTMT=0 (no end-of-statement in the area), ENDPAR is bumped one entry and another check for end-of-statement is made. This continues until one is located or ENDPAR = ENDSEQ. For no end-of-statment, the NOFIRST flag is set and an attempt is made to process the sequence without squeezing.

e. If FIRSTMT≠0 and ENDLIST=0, ENDLIST is set to FIRSTMT and the first statement alone is copied for processing.

f. If FIRSTMT≠0, ENDLIST≠0, ENDLIST is used as the copy boundary, unless it equals ENDSEQ-3, in which case FIRSTMT is the boundary.

g. After locating a copy boundary, a failure copy routine is used to make the copy of the sub sequence.

h. Once the sequence is copied SWITCH1 controls subsequent processing:

Value	Action
0	call SQUEEZE,PURGE,BUILDDT,OPTB
1	call BUILDDT,OPTB
-	call JAMMER

5.3.2 Second Fail

On a second fail when FIRSTMT ≠ ENDLIST, an attempt is made to issue just the R-list up to FIRSTMT without squeezing or purging.

5.3.2 Third Fail

At this point, OPTB has failed in issuing a single statement. No further sequence splitting may be done so JAMMER is called. JAMMER will issue the statement generating memory temporaries if needed.

5.3.3 Success after Fail

When a fail sequence is finally issued, all failure indications are reset and normal sequence processing resumes on any protions of the sequence that were split away by PROSEQ.

5.4 OPT =2 Register Allocation Processing

If this group of sequences is a prologue and loop body over which an attempt is being made to lock registers (SQZFLG ≠0), special considerations are taken. First, the prologue sequence must be processed. This consists of scanning the prologue (once for each locked register) to locate any stores into symbols whose IN field and CA field matches that of one of the register candidates presently assigned. If such a store is formed, the preload generated for the locked quantity is changed to a transmit from the register used in the store to the locked register. If the sequence being examined is in the body of the loop, then loads and stores of locked variables must be eliminated when a load is located whose IH-CA fileds are the same as one of the locked registers. The kill bit is set in the descriptor and REPLACE is called to replace all references to the killed R number into references to the locked R number. Finally, SSU (Scan For Shift Usage) is called. Since OPTB forces the destination register of a constant shift to be the same as the source register, the load just killed must not be the object of a constant shift. If it is used by a shift, the load that was killed must be restored as a transmit from the master register to the register the load was to use. Then, the shift is modified to use the result of the transmit as its operand.

When a store into a locked quantity is found, it is killed and the result of the operation which created the R number to be stored is routed to the locked register. Finally, CKS (Conditional Kill Stores) is called. CKS examines the sequence to determine if multiple stores occurred on the R number just killed. Should this be the

case, the store that was killed must be modified to a transmit from the store register to the locked register.

Once this processing has occurred, the sequence follows normal OPT=1 paths through SQUEEZE, PURGE, BUILDDT and OPTB. If a failure occurs on a sequence with locked registers on, exit is made to recovery logic in PRE.

5.5 VARDIM Processing

If VARFLAG is non-zero the sequence passed to PROSEQ is a sequence of variable dimension code accumulated by the DO processor in pass two. In this case, SQZVARD is called to remove redundancies from the VARDIM sequence, the sequence is purged and normal flow entered at dependency tree construction. Special logic exists for processing failures on a VARDIM sequence. In particular, since there are no end of statement markers, the store instructions are used as locations to divide the sequence. In locating a point to divide the sequence, the midpoint is located and an alternating check is made above and below this midpoint until a store is found. Once a subsequence is isolated, processing uses the normal failure copy mechanism.

6.0 Table Formats

6.1 Working Storage on Entry

```
RA
     Compiler

                                S.FWA
     Sequences
                                FWAWORK
     Working Storage
                                LWAWORK
     VARDIM & APLISTS
     Symbol Table
FL
```

6.2 Working Storage After Copy

```
          Compiler

                                S.FWA
          Sequences
```

```
                                    FWARLIS
RLIST1
RLIST2
                                    FWADESC
Descriptors
                                    FWAWORK
Working Storage
                                    LWAWORK
VARDIM & APLISTS
Symbol Table
```

6.3 Working Storage after BUILDDT

```
Compiler

                                    S.FWA
Sequences
                                    FWARLIS
RLIST1
RLIST2
                                    FWADESC
Descriptors
                                    FWATREE
TREE
                                    FWAWORK
Working Storage
                                    LWAWORK
VARDIM & APLISTS
Symbol Table
```

JAMMER

1.0 General Information

On entry, JAMMER sets TREEINIT to LWAWORK-1 and sets LWAWORK to LWAWORK-1 and sets TREEJAM non-zero to indicate JAM mode. BUILDDT is called to build the dependency tree. Since the TREEJAM flag is set, he will only construct a three part tree and exit. The three portions are shown in section 6.1. If the tree could not be built due to a lack of memory, a fatal to compilation exit is made. The size of the tree is computed and stored in TREESIZE. If there is insufficient storage to sort the tree (52 additicnal words), abort. Next, we sort the tree on the following keys: priority (ascending), R-list (descending), PREDT (ascending), and RPRED (descending). Next, all dangling code (zero priority items) are removed from the tree and the usage counts on their predecessors reduced.

The next section of JAM constructs the information table describing entries in the dependency tree. The table is built from FWADT3 towards RA. Set NOTISSUD to the number of R-list entries remaining after the elimination of dangling code. After some register initialization, a zero word is placed at FWADT3-1 and the tree is examined starting at TREEINIT.

a) If an entry has no predecessor (RPRED=0), go to c.

b) If it is a register predecessor, increment the number of predecessors count (NRPRED) by one.

c) Pick up the next tree entry.

d) Clear the priority and garbage areas in the previous entry and store it back.

e) If the previous entry and the current entry have the same successor (R-list) field, go to a.

f) Make an entry in the tree information table with the number of predecessors found, and the USED field zero. (See 6.2 for format.)

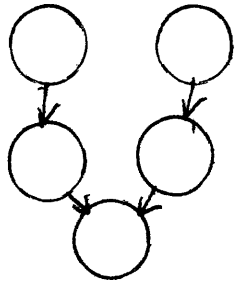
In the associated descriptor (pointed to by the R-list field), we set FINAL=1, USABL=1 if no logical predecessors were found, else 0, INFO=address of the tree information word for this instruction. (See 6.3), and ZEROP=1 if there were no register predecessors. Finally, the associated R-list tree address is placed in the descriptor.

g) Load the next entry, reset number of predecessors to zero.

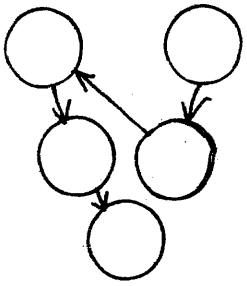
h) If this is the end of the tree, go to the next phase of JAMMER.

i) If the information table has run out of storage, then abort. Otherwise, go to a.

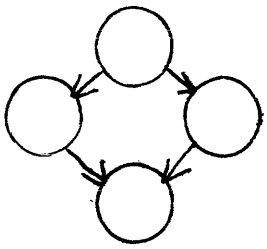
After creating the information table, the tree restructuring phase of JAMMER (STRUCT) is entered. In this portion of the processing, we scan forward through the instructions computing the cost in registers at each multientry node. When the full tree has been scanned, the next phase (TRYISSUE) is entered with SAVETLOC pointing to the multinode entry whose predecessors we will try to issue. TRYISSUE creates a set of forward link fields working back from the multinode entry until an entry with no predecessors is found. This is the top of the issue tree. Succeeding tree entries for the multinode entry (at most 2 more in the case of a store) are linked in by iterating through TRYISSUE until the number of predecessors of the entry has been reduced to zero. To prevent destroying previously formed link fields, all branches of the multinode entry are linked top to bottom and right to left. Thus the tree



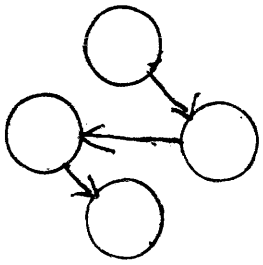
is linked as



This removes the possibility of circular linking in the case



which links as



When TRYISSUE has forward linked all the entries, we scan down the linked list calling ISSUE to perform logical issues of the instructions, maintain register availability information and construct an actual issue tree. If we issue the entire subtree, we reiterate the STRUCT code to choose another subtree, TRYISSUE, to form the issue chain and then issue as above.

If no issuable instructions were found, the cost was greater than the available registers). UNISSUE is called to restore anything on the issue chain and CREATEOT is called to generate an optimizing temporary. Then, we try to structure the tree again.

If all instructions are on the issue chain, we call MOVETREE to form the actual tree and call OPTA to issue the sequence.

2.0 Entry Points

2.1 JAMMER

This is the primary entry point and is called to process a sequence that cannot be issued by other means.

2.2 RADIXIT

This is the entry point for the radix sort routine. The calling sequence is

```
SA1  APLIST
RJ   RADIXIT
```

where the parameter list is

```
APLIST    BSS  0
          VFD  60/TABLE
          VFD  60/SIZE
```

```
        VFD  60/FIELDS
        VFD  60/WORK
```

where

TABLE	=	fwa of table to be sorted
SIZE	=	number of entries in the table
WORK	=	fwa of the work area which should be at least n+2 8 words when there are n bits in the key
FIELDS	=	array describing fields to be sorted. The list is terminated by a zero word. The form of an entry is

```
        VFD    1/AD,23/0,18/LEFT,18/RIGHT
```

where

AD	=	0 for ascending or 1 for descending
LEFT	=	leftmost bit number of the field
RIGHT	=	rightmost bit number of the field

3.0 Diagnostics

No diagnostics are issued by JAMMER

4.0 Environment

LWAWORK

Holds the lwa of working storage.

NORLIST

Holds the number of entries in the current RLIST sequence.

FWAWORK

Holds the fwa of working storage.

FWADT1

fwa of the first section of the dependency tree.

FWADT2

fwa of the second section of the dependency tree.

FWADT3

fwa of the third section of the dependency tree.

TREELNG

Length of the tree.

FWARLIS

fwa of the RLIST sequence.

5.0 Structure

STRUCT - Restructure the Tree

a) Set the optimizing temporary table to start after FWADT3.

b) Mark zero as an unusable OT. Set new LWAWORK.

c) Clear the area of JAMMER from ZERO1ST to ZEROLAST.

d) Set the number of available load registers (NLOADREG) to five.

e) If there are no register defines preceding the sequence, go to n.

f) Set SAVETLOC to point to the tree entry for the register define and call ISSUE to place it on the issue chain.

g) Pick up the next R-list instruction. If it is a register define, go to f.

h) Build the subtree from the linked list (constructed by ISSUE) by calling MOVETREE.

i) Call OPT to issue the register defines.

j) Reset crucial registers and call KILLEM to remove the entries just issued.

k) Set LASTR so that OPT will not clear registers. Then set the exponent of all the issued register defines to minus zero.

l) Clear ISSUE1ST list, ISSUED. If unissued instructions remain, go to n.

m) Reset LWAWORK to TREEINIT. Update the largest number of OT.'s ever used and exit JAMMER.

n) Set REGCOSTP = REGISTRS, REGCOST = MAXREGS+1, USESMIN = USESMINP = 2000B, SAVETLOC = MULTINOD = 0, number of load registers available (NLOADRGP) = NLOADREG, initial tree pointer to TREEINIT.

o) Scan down the tree until a final tree entry is found. If the entry is alive, unissued and usable, go to q.

p) If we have not reached the end of the tree, go to o. Otherwise, go to cc.

q) If the entry has only logical predecessors, go to p.

r) Extract the number of predecessors from the information word. If the entry has more than one predecessor, go to u.

s) If the predecessor is not usable, go to p. If it is not marked (on the issue chain), go to p. If all uses of the predecessor have not been satisfied, go to p.

t) Add the instruction to the issue chain by calling ISSUE. Go to n.

u) Set the MULTINOD flag non-zero (indicates multiple predecessors).

v) Call REGCOSTS. If REGCOSTS indicates failure, go to bb.

w) Call REGCOSTS with the current tree entry + one (towards FL). If a failure return is made, go to bb.

x) If the number of predecessors is not three, go to z.

y) Call REGCOSTS with the tree entry two above the current tree entry. If a fail return is made, go to bb.

z) If REGCOSTP + 1 is greater than REGCOST, go to bb. If REGCOSTP + 1 is less than REGCOST, go to aa. If USESMINP is greater than or equal to USESMIN, go to bb.

aa) Set SAVETLOC to the current tree entry, REGCOST to REGCOSTP=1, USESMIN = USESMINP.

bb) Reset REGCOSTP to REGISTRS, USESMINP to 2000B, number of available load registers to NLOADREG (=5). Go to p.

cc) If an issuable instruction was found, go to hh.

dd) If any unissued multibranch nodes have been encountered, go to rr.

ee) If no instructions remain to be issued, go to ss. Initialize tree pointer to FWADT3-1.

ff) Scan up the tree until a final tree entry (FINAL=1) is found. If the entry is not usable or already marked or dead, continue scanning.

gg) Save the address of the entry located in SAVETLOC. Make REGCOST be REGISTRS+1.

hh) If REGCOST is greater than MAXREGS (=8), go to rr.

ii) Clear the LINKF field in the tree information word for this entry.

jj) If the number of predecessors for the final tree entry is zero, go to pp.

kk) Decrease the number of predecessors count by one. Advance to the next (towards FL) tree entry.

ll) If the predecessor of the current entry is marked, go to jj.

mm) Set the mark bit in the current entry. Change the forward link in the tree information word of the predecessor to point to the current tree entry.

nn) The current tree entry pointer is advanced to point at the predecessor tree entry.

oo) If the number of predecessors of the initial final tree entry is non-zero, go to jj.

pp) Set SAVETLOC to the current tree entry on exit from the above loop. Set NEXTTLOC to the address in the forward link field for the information word whose tree entry is denoted by SAVELOC. Call ISSUE to link the final tree entry to the list of issued tree entries.

qq) If NEXTTLOC (next tree location) is non-zero, go to pp. If instructions remain to be issued, go to n, otherwise go to tt.

rr) If any instructions are on the issue chain, go to tt.

ss) Call UNISSUE to reset register allocation information for issued instructions not posted by OPT. Call CREATEOT to release a register. Go to n.

tt) Fetch the number of instructions issued (ISSUED).

uu) Set NISSUED to the value. Call MOVETREE to form the subtree for this issue chain. If there is not enough room to form the subtree, go to zz.

vv) Set NEXTTLOC to the address of the final tree entry of the next tree entry in the issued list.

ww) Save OPT table areas and then call OPT. Restore registers destroyed by OPT. If OPT succeeded in issuing the subtree, go to yy. Restore OPT's table areas.

xx) Divide the number of instruction on the issue chain that we will try by four. If that number is greater than one, go to uu. If the number is zero, go to ss to release a register. If the single instruction is the one last placed in an OT., go to ss, otherwise go to uu.

yy) Call KILLEM to kill the tree entries for the instructions just issued. Clear last OT. used cell. If the next tree location is zero, go to zz. Set

ISSUED = ISSUED - NISSUED. Set the first field in ISSUE1ST to point to the next tree location. Set the backward link in the information word for the new head of list to zero. Go to 1.

zz) Reduce the number we will try to construct a subtree for by half. If the remaining number is zero, abort. Otherwise go to uu.

ISSUE

ISSUE will link the R-list instruction whose dependency tree address is pointed to by the contents of SAVETLOC to the list of instructions issued. Current register usage information is updated to reflect the issuing. The list of issued instructions is maintained by a cell ISSUE1ST and the LINKF, LINKB fields in the tree information words. ISSUE1ST contains pointers to the first and last entry in the issued list. Intermediate entries on the issued chain are connected via the forward (LINKF) and backward (LINKB) fields in the information word.

a) On entry, the mark bit is set into the tree entry to be issued. The number of predecessors is extracted from the information word.

b) If the number of predecessors is zero, go to i.

c) Reduce the number of predecessors by one.

d) Add one to the used count field in the information word of the predecessor to the entry we are issuing.

e) Fetch the next tree entry.

f) If the uses on the current predecessor is still greater than the number of times it has been used, go to b.

g) Release the register containing the predecessor by calling RESETREG with a -1.

h) Go to b.

i) Increment the ISSUED count by one and decrement the NOTISSUD count by one.

j) If the uses count in the descriptor of the instruction to be issued is non-zero, call RESETREG with a 0 to allocate a register.

k) Now link this instruction to the list of issued instructions. If instructions have been issued, go to m.

l) Set the first issued entry address in ISSUE1ST to the address of the entry we are issuing. Set last issued entry address to zero.

m) If there is no last issued field, go to o.

n) Modify the information word of the last issued tree entry, inserting the address of the tree entry we are issuing in the forward link field.

o) Modify the information word of the tree entry we are issuing by inserting the last issued field from ISSUE1ST into the LINKB field.

p) Modify ISSUE1ST so that the entry we just issued is pointed to as the last issued entry.

q) Call MARKEM to mark the instruction we issued and check for its use as a logical predecessor.

r) Exit from ISSUE.

RESETREG

RESETREG is called to allocate or deallocate a register. If B3 is negative, we are to deallocate, otherwise allocate. On entry, X7 holds the descriptor address of the instruction for which the register is to be allocated.

a) If it is a load instruction, go to f.

b) If the information word entry indicates there is no succeeding issued instruction, go to g.

c) If the instruction for which we are allocating a register is not a register store, go to g.

d) If it is a register store into an A or a B register, exit RESETREG since it only handles X registers.

e) If it is a register store of X0, X6 or X7, go to g.

f) Increment or decrement the number of load registers available depending on whether we are allocating or deallocating.

g) If we are allocating a register, go to i.

h) Search the TEMPREGS table until a match is found and zero the entry to release the register. Decrease REGISTRS by one and exit from RESETREG.

i) Search TEMPREGS forward until a zero entry is found and store the tree address there. Add one to REGISTRS and exit.

MARKEM

This routine will determine if an R-list instruction is eligible for issuance (i.e. all logical predecessors have been issued). If so, the USABL bit is set in the final tree entry for the R-list instruction. On entry, X1 holds the address of the final tree entry for the just issued R-list instruction.

a) Set the USABL bit register to 1.

b) Load the next tree entry (towards FWADT3 from the current entry).

c) If the entry is zero, exit from MARKEM (end of tree).

d) If the entry has the logical predecessor bit set, go to h.

e) If the final entry bit is set, go to g.

f) Scan down the tree until the final tree entry for this instruction is found.

g) Set the value in the USABL bit register into the final tree entry. Go to a.

h) If this is the final tree entry, go to m.

i) If this entry is not marked (MARK=0), go to j, otherwise load the next tree entry and go to d.

j) If the predecessor of this tree entry is not the same as the instruction just issued, go to l.

k) Set the mark bit since the logical link formed by this entry has been satisfied with the issuing of the predecessor. Go to d.

l) Clear the USABL bit register since we have found an unmarked logical predecessor different from the one we have issued. Go to d.

m) If the final entry is already usable (USABL=1), go to a.

n) If it is already issued (MARK=1), go to p.

o) If the predecessor of this entry differs from the instruction issued, go to a.

p) If a previous logical link for this instruction has not been issued, mark this entry as issued since they are duplicate tree entries. Otherwise, set USABL=1 and mark the entry unissued (MARK=0). Go to a.

MOVETREE

This routine takes the issued instruction list which is pointed to by ISSUE1ST and forms a tree from their associated tree entries. The tree is built starting at LWAWORK and going towards FWAWORK. On entry, X1 holds the number of issued instructions. Upon exit, X6 will be minus if insufficient storage was available for building the tree. Otherwise, X6 holds the address of the next final tree entry to be moved.

a) LWAWORK is saved in MOVELWAW. Several pointer registers are initialized for constructing the tree.

b) Load a tree entry (first entry pointed to by ISSUE1ST and the remainder are chained off of it).

c) Load up the associated tree information word so it is the next tree pointer.

d) Clear all but the logical predecessor, successor and predecessor fields in the tree entry to be passed to OPT. If the entry has no predecessor, go to

e) If the tree entry of the predecessor indicates it is dead (DEAD=1), clear the predecessor field in the current entry.

f) Store the current entry in the subtree under construction.

g) If insufficient core is detected, exit with X6 negative.

h) Load the next tree entry associated with the final entry, we started at (moving towards FL).

i) If the final bit is not set in this entry, we have not yet reached the tree entries for a different instruction. Go to d and continue processing the tree entries for this instruction.

j) Reduce the number of instructions to be issued count. If it is non-zero, go to b to begin processing the next link in the issue chain.

k) If the forward link of the last entry is zero, we have issued the entire chain. Go to

l) If the next entry on the chain is not a store instruction, go to n.

m) Add the tree entry for the store to the subtree. Set NISSUED = NISSUED+1.

n) Set TREELNG (tree length) and FWATREE (start of tree).

o) Call INVERT to flip the subtree in memory and exit MOVETREE.

GETOT.N - obtain an optimizing temporary

This routine will provide the number of an available optimizing temporary. On return, X6 holds the number.

a) Extract the base of OT. table, highest current allocated OT., and highest allocated so far.

b) Initialize to the top of the table.

c) Is this entry available (less than zero)? If so, go to g.

d) If we have not exceeded the highest currently allocated, go to c.

e) Increment highest so far allocated this sequence by one. If it exceeds the highest so far allocated, increase the highest so far allocated field to match the highest allocated.

f) Repack the OT. control word and store it back. Change LWAWORK to reflect the table growth. Set the nth OT. table entry to +R and exit from GETOT.N.

g) Allocate the entry by complementing it and exit from GETOT.N.

KILLEM

This routine will place the dead bit into n instructions (in a linked list whose start is specified by ISSUE1ST). The value of n is in X1 on entry.

a) Load an entry from the linked issue list.

b) Set the dead bit in the tree entry.

c) Pick up the next link from the tree information word.

d) Decrease the instructions to be killed count by one.

e) If this entry did not need an optimizing temporary, go to g.

f) Extract the OT ordinal and call RELOT.N to release the optimizing temporary.

g) If more tree entries remain to be killed, go to a. Otherwise exit KILLEM.

RELOT.N

RELOT.N will release the optimizing temporary whose number is in X1 on entry.

a) Fetch the OT. control word (CTLOT.N) whose format is

```
VFD  6/0, 18/OTN, 18/N, 18/P
```

P = first word address of the optimizing temporaries table

N = the number of the highest currently allocated temporary in this sequence

OTN = the highest OT. number allocated for any sequence so far

b) Fetch the table entry to be released and set it negative.

c) Scan backward from the highest currently allocated OT. and release any entries that have been freed from the end of the list.

d) Reconstruct and update CTLOT.N and set LWAWORK to reflect a possible shrinkage of the temporary table.

REGCOSTS

On entry, X2 holds the tree entry of an instruction for which to compute cost. On exit, X6 is the cost or minus to indicate failure.

a) Fetch the descriptor and tree entry of the predecessor of the current tree word.

b) If the predecessor is not usable, exit with failure indicated.

c) If the predecessor is marked, go to o.

d) If the predecessor has no register predecessors, go to s.

e) If the predecessor has other than one predecessor, exit with a failure indicated.

f) Fetch the descriptor of the predecessor of the predecessor to the tree entry we are working with.

g) Fetch its tree entry in preparation for recursion.

h) If its uses count is one, go to j.

i) Increment REGCOSTP by one and go to a.

j) If it is a load instruction, go to n.

k) If it is not a store, go to a.

l) If it is not a register store, go to a.

m) If it is a register store to an A or a B register or to X0, X6, or X7, go to a.

n) If load registers are available, go to a, else return with failure indicated.

o) Compute UMIN as the number of uses for the predecessor minus the number of times it has been used minus one. If UMIN is zero, go to r.

p) If UMIN is greater than or equal to USESMINP, return with a success indication of zero.

q) Set USESMINP to UMIN and return with a success value of zero.

r) If it is a load instruction (the predecessor), decrease the number of load registers available by one. Decrease REGCOSTP by one. Return with a success of zero.

s) If the predecessor is a load, decrease the number of available load registers (NLOADRGP) by one. Increase REGCOSTP by one and return a cost of MAXREGS (=8) minus REGCOSTP. Exit from REGCOSTS.

UNISSUE

a) If there are no instructions on the issue chain, exit UNISSUE.

b) Add the number we tried to issue (ISSUED) back into the total number of unissued instructions (NOTISSUD). Zero out ISSUED. Start at the last entry on the issue list.

c) If this is the end of the linked issued list, go to h.

d) Clear the mark bit in the tree entry. Extract the number of predecessors from the information word of this tree entry. If the uses on this entry is zero, go to e. CAll RESETREG to release the register.

e) If all the predecessors have been processed, go to c.

f) Decrease the predecessor count by one. Fetch the next previous (towards fl) tree entry. Fetch the descriptor of the predecessor to the entry we are unissuing. Decrease the USED count in the information word of the predecessor. (Since we were unable to issue the successor, the predecessor has not actually been used.) If the used count was less than the USES count, go to e.

g) Reallocate a register for this entry by calling RESETREG. This must be done because the register would have been released, in the process of issuing, when the USED count matched the USES count. Go to e.

h) Clear ISSUE1ST. Before doing so, prepare to go down the linked list starting at the first entry.

i) If this is the end of the list, exit from UNISSUE.

j) Fetch the next tree entry (towards FWADT3). If we have reached the end of the tree, go to n. If this entry is not a logical predecessor, go to j. If the predecessor of this tree entry does not match the final tree entry instruction address, go to j. (This means that this entry is not a successor of

the one we are unissuing and hence need not be considered.)

k) If the entry is a final entry, clear the usable and mark bits in this entry and go to j.

l) Clear the mark bit in the entry.

m) Load the next entry (towards FWADT3). Scan forward until the final tree entry is found. Then clear the usable bit in the final tree entry and go to j.

n) Link forward to the next tree entry on the issue chain and go to i.

CREATEOT

This routine will free up a register by storing away in an optimizing temporary (OT.) one of the values presently in a register.

a) On entry, clear all the optimizing temporary search registers. Set maximum registers to eight. Set MAXDIST=0.

b) Fetch a register entry from TEMPREGS. If this pseudo register is empty, go to *l*.

c) The TEMPREG entry holds the descriptor address for the instruction whose RI field is contained in that register. Compute the address of the R-list1 entry and extract the RI field. Scan the OPT.XOR table (OPT's register table) until an entry is found with a matching RI. Preserve the number of the physical register it was found in. Load the tree entry associated with that instruction and set DIST to one.

d) Scan down the tree (towards FWADT3) until an unmarked entry is found.

e) If the R-list1 address of the starting tree entry matches the predecessor field of the current tree entry being examined (i.e. this is a successor of the starting tree entry), go to g.

f) If this is not a final entry, go to d. Set DIST = DIST+1. Go to d.

g) If this is not a final tree entry, scan down the tree until the final entry is located.

h) If the final entry is marked, go to d.

i) Form the OT.REG entry by combining DIST, address of the R-list1 for the RI in the physical register, and the uses remaining in the OPT.XOS table when OPT failed.

j) Store this entry in the OT.REG associated with the physical register number.

k) If DIST is greater than or equal to MAXDIST, set MAXDIST to DIST and retain the physical register number in SAVEN.

l) Advance to the next entry in TEMPREGS. If all registers have not been examined, go to b.

m) Set the selected physical register number to the one at MAXDIST. If a store register is free, go to o.

n) Select from X6 or X7 the one with the larger distance as the physical register. Go to p.

o) If the physical register at the maximum distance is a load register, increment the number of available load register by one (since we are going to free that load register by placing it in an OT.).

p) REGISTRS = REGISTRS-1.

q) Scan forward through TEMPREGS until the pseudo register matching the physical register being released is found. Zero the pseudo entry to release it.

r) Insert the RI to be saved away into the R-list to reload it. Set it into the RJ field in the transmit prior to storing it. Set the uses count in OPT's remaining uses for registers to one. Place the value that was in that field into the uses field of the load for reloading the saved value. Set the bit in the descriptor denoting a load from an optimizing

temporary. Call GETOT.N to allocate an optimizing temporary. Place the optimizing temporary number into the store instruction and into the load instruction. Store the load R-list1 in place of the R-list1 which defined the RI to be saved. Replace the descriptor with the load descriptor retaining the tree pointer field. Put a zero used count in the tree information word. Set the number of predecessors to zero.

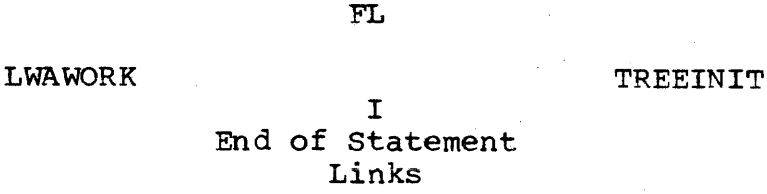
s) Clear the predecessor field and restore the tree entry. Clear the predecessor field in all the associated non-final tree entries.

t) Store R-list2 over the R-list2 of the instruction defining the RI to be saved. Clear the mark, dead, and predecessor field in the final tree entry of the new load instruction. Set the no register predecessors bit (ZEROP). Save the address of the tree entry for the last instruction placed in an OT. in LAST.OTR. Save the values of NORLIST and FWARLIS. Set NORLIST to indicate two instruction (XMT and ST). Set FWARLIS to point to the transmit. Form a tree for the transmit and the store. Set FWATREE and TREELNG to point to this tree. Call OPT to issue the transmit and the store. If OPT fails (he shouldn't), then abort.

u) Restore NORLIST and FWARLIS. Increase the number of unissued instructions by one (this is the load we added). Clear all indications of the register number 77777B used to issue the transmit and store from OPT's tables. Clear OPT's register which we just freed up to mark it released. Exit from CREATEOT.

6.0 Tables

6.1 Dependency Tree after Initial Call to BUILDDT



FWADT1

II
R-list Links

FWADT2

III
Equivalence Links

FWADT3

RA

Only part III may have a zero length. Each entry in the tree will be of the form

VFD 13/P, 10/XX, 1/PREDT, 18/R-list, 18/RPRED

where P = priority

XX = garbage

PREDT = predecessor type (1 indicates a logical predecessor, 0 indicates a register predecessor)

R-list = address of the R-list1 element for this tree entry

RPRED = address of a predecessor of the R-list entry at the address specified by R-list

6.2 Dependency Tree Information Word

VFD 12/USED, 12/NPRED, 18/LINKB, 18/LINKF

where USED =

NPRED = number of predecessors

LINKB = address of the R-list instruction issued before this instruction or zero if this is the first instruction issued

LINKF = address of the next R-list instruction issued after this one or zero if this is the last instruction issued

6.3 Dependency Tree Element Format

```
VFD  1/DEAD, 1/FINAL, 1/MARK, 1/USABL, 1/ZEROP,
     18/INFO, 1/PREDT, 18/R-list, 18/RPRED
```

where

DEAD — flags a dead entry

FINAL =

MARK = 1 if the instruction is issued

USABL = 1 if all logical predecessors are issued

ZEROP = 1 if no register predecessors

INFO = address of information word (section 6.2) for this entry

PREDT = 1 if this is a logical predecessor

R-list = address of R-list instruction (successor)

RPRED = address of R-list instruction which is a predecessor to R-list

6.4 Optimizing Temporary Table

The nth entry has +n if it is allocated and -n if it is free. The table is pointed to by CTLOT.N whose format is

```
VFD  6/0, 18/OTN, 18/N, 18/P
```

where

P = base of OT. table

N = highest OT. number currently allocated

OTN = highest OT. number so far allocated

SQUEEZE

1.0 General Information

SQUEEZE recognizes and removes redundant operations from an R-list sequence.

2.0 Entry Points

2.1 SQUEEZE

This is the primary entry and is called from PROSEQ to mark redundant operations.

2.2 REPLACE

This is primarily an internal routine. However, the entry is used by OPT=2 from PROSEQ. Its function is to replace all occurrences of one R number by another.

2.3 PURGE

This entry, which is called from PROSEQ, physically eliminates all RLIST entries whose kill bit was set by earlier processing (usually SQUEEZE).

2.4 RI(J)

The master R number (the one used to replace with) used by REPLACE is stored here before calling REPLACE.

2.5 FWADNEW

The first word address of the descriptors after purging.

3.0 Diagnostics and Messages

None

4.0 Environment

4.1 Externals

4.1.1 MOVE

Used in PURGE to collapse together R-list1, R-list2 and descriptors after packing down.

4.1.2 FWARLIS

First word address of the R-list1 words for the current sequence.

4.1.3 FWADESC

First word address of the descriptors for the current sequence.

4.1.4 NORLIST

Number of entries in the current R-list sequence.

4.1.5 FWAWORK/LWAWORK

First word and last word address respectively of working storage.

5.0 Processing

a. Initialize I, J and K to zero. Setup pointers to the R1 and descriptor area.

b. Scan backward through the sequence. If a register store is found, set the unkillable bit in the descriptor of the R-list entry which defines the RI to which the register store applies.

c. Set I to J. I=I+1. Pickup the descriptor of R-list(I).

d. If I is greater than NORLIST, exit from SQUEEZE. Set J to I.

e. If the I entry has been killed, go to c.

f. If it is not squeezable, go to c.

g. If the unkill bit is set, it may not be used as a master. Go to c.

h. For a load instruction, set K=0 and

(1) Set RI(J) to the master entry R-list(J).

(2) If it is commutative, go to h(30).

(3) I=I+1, if the end of sequence is reached, exit.

(4) If R-list(I) has been killed go to h(3).

(5) If R-list(J) is a jump, set K=0 and go to h(3).

(6) If R-list2(J) equals R-list2(I), go to h(11).

(7) If the master (RLIST(J)) is not a load, go to h(3).

(8) If R-list(I) is not a store, go to h(3).

(9) If R-list(I) or R-list(J) has no IH field or if the IH fields differ, go to h(3).

(10) Terminate squeezing since a store into the base has been found. Go to c.

(11) If R-list(I) is a load, go to h(24).

(12) If it is not a store, go to h(3).

(13) Do not squeeze if the types of I and J differ.

(14) Do not squeeze if differing commutivity.

(15) If CA (RJ,RK) fields do not match, go to h(3).

(16) If master (J) is a load, go to h(21).

(17) If so fields differ, go to h(21).

(18) Set the kill bit in the descriptor for K (the first store).

(19) Set K equal to I (make this the current store).

(20) Set RI(J) to the R-list1(K). and go to h(3)

(21) If R-list(I) is type 3, go to h(19).

(22) If master was a store (K not equal to zero) go to h(3).

(23) If the rest of the RK fields match, go to h(19); otherwise, go to h(3) (this is a short load, short store combination).

(24) If R-list(J) is not the same type as R-list(I) go to c.

(25) Go to c if differing commutivity.

(26) If CA and SO fields do not match, go to h(3).

(27) Mark R-list(I) as killed.

(28) Call REPLACE to change all references to RI(I) into references to RI(J).

(29) Go to h(3).

(Commutative bit on processing)

(30) I=I+1, if this is the end of sequence, go to c.

(31) Go to h(30) if entry I has been killed.

(32) If entry I is a jump, set K to zero and go to h(30).

(33) If master (J) is not a load, go to h(36).

(34) If R-list(I) is not a store, go to h(36).

(35) If R-list(I) is a store into the base of R-list(I), inhibit further squeezing (go to c).

(36) If R-list(I) is not commutative, go to h(30).

(37) If it is a load, go to h(43).

(38) If it is not a store, go to h(30).

(39) Compare the two entries directly and then after commutative RJ and RK. If neither match, go to h(30).

(40) If the master is a load, go to h(42).

(41) Set the kill bit for R-list(K) (the previous store).

(42) K=I, RI(J) = RI(I), go to h(30).

(43) Compare the two entries directly and then after commuting RJ and RK. If neither match, go to h(30).

(44) Set R-list(I) kill bit. Replace all references to RI(I) by references to RI(J) using REPLACE.

(45) Go to h(30).

i. For a store operation master (J).

(1) Set K to I and go to h(1).

j. Process non-memory reference master.

(1) If the master entry is commutative, go to j(9).

(2) Pickup R-list1(J) and R-list2(J).

(3) I=I+1, if this is the end of sequence, go to c.

(4) If the I entry is killed, go to j(3).

(5) If R-list2(J) not equal to R-list2(I), go to j(3).

(6) If R-list1(J) not equal to R-list1(I), go to j(3).

(7) If R-list(J) is not a shift transmit, set the kill bit and replace all references to RI(I) with RI(J) and go to j(2).

(8) For a shift transmit, R-list(I+1) must be the same as R-list (J+1) (excepting RI) if the kill is to occur. For no match, go to j(3).

(9) I=I+1, if this is the end of sequence, go to c.

(10) If R-list(I) is killed, go to j(9).

(11) If R-list (I) is not commutative, go to j(9).

(12) Compare entries both directly and commuted. If no match, go to j(9).

(13) Kill entry I and replace all reference to RI(I) by RI(J). Go to j(9).

OPTB

1.0 General Information

The function of OPT is to determine the order in which instructions are to be issued and to fix register assignments. The basic input to OPT is an R-list sequence and its associated dependency tree; the output is a series of calls to POST, each call giving the next R-list entry and register assignments to be converted to COMPASS line images. OPT is located in Pass 2.

2.0 Usages

OPT has one control entry point, OPTA, and two information entry points, LASLBL and LASTR.

2.1 Entry Point Name: OPTA

2.1.1 Entry Point Function: PROSEQ calls OPT at the entry point OPTA to cause an R-list sequence and associated dependency tree to be converted to COMPASS line images with fixed register assignments and code ordering appropriate to the target computer.

2.1.2 Calling Sequence and Returns: OPTA is entered by a return jump from PROSEQ. On return, if X6 is positive, successful processing of the entire sequence has taken place. If X6 is negative, the attempt has been unsuccessful, no COMPASS output has been produced, and remedial action (currently sequence reduction) must be taken.

2.1.3 Processing Flow Description: When OPTA is entered, all working cells are initialized according to the flag word, LASTR, which describes the terminal entry in the last previous sequence, and OPNPOST is called. Control is then transferred to the OPTB section which determines the next R-list entry to be processed and which, if any, registers are to be used. This information is passed on to ISSUE which updates all counts, tables, and clocks as required and calls POST to cause production of the desired COMPASS line image. The cycle, OPTB to ISSUE to POST to OPTB, continues until the last R-list entry in the sequence has been processed or until it is determined

that OPT cannot produce code successfully for the complete sequence. If production was successful, ISSUE will call CLSPOST to transfer the generated line images to the COMPASS file (COMPS) and exit to PROSEQ with X6 positive. If the conversion attempt was unsuccessful, control is returned to PROSEQ with X6 negative.

2.2 Entry Point Name: LASLBL

Entry Point Function: LASLBL is a flag word containing the R-list word specifying the last label encountered. This is set ordinarily by OPT at CLSPOST time but will also be set by PROSEQ for a sequence consisting solely of a label. LASLBL is used by OPTA during the initialization process.

2.3 Entry Point Name: LASTR

Entry Point Function: LASTR is a flag word containing the first word of the R-list entry terminating the last sequence if it was successfully processed or a zero otherwise. This is set by OPT before return to PROSEQ and by PROSEQ for a single-entry sequence. LASTR is used by OPTA to determine which working cells must be initialized.

3.0 Diagnostics

No diagnostics are produced by OPT.

4.0 Environment

The following cells are required to be set up before OPT is called:

Cell Name	Contents	Producing Processor
FWARLIS	first word address of R-list sequence being processed.	COPY
NORLIST	number of R-list entries in current sequence.	PURGE
FWATREE	first word address	BUILDDT

	of the dependency tree.	
TREELING	length of the dependency tree.	BUILDDT

In addition, OPT references all the 10 entry points in POST and the entry point NFPUNT in PRE.

5.0 Structure

The three major areas of OPT are OPTA, OPTB, and ISSUE.

5.1 OPTA

OPTA performs two main functions which prepare the environment for OPTB and ISSUE:

1) Clearing out cells containing unwanted information left over from the last sequence processed.

2) Processing R-list defines and initializing flag words.

5.1.1

The contents of LASTR determine whether OPTA performs the clearing function.

Clearing results when LASTR = zero, return jump, entry, label.

The following cells are cleared:

A-Register Contents
X-Register Contents
Register Availabilities (packed zeros)
Scratch Registers
Function Unit Availabilities
CLOCK, PARCEL
NXTSTOR, NXTLOAD
N6AVAIL, X7AVAIL

In addition, the following cells are set:

FCHAR = 45B (this forces upper for the first generated COMPASS instruction)

ADR = 1 (this defines the contents of A0 to be R=1)

5.1.2 OPNPOST is called for every sequence. NORLISTR, FINALR, TREETOP, TPRIME, IPOINT and SIZEFALT are initialized.

Defines are processed by storing the RI field in the corresponding Register Contents Cell, packing the descriptor 'uses' field into the Register Availability word, removing the defined register from the scratch list, and calling PRUNE to remove the RI from the tree of unprocessed R-list entries.

The descriptors for the sequence are scanned to initialize CODSIZE (total parcel count) and NOSTORE (number of store instructions). X6 and X7 are removed from the scratch register list if there are more than two stores, and X7 is removed if there are exactly two.

The interword time, WRDTIME, is set to 1 if the target machine is the 6400 or if in-stack timing is to be used on the 6600. This latter condition exists when CODSIZE≤28 and the last entry in the sequence is a conditional branch to the most recently encountered label. Otherwise, WRDTIME=5.

Exit is to OPTB at OPT.1B.

5.2 OPTB

OPTB selects the next R-list entry to be processed and determines which registers are to be used. This information is transmitted to NOCODE if no COMPASS code is to be produced or at one of the ISSUE entries. If no issuable entry can be found for the current time and parcel in the object machine, exit is made to ISSUEI with the pointer, IPOINT, negative to request remedial action by ASYNCH.

OPTB starts by scanning the dependency tree, starting at TPRIME, looking for the first entry which will fit at the current parcel and all of whose predecessors have already been issued, i.e., an R-list entry all of whose predecessor fields in the tree have been set to zero. Such an entry is said to be logically issuable (LI).

Each LI entry is then checked to see if it is machine issuable (MI), i.e., to see if there is a function unit and uncommitted destination (result) register available at the current simulated clock time. (for 6400 code

selection, functional units and result-registers are always available). The first check is for availability of the function unit. If none is found, the dependency tree is further scanned looking for LI entries. Most of the OPTB code is concerned with location of a destination register. If one can be located, then the entry is MI. If the instruction is a jump or a store, no destination register is involved. If a specific destination can be determined from the SO field in the R-list entry or if the instruction precedes a register store specifying the destination, it is necessary only to check whether the register is otherwise uncommitted and can be used for a result at current clock time. If neither of the above conditions exists, all feasible registers are checked for remaining uses and for availability at current clock time. The uses check includes the possibility of using a source (operand) register as the destination register. If a destination register is located under the above conditions, the instruction is MI.

Next, the instruction is checked to see if it is machine executable (MX). If it is a register store or a dummy transmit, exit is made to NOCODE with IPOINT designating the instruction. Otherwise, the entry is MX if its operands are available at the current simulated CLOCK time. If they are, IPOINT is set to indicate the instruction, the function unit and registers are recorded in FUNPRIME, DESTPR, JPRIME, KPRIME, the delay (DELTAT) is set to zero and control is transferred to ISSUEX. If they are not available and IPOINT is not already pointing to another instruction, then IPOINT is set to point to this R-list entry, the function unit and registers are recorded, DELTAT is set to the delay before execution, and the scan of the tree is continued, looking for an MX entry. If the end of the tree is encountered before an MX entry is found, control is transferred to ISSUEI which will process an issuable instruction if IPOINT designates an MI entry or attempt corrective action (ASYNCH) if IPOINT =-1.

5.3 ISSUE

The primary function of ISSUE is the maintenance of various clocks, tables, and pointers involved in machine simulation and code selection. ISSUE also passes information to POST for the generation of COMPASS line images. ISSUE receives control from OPTB and returns to OPT.1B unless it determines that code selection for the

current sequence has been completed or that no further progress can be made in code selection. In either of the latter cases, exit is made through OPTA to PROSEQ with X6 flagging success or failure.

Control is transferred to ISSUE at NOCODE, ISSUEI, and ISSUEX from OPTB and to PRUNE from OPTA.

5.3.1 Tables maintained by ISSUE:

Register Contents (Sec. 6.1)
Register Availability (Sec. 6.2)
A-Register Contents (Sec. 6.3)
Scratch Register List (Sec. 6.4)
Function Unit Availability (Sec. 6.5)

Cells maintained by ISSUE:

CLOCK	current simulated object machine time for this sequence
ISUCLOK	minimum simulated clock time for the next instruction issue
PARCEL	currently available parcel in object machine word
ISUPRCL	minimum parcel for next instruction
NXTSTOR	minimum clock time for next store instruction
NXTLOAD	minimum clock time for next load instruction
X6AVAIL	clock time when X6 is available as a destination register
X7AVAIL	clock time when X7 is available as a destination register
NORLISR	current number of R-list entries left to process
THISR	R-list address of entry currently being processed
NOSTOR	number of store instructions remaining in

the sequence

TPRIME	address of the first (highest priority) unissued tree entry

5.3.2 PRUNE

PRUNE is a closed subroutine within ISSUE which removes all references to a given R-list address from the dependency tree. It is used primarily by ISSUE but also by OPTA during the processing of register defines.

5.3.3 NOCODE

Entry is made at NOCODE for processing R-list entries which produce no object code. Register stores and dummy register transmits are handled by NOCODE. Appropriate register contents and availability words are updated and the tree is pruned, but no COMPASS output is generated.

5.3.4 ISSUEI and ISSUEX

If no issuable instruction can be found at the current clock time and parcel number, entry is made to ISSUEI with IPOINT negative and control is transferred immediately to ASYNCH. If a machine executable instruction has been chosen by OPTB, entry is made to ISSUEX: if a machine issuable instruction has been chose, entry is made to ISSUEI with X5 containing the number of minor cycles until the instruction begins execution. In either case, IPOINT points to the tree entry containing the address of the R-list entry to be processed.

After appropriate initialization, the two paths merge and the following activities are performed:

1. The uses count for each operand entering into the instruction is decremented and X-registers are added to the scratch list when the uses count becomes zero.

2. PARCEL count and CLOCK are updated, taking account of word boundary crossing to reset PARCEL to zero and add WRDTIME into CLOCK.

3. The various tables and cells described in 5.3.1 are made current.

4. If the result register is an X, it is removed from the scratch list.

5. PRUNE is called to remove references to the current R-list entry from the dependency tree.

6. POST is called to generate COMPASS output.

7. Exit is made to OPTB if more work is to be done for the current sequence or to PROSEQ (after calling CLSPOST) if the last R-list entry has just been processed.

5.3.5 ASYNCH

ASYNCH attempts corrective action when OPTB cannot find an R-list entry to issue. If code selection is for the 6400, this corrective action consists of generation of a NO instruction if PARCEL = 3 or of adding X6 or X7 to scratch if possible. If ASYNCH is successful, exit is back to OPTB: if not, exit is through OPTA to PROSEQ with X6 marking failure. If code selection is for the 6600, the processing in ASYNCH is concerned primarily with advancing CLOCK to the next point in time when a function unit or result register becomes available. At each new future time, exit is made back to OPTB for another try at code selection. If PARCEL = 3 and out-of-stack timing is being used, the effect of NO generation is investigated at the various future times. When all else fails, attempt is made to add X6 or X7 to the scratch list. If ASYNCH is unsuccessful, exit is made through OPTA to PROSEQ with X6 marking failure.

6.0 Formats

Five tables are used by OPT in connection with machine simulation.

6.1 Register Contents

Cells X0R through B7R contain, in the rightmost 18 bits, the number of the R currently residing in that register in the simulated machine. If the sign bit is set, the R has been locked into that register.

6.2 Register Availability

Cells X0S through B7S contain, in the rightmost 18 bits, the simulated clock time when the contents of the associated register are available as source input to a function unit. The upper 12 bits of each entry contains (in packed format) the number of remaining unissued instructions requiring the R found in the corresponding register contents cell.

6.3 A-Register Contents

Each pair of cells, A0CON through A7CON, contain the complete R-list1 and R-list2 entries used to set the corresponding A-register. These are used in the current implementation to convert 30-bit store instructions to 15-bit instructions when possible.

6.4 Scratch Register List

Cells X0SCR through X7SCR are used to keep track of the availability of the various X registers for use as destination registers. If the sign bit of a cell is set, the corresponding X register is unavailable as a destination. Otherwise, the lower 18 bits give the simulated clock time at which the register can be used as a destination.

6.5 Function Unit Availability

The 12 cells starting with BRANCH and ending with INC2 contain, in the rightmost 18 bits, the simulated clock time when the function unit becomes available for issuing an instruction.

SQZVARD

1.0 General Description

Eliminates redundant computations from a sequence of variable dimension code generated by DOPE.

2.0 Entry Points

SQZVARD

This is the only entry and is called from PROSEQ.

3.0 Diagnostics and Messages

MEMORY OVERFLOW IN SQZVARD

4.0 Environment

Externals

FWAWORK	Holds the first word address of working storage
LWAWORK	Holds the last word address of working storage
NORLIST	Holds the number of R-list entries in the sequence
PUNT	Error exit for memory overflow
FWADECS	First word address of the descriptors
OVERS	Number of parcels left over from the previous sequence
CODE.	Length of the CODE. block
PARCEL.	Holds the current parcel number
VDTAB	Holds the base address of the VARDIM table
FWARLIS	Holds the first word address of the R-list1 words

WRWDS Routine to write information to a file

5.0 Processing

a. Setup registers for accessing the R-list sequence.

b. If the VARDIM storage has been issued to COMPS go to s.

c. If there are no left over parcels, go to e.

d. Increase CODE. by one and clear the parcel count.

e. Initialize the address of the VDTAB to LWAWORK.

f. Pick up an R-list descriptor.

g. If it is not a store, go to f unless the end of the sequence is found. Then, go to s.

h. If this entry has been killed, go to f.

i. If we are not looking for a master store instruction go to q.

j. Set the master store flag (X0). Save the address of this store in STORE1.

k. If VARDIM storage has gone to COMPS (this means we failed if this condition is true), go to f.

l. Increment the length of CODE. by one and save the address for this VARDIM cell. Load a BSS 0.

m. Store the BSS at FWAWORK +1. Test for memory overflow and update FWAWORK by two.

n. Store the address definition in the appropriate cell in the VDTAB. (ordinal = H field in R-list2).

o. Check to see if this created a new maximum size for the VDTAB. If so check for memory overflow.

p. Convert the H field to display code and generate the VARDIM label and store it before the BSS, go to f.

q. If the RI of the master is different from that of the current store, set the kill bit for the current one.

r. If a failure has occurred, go to f. Otherwise, load a BSS 0 and go to m.

s. Setup to advance to the next master store instruction. If not end of sequence, go to f.

t. If failure has occurred, then exit SQZVARD.

u. Move all BSS 1 instructions to the end of the following BSS 0's in working storage.

v. Set VARTAGS and dump COMPS images for the BSS storage. Exit SQZVARD.

MACROE

1.0 General Description

MACROE expands a macro reference into proper R-list information.

MACROE is part of Pass 2. It processes one macro reference each time it is called.

2.0 Entry Points

MACROE has only one entry point.

MACROE

One macro reference will be expanded into R-list.

Calling Sequence:

SA1 "first word address of the macro reference". MACROE is called by PRE. Upon returning to the caller, the macro will have been expanded starting at MACBUF the external location and the number of R-list entries the macro expanded into will be found in MACWRDS. Both MACBUF and MACWRDS are contained in READRL.

The macro parameters are strung out one per central memory word starting at PARAN which is an entry point to MACROX. The macro descriptor is picked up and the macro text is transferred to the macro expansion area. All fields of the test that have to be modified are processed during this transfer. When the complete text has been transferred, MACROE will exit.

3.0 Diagnostics

No diagnostics are produced.

4.0 Environment

4.1 The following cells are referenced and expected to be set accordingly:

RNAME (RA+64B) (same as NRLN) contains the next R register to use.

DESCR external to MACRO start of the descriptors. This address is used as the starting point to index into the descriptors.

PARAN external to MACROE - is the starting address of storage used for expanding the calling parameters for each macro reference. Starting at PARAN (in MACROX), there is sufficient room to expand the maximum calling sequence.

MACORG - external to MACROE that is set by an EQU in MACROX. It is set to the table bias for the macro numbers.

DOMACK the bias added to the macro descriptor to find the DO macros.

MACBUF is the starting a ddress for storing the expanded macro.

4.2 MACROE will have set the following cells upon exit:

RNAME will be updated to reflect any register numbers (R's) generated in the macro expansion.

MACWRDS - will contain the number of words in the macro expansion.

MACBUF external to MACROE. First word address of the area in which the macro is expanded.

MACWRDS set to the number of words in the macro expansion.

MACNXT - set to MACBUF if there were no R-list entries to expand.

5.0 Structure

The expansion can be broken into three distinct portions.

5.1 Pick up macro descriptor.

5.1.1 The macro number is adjusted by the table bias (MACORG), and the descriptor is picked by indexing into the macro descriptor table which starts at MACDESC. This descriptor contains information to do the following:

5.1.2 The initial value of the register number (R's) to be used for this macro expansion is saved and then updated to reflect the number of registers to be generated in this expansion.

5.1.3 The macro length is extracted and placed in MACWRDS.

5.1.4 The descriptor for each macro contains the number of registers passed as formal parameters. The number of symbolic fields (IH) and the number of constant fields (CA) passes as formal parameters. These fields are extracted from the descriptor and saved.

5.1.5 A read register is initialized to the first word of the actual macro text.

5.2 Expand the actual macro parameters.

5.2.1 The number of R's, IH's and CA's have been saved from the macro descriptor. They are now extracted from the macro referenced area and expanded starting at location PARAN. They are unpacked so that each parameter occupies one word.

5.2.2 The addresses of the start of each field are kept in B registers. This allows indexing into the list when an actual parameter substitution is required.

5.3 Expand the macro text.

5.3.1 One word is read from the text. The OC is extracted and used to index into the descriptors to determine the type of R-list desired.

5.3.2 TYPE1 - The three R fields are extracted, examined to determine what, if anything, should be substituted for them. They are then reformed and added to the expansion area. In all cases, R fields may be replaced with a parameter R, a generated R, or they may be left alone (in case of A0 or B0).

5.3.3 TYPE2 - The RI field and CA field are examined to determine if any change is necessary. The lower 13 bits

of the 14 bit S0 field are saved. (The upper bit is an indication as to whether or not the CA field is an actual parameter.) The R-list entry is combined and added to the macro expansion area.

5.3.4 TYPE3

First word has a replaceable RI and CA field. The lower 13 bits of the S0 field are saved. The RI and CA fields are replaced if necessary. The entry is reformed and added to the macro expansion.

The second word is read up. It has a replaceable IH and R field. The H2 field, if any, is destroyed. The fields are processed, the entry recombined, and added to the expansion.

5.4 This continues at 5.3 until all the text has been processed.

6.0 Formats

6.1 No flags or tables are held or built by MACROE.

6.2 The expanded macro is in the form of regular R-list as described in the R-list write-up. The formats of macro descriptor and macro text are given there also.

7.0 No Modification Facilities

8.0 Method

The macro is expanded one word at a time, replacing actual fields as necessary.

9.0 The coding in MACROE has been reordered several times to improve its timing. Any modification should be done with care.

BUILDDT

1.0 General Information

1.1 BUILDDT builds a dependency tree from a specified sequence in the R-list. The tree will reflect the following:

1.1.1 The requirements of an R-list entry for results produced by another R-list entry.

1.1.2 The requirement that store operations take place within the most immediate surrounding jumps, (stores must take place prior to the following jump and after the preceeding jump).

1.1.3 The requirement that all references to identical IH fields are in the same relative order as originally in the R-list. (This is to make sure A(5) is stored into last in the following example.)

```
      DO 10 I=1,10

      A(I)= expression

10    A(5)= expression
```

1.1.4 The requirement that the updating of DO loop index functions takes place after all of their uses within the DO loop.

1.2 The following conditions will prevail when BUILDDT exits:

a) If there was insufficient working storage for the tree to be built, X6 will be negative.

b) Otherwise, X6 will be positive and the entry point TREELNG contains the length of the tree, the entry point FWATREE contains the start of the tree, FWAWORK has been updated to save the tree, the descriptor for each R-list entry include the number of times the entry is used as a predecessor. Each entry in the tree occupies one computer word in the following format.

```
VFD 24/0,18/Successor,18/Predessor
```

The predecessor and successor fields are each 18 bits long and contain the address of the operation within the R-list. The tree is ordered such that the entry with the highest priority appears first.

2.0 Usage

2.1 Entry Point Name: BUILDDT

2.1.1 BUILDDT is entered with a sequence in the R-list specified for which a tree is to be built. The start of the R-list (FWARLIS), the number of entries in the R-list sequence (NORLIST), and the available core limits FWAWORK, LWAWORK are all accessed.

2.1.2 BUILDDT is entered via a return jump. It exits through its entry point with X6 negative if there is insufficient room; otherwise, X6 is positive and the conditions listed under 1.2 above prevail.

2.1.3 BUILDDT is divided into three parts; the building of the tree, the calculation of priorities, and the sorting of the tree. Initially, a temporary tree is built starting at LWAWORK and extending toward FWAWORK. The tree is then sorted such that all equal predecessors are together and in ascending order. The priorities are then calculated. A priority sort is then performed and the tree is moved to start at FWAWORK and grow toward LWAWORK.

3.0 No diagnostics are produced.

4.0 Environment

The following information must be provided in the following cells, all external to BUILDDT.

4.1 FWAWORK, LWAWORK must contain the available working storage limits.

4.2 FWARLIS must contain the first word address of the sequence.

4.3 NORLIST must contain the number of R-list entries in the sequence.

4.4 It is expected that the sequence will have been transformed into three sections that lie in series. The first section contains the first word of each R-list entry, the second section contains the second word of each R-list entry, (zero for R-list types of one word length) and third section contain the descriptor for each R-list entry. PRE placed the sequence in this format.

5.0 Structure

5.1 First Section - Forming the Tree

5.1.1 After initialization, one pass is made through the R-list to link any store instructions to surrounding jump instructions. The first store instructions are linked to the end of the sequence with an instruction time of 10 (time for a store) and a total time of minus one. (This total time and instruction time are used in the priority calculation.) As each store instruction is linked, its address is kept in another list. When/if a jump instruction is found, all of these store instructions are made successors of the jump instruction. After this list of store instructions is depleted in this manner, this search is continued. Store instructions are now made predecessors to this new jump instruction and will be made successors to a preceding jump instruction if there is one. This search terminates upon detecting the start of the sequence. During this pass through the R-list, the uses of B1 are noted and made predecessors to the initialization of B1 for DO loop usage (if there is such an initialization).

5.1.2 Now the tree entries for register dependency and for DO loop functions are made. At the same time we do this, information is saved to enable us to later accomplish the sequential linking of IH fields. Since, in a well behaved DO loop, addresses can be accessed through B registers, short loads and stores are also entered into this list of IH fields. An R-list entry is read and unpacked. If it has a negative exponent, this indicates a DO loop function is being updated and linkage to this functions use is required. The RI field is extracted and compared to all prior RJ, RF, and RK fields in the sequence. Any that are equal to this RI result in a tree entry. This is done in a routine called SUBLINK. The instruction is repacked, with a positive exponent, and stored back into the R-list1 section. We then determine

the type of R-list entry, extract the appropriate registers and search the rest of the R-list for the definition of these registers. If they are not defined in the sequence, they are linked to the start sequence. If they are defined, the instruction for the definition is extracted from the descriptor and included in the temporary tree. The predecessor count in the descriptor is updated by 1 only when its resultant register is needed in another operation. This continues until the whole R-list has been examined and proper tree entries made for it.

5.1.3 At this time, the list containing the IH field information is processed and tree entries made, if necessary. The list is searched for a usable entry. When one is found, its IH field is extracted and the rest of the list is searched to find if there is an entry specifying that a store was made into this IH group. If a store was made, the entries using this IH field are sequentially linked so that these operations take place in the same order in which they are specified in R-list.

5.2 Second Section - Priority Calculation

5.2.1 The tree is sorted so all entries with the same predecessor are together and the list is in ascending order. This is done to reduce the number of memory accesses to accomplish the building of priorities.

5.2.2 The building of priorities starts with entries linked to the end of the sequence. The priorities are built by working backwards in the tree and calculating the maximum amount of time it will take to reach each point in the tree. When a node is found in the tree for which the time to reach has not been determined for all paths to the point, one of these paths is chosen and a forward search is made along this new path until a node in this path is found for which the time has been determined. The priority building starts again, backwards from this point. The priority field, which is in the upper 13 bits of each tree entry is set negative as each path to the point is analyzed and set positive when all paths to that point have been analyzed. When there are no entries other than those linked to the start of the sequence with negative time, the priorities have been calculated.

5.3 Sorting of the Tree - Two Parts

5.3.1 A simple sort of the tree is performed by comparing two entries and switching them if the second has a higher priority than the first one. Each time a switch is performed, a flag is set and after each pass through the tree, if the flag is set, another pass is made. This method was employed because the tree is fairly well sorted from the start as a result of the order the tree entries are initially made.

5.3.2 Any entries with a priority of zero (dangling code) are eliminated from the tree at this time and the uses count (see description of descriptor, section 27) of its predecessor is decreased by 1.

5.3.3 The tree is then transferred so it starts at FWAWORK and grows toward LWAWORK. During this transfer, all information but the predecessor and successor is extracted from each entry. When entries with equal priorities are encountered, the section of equal

6.0 Formats

6.1 Tree

VFD 13/TT,15/IT,18/Successor,18/Predecessor

TT - total time to get to this successor in tree

IT - time to get from this predecessor to this successor

PASS15$

1.0 General Information

Task Description

The interface controller for pass 1.5 sets up subscripts used by the FORTRAN portions of pass 1.5 in referencing tables created in pass 1. After initialization the main processing routine is called. When all processing is completed the random file index is collapsed, the random file is closed, and pass 2 loaded.

2.0 Entry Points

2.1 PASS15$

PASS15$ is the main entry point. It is entered by a transfer and exits by loading pass 2.

2.2 CLOPT

CLOPT is referenced by READR when it finds that the OPT file is not a random file on a device capable of performing direct access. It is entered by a transfer and exits by loading pass 2.

3.0 Diagnostics and Messages

None

4.0 Environment

O.SYM and O.SYMEND are expected to contain the origin and end of the two word symbol table, respectively.

5.0 Structure

5.1 The structure can be divided into two portions: initialize and cleanup.

5.1.1 The initialization phase takes the origin of the two word symbol table minus the base of blank common and puts the result in the first word of the SYMDAT common block. Similarly, the end of the symbol table is biased and placed in the second word of the common block. Then BLDSEQ is called to perform primary processing.

5.1.2 When control returns from BLDSEQ, the number of invariants is computed (NOOFINV-1) and stored in IT.SIZE. The starting value for pass 2 R numbers is then initialized. The index is then collapsed. This is done by summing the total words for each sequence in the same PRU and creating a single entry holding the total and the PRU number. The reduced index length is placed in the FET and the OPT file is closed with autorecall to write out the index. Then, the registers are initialized and control is transferred to LOVER to load pass 2.

6.0 Formats

6.1 Index for the OPT File

6.1.1 Before Collapse

```
        VFD   1/R,17/a1,18/l1,24/P1
                    .
                    .
                    .
        VFD                 ,24/P1
        VFD                 ,24/P2
                    .
                    .
                    .
        VFD                 ,24/Pm
                    .
                    .
                    .
        VFD   1/R,17/an,18/ln,24/Pm
```

P = pru number
l = sequence length
a = address in block
R = residence bit
    0 if on disk
    1 if in core

6.1.2 After Collapse

```
        VFD   18/0,18/l1,24/P1
                    .
                    .
                    .
        VFD   18/0,18/ln-1,24/Pn-1
        VFD   18/0,18/ln,24/Pn
```

P = pru number
l = length of block

MACRS

1.0 General Information

This routine is a collection of tables used by the FORTRAN programs and describes all R-list macros.

2.0 Entry Points

None

3.0 Diagnostics and Messages

None

4.0 Environment

Not applicable

5.0 Processing

None

6.0 Table Formats

6.1 RMACRO Definition

The form of an RMACRO reference is:

```
name      RMACRO          nosy, nor, nok, sqt, cand
```

name - name of the R-list macro
nosy - number of symbols
nor - number of R numbers
nok - number of constants
sqt - sequence terminator value
cand - candidate field value

6.2 ENDR

This macro terminates an R-list macro. The form is ENDR.

6.3 Parameter Block

A group of common blocks are constructed which describe parameters to each macro. Each macro has an entry in every block of the following form:

```
VFD  6/P1,6/P2,6/P3,6/P4,6/P5,6/P6,6/P7,6/P8,6/P9,6/P10
```

where P1,...,Pn represent parameter numbers and the first P value of 77B terminates the list.

Blocks constructed are:

- LD BLK - parameters which are loaded
- ST BLK - parameters that are stored into
- JP BLK - parameters which are transferred to
- CA BL - constant parameter number which contains the bias associated with each entry in LDBLK
- CA BS - constant parameter number which contains the bias associated with each entry in STBLK
- RF BL - register parameter number of an RF field associated with each entry in LD BLK
- RF BS - register parameter number of an RF field associated with each entry in ST BLK

6.4 Descriptor Block

An additional common block MACDESC has the following form:

```
VFD  6/SQT,6/CAND,6/NOSY,6/NOR,6/NOK
```

where:

- SQT - sequence terminator field
- CAND - candidate field, used in invariant code selection
- NOSY - number of symbols
- NOR - number of R numbers
- NOK - number of constants

Values of SQT are:

Value	Meaning
1	Terminate sequence after this macro
2	Terminate sequence before this macro
3	This macro is a conditional transfer
4	This macro causes an external reference
5	This macro exits (STOP, RETURN)

Values of CAND are

Value	Macro Type
0	Normal, no action
1	Single Binary operation, candidate for removal
2	Double Binary operation, candidate for removal
3	Load
4	Store
5	Set to a constant
6	Initial I/O call
7	Intermediate I/O call
8	Final I/O call
9	Special I/O call
10	Mode Change
11	Aplist entry

BLDSEQ

1.0 General Information

Task Description

BLDSEQ reads the sequential R-list file produced by pass 1 and constructs the random OPT file. When this is completed, the routine controlling the optimization of the OPT file is called. When optimization is completed, control returns to BLDSEQ and BLDSEQ exits to PASS15$.

2.0 Entry Points

BLDSEQ

This is the only entry point. It is entered by a return jump and exits via a RETURN.

3.0 Diagnostics and Messages

None

4.0 Environment

4.1 Requires SYM1, SYMEND and the two word symbol table.

4.2 Requires the following common blocks to be initialized: LDBLK, STBLK, JPBLK, MACDESC, RBLK.

5.0 Structure

5.1 Setup and GET Phase

First a call is made to OPENMS to establish the OPT file index address and index length. Then, we call GET which returns a word of R-list. If the word is negative, we iterate the GET process. At this point, we split processing for macros from that for ordinary R-list instructions.

5.2 Macro Processing

Using the unbiased macro number, we now index into the macro information table (MACINF) to find out if it can end a sequence. If it is a normal macro, we simply copy the remaining words to the sequence area and return to the GET phase. If the sequence ends after this macro, we copy the remainder and perform "end sequence" processing. When the sequence should have terminated before this macro, we perform "before sequence" processing. If the sequence exceeds half the size of working storage, we force an artificial termination.

5.3 "Before sequence" Macro Processing

If the macro is a label macro, we extract the IH field and check the symbol table to see if the label is do loop generated or active. For inactive labels, we merely continue sequence accumulation. Otherwise, we call ADDROW and CONNECT to make new entries in the connection matrix. Then, we call PUTMS to place the sequence on the OPT file, increment the sequence number and use the "before sequence" entry to initiate a new sequence.

5.4 "End sequence" Macro Processing

If the macro is a computed or assigned go to macro, we perform special actions to accumulate the transfer lists and make entries for them in the connection matrix. Then, we perform an ADDROW and CONNECT call to extend the connection matrix. If the next R-list word is an end-of-statement, we will include it in the current sequence. Then, we use PUTMS to place the sequence in the OPT file and increment the sequence number. If the last macro was not an end-of-R-list, we return to the GET phase to begin a new sequence. For an end-of-R-list, we execute terminal procedures.

5.5 External Calls and APLISTS

All external calls are recognized by a type of "D" but do not break a sequence since it may be possible to remove invariant code from a loop with external references.

APLIST macros must be checked in case they form part of a RETURNS list and hence an implicit flow path. A

RETURNS APLIST macro is treated as an unconditional transfer and entered into the connection matrix. It also terminates the sequence under construction.

5.6 R-list Instruction Processing

The type of the R-list instruction (before sequence or end sequence) is extracted from the RCOD table. The number of words in the instruction is computed and processing proceeds via the code used for macro processing.

5.7 Terminal Processing

When we encounter the end-of-R-list, we call DUMPMS to force out any portion of the OPT file remaining in core. Then DOOPT is called to perform all phases of optimization. Upon return, DUMPMS is called again if any sequences remain in core (PEND .NE. 0). Then, we return to PASS15$.

6.0 Formats

SEQ (SEQTAB)

The sequence table is a sequential table containing the current R-list sequence. This table may contain a number of sequences but only one can be referenced at a time. The subscript ABSB marks the absolute base of the table. SEQB is the relative base of the current sequence and LOCINSQ marks the location currently being examined. Subscripts other than these three should not be used in referencing the table since the whole table may be moved during a call to CHECK. In this case, only these subscripts will be adjusted.

GET

1.0 General Information

Task Description

GET returns the next word from the R-list file. The word is obtained from the working storage area until that becomes exhausted. Then more R-list will be read into the working storage area.

2.0 Entry Points

2.1 GET

This entry is referenced by CALL statements. The single parameter returned is the next word from the R-list file. Exit is by a RETURN statement.

2.2 PEEK

This entry is referenced by a CALL. The parameter returned is the next word of R-list but the word returned is still available to the next PEEK or GET call.

3.0 Diagnostics and Messages

None

4.0 Environment

Not applicable

5.0 Structure

If GET is called, the MODE flag is set to 0; for PEEK, the value is 1. If words remain in RBUFF, we extract the one at RBUFF (LOC IN RB). If PEEK was called, we now exit. For GET, we increment LOC IN RB. When working storage is empty, we call READR to replenish the

array, set NWDS to the amount read and then return a word as above.

CONNECT

1.0 General Information

Task Description

CONNECT takes the R-list macro which broke the current sequence and builds connection information regarding this sequence and other sequences.

2.0 Entry Points

CONNECT

This is the only entry point and it is entered by a CALL. There are no arguments. Exit is via a RETURN.

3.0 Diagnostics and Messages

None

4.0 Environment

The following variables in COMMON must be setup before entry:

SEQ NO	- the current sequence number
ROP	- 2xxx if it is a basic R-list instruction or 0xxx if it is a macro
ENDSEQ	- type of sequence terminator
LEN	- length of the last macro
LOCINSQ	- points to next place to put a word of R-list in the sequence

5.0 Structure

5.1 Macro Processing

First, we extract the jump information about the macro from the data table JUMP. Next, processing divides according to the type of the sequence terminator (A, B, C, D, E).

5.2 A Type and C Type

A type indicates the sequence terminates after the current macro. We proceed to scan successive 6 bit fields in ISEQ. The first 77B entry ends the jump entries. If an entry is not 77B, it represents the number of the IH macro parameter which is used in a transfer within the macro. We extract this parameter and call SYMFND to obtain the associated column number in the connection matrix. If the label is still unknown, we merely resume our scan because SYMDEF will make the connections when the symbol becomes defined. If the column is known, we call SETBIT to mark a transfer from ROW to COL. Then, we increment JMP to indicate that we have made an entry in the connection matrix. When the scan terminates, we return unless the type was C and entries were made in the connection matrix. In this case, we also show a fall through to the next sequence from the current conditional transfer sequence and then return.

5.3 B Type

B type indicates that the sequence terminates prior to the current macro. If the current macro is a label, we extract the IH and call SYMDEF to associate the label with a sequence number and to fill in any forward references. We then add a fall through connection for the label. If it is not a label, we simply show a fall through connection, make entries for any jumps the macro may have and RETURN.

5.4 Type D and E

Type D is an external call and type E is an exit macro. Both types simply RETURN.

SYMFND

1.0 General Information

Task Description

SYMFND takes the IH field passed to it and searches the defined label table for it. If a match is found, it returns the associated sequence number. Otherwise, an entry is made in the forward reference table giving the current sequence number and the IH.

2.0 Entry Points

SYMFND

This is the only entry point. It is called by a statement of the form: CALL SYMFND (IH, NO OF SEQ), where IH is the IH field of the label sought and NO OF SEQ will contain the sequence number associated with the label if it was found, or zero if it was not.

3.0 Diagnostics and Messages

None

4.0 Environent

Not applicable

5.0 Structure

5.1 Search and Success

A scan is made of the defined label table (SYM) from the base (SYMB) to the last symbol entered (MAX SYM). Bits 0-29 of each word are compared against the IH for a match. If a match is found, the upper 30 bits of the same word hold the associated sequence number. This is placed into the second parameter, and we return.

5.2 No Match

If there is no match, we scan the forward reference table (TRA) from the base (TRA B) to the top (TRA TOP) seeking a zero word in which to place our forward reference. If we locate a zero word, we store the sequence number and IH of the symbol in it, zero the returned sequence number and exit. If the table is full, we expand it by fifty words, zero the fifty words, insert the forward reference, zero the sequence parameter and exit.

6.0 Table Formats

```
SYMB      VFD   30/Seq no 1,30/IH1
          VFD   30/Seq no 2,30/IH2
                      .
                      .
                      .
          VFD   30/Seq no n,30/IHn

MAXSYM


TRAB      VFD   30/Seq no 1,30/IH1
          VFD   30/Seq no 2,30/IH2
                      .
                      .
                      .
          VFD   30/Seq no n,30/IHn
          VFD   60/0
                      .
                      .
                      .
TRATOP    VFD   60/0
```

SYMDEF

1.0 General Information

Task Description

This routine is called when a label is encountered. It will make an entry in the defined label table (SYM) giving the IH of the label and the upcoming sequence number. Then it will satisfy any forward references in the TRA table making the appropriate entries in the connectivity matrix.

2.0 Entry Points

SYMDEF

This is the only entry point. It is entered by a statement of the form: CALL SYMDEF (LABEL) where LABEL is the IH of the label to be defined.

3.0 Diagnostics and Messages.

None

4.0 Environment

Variables

SEQ NO - should contain the number of the current sequence
MAX SYM - indicates the next variable location in SYM
TRA TOP - marks top of the TRA table

5.0 Structure

Processing

To obtain the proper sequence, we must make a simple check. Generally, the value should be that of the present number plus one since a label terminates the

sequence. In the case where some other macro such as an unconditional transfer ended the sequence just prior to the label, we must not increment the sequence number. It is sufficient to increment if the sequence area (LOCINSQ - SEQB) is non-empty.

The symbol and correct sequence number are entered into SYM at MAXSYM and MAXSYM is incremented with a check for table overflow. If the table overflows, it is increased by 100 words.

We now search the forward reference table for any matches with the IH of the label being defined. When a match is located, the origin sequence number of the transfer is extracted from the upper part of the TRA table entry and placed in ROW, and the destination sequence number is placed in COL. SETBIT is then called to make the appropriate entry in the connection matrix. The TRA word is zeroed and the scan continues. When the table has been completely searched, we exit.

6.0 Table Formats

See section 6.0 for routine SYMFND.

DOOPT

1.0 General Information

Task Description

This is the master controlling routine for all optimization performed in pass 1.5. It calls a number of secondary routines to perform the actual work.

2.0 Entry points

DOOPT

This is the only entry point and it is called from BLDSEQ. This routine has no parameters.

3.0 Diagnostics and Messages

None

4.0 Environment

Not applicable

5.0 Structure

5.1 Invariant Code Moving

To perform motion of invariant code, it is necessary to locate loops within the program. First, we initialize the sequence counter (LAST) to indicate the first sequence. Then, we call FNDLOOP which will advance LAST until it locates a suitable loop or exhausts all sequences. In the latter instance, it will return with LAST = 0. If it locates a loop, it returns with LAST equal to the sequence number of the loop. We then call FNDINVR to find all invariants within the selected loop. Upon return, we check the number of words in the invariant R-list table. If there are none, we increment LAST and search for another loop. If invariants were found, we call MOVINVR to move them to the proper spots

outside the loop. Then, we bump the value of LAST and search for another loop.

5.2 Register Candidates

Upon return from FNDINVR, the invariant code array may contain a loop begin macro describing the register candidates, invariant code or both. MOVINVR then properly moves the contents of the invariant block to a location preceding the loop.

FNDLOOP

1.0 General Information

Task Description

This routine searches the connection matrix for a loop. When it locates one, it returns with a count of the total number of entries and exits associated with the loop.

2.0 Entry Points

FNDLOOP

This is the only entry to the subroutine. The CALL has no parameters and exit is via a RETURN.

3.0 Diagnostics and Messages

None

4.0 Environment

Common Variables

LAST - common variable containing the next sequence number to examine for a loop.
SEQNO - largest sequence number.
ENTRY - variable containing the total entries to the loop; set on exit.
EXIT - variable containing the total exits from the loop; set on exit. The value -1 means not a loop.

5.0 Structure

5.1 Loop Detection

We initialize ROW to LAST and scan until we exhaust the connection matrix or find a loop. If we exhaust the matrix, LAST is set to zero and we return. To detect a

simple one sequence loop, we set COL equal to ROW and call TESTBIT. If TESTBIT returns a true value, then we have located a transfer from the row sequence entry to the column sequence entry. At this point, we exit the loop scan and proceed to count entries and exits.

5.2 Entry Counting

We set LAST to indicate the ROW currently being examined and initialize ENTRY to -1 since we are guaranteed at least two entries to a simple loop. For values of ROW from sequence 1 to the largest sequence (SEQNO) and holding COL fixed, we call TESTBIT. Each time TESTBIT succeeds we bump the entry count (ENTRY) of the loop.

5.3 Exit Counting

First, we initialize EXIT to -1. Then, we extract the pointer to the column vector from the row table. Using this, we pull up the first of the two column words and scan successive 18 bit fields from the left until we scan 3 fields (bits 0-53) or hit a zero field. For each non-zero field, the EXIT counter is incremented. We exit on the first zero field. Similarly, we examine field 1 and 2 of the second word. If we still have not encountered a zero field, we use the third field as a link to subsequent two word column entries. In addition to bumping the exit counter, we check to be sure an exit is either to the next sequence or to the head of the loop. If this conditon is not met, we reset EXIT to -1 and RETURN.

FNDINVR

1.0 General Information

Task Description

This routine detects invariant R-list within a loop found by FND LOOP. The invariant portion is removed to an invariant table (INV) where it is held until MOVINVR places it outside the loop. The code is squeezed out of the body of the loop leaving only, in some cases, a load of an invariant temporary. Upon exit, the invariant table may contain a loop begin macro and the body of the loop will be delimited by a loop end macro. The loop begin macro contains information about register candidates that may be assigned during pass 2 processing.

2.0 Entry Points

FND INVR

This is the only entry point. The call has no parameters and exit is via a RETURN.

3.0 Diagnostics and Messages

None

4.0 Environment

Common Variables

LAST - sequence number of the loop to be optimized
SEQB - base location of the sequence to be optimized

5.0 Structure

5.1 Setup

Initially, we reset pointers to the description table(DESC), the invariant table, the last end of statement flag (LASTEOS) and several other indicators. In order to optimize the sequence, it may be necessary to retrieve it from mass storage. To ensure its presence in core and to get SEQB set properly, we call GETMS with the sequence number (LAST) and it returns the length (LENGTH).

Before proceeding to locate invariants, we must know about every variable or array which is used or defined in the loop. A call to USE DEF results in the construction of a table (USE) providing the necessary information.

Upon return, if the common variable REDEF is non-zero, we abandon all optimization attempts for this loop. This can result from several things occurring in the text of the loop. An example of such a loop would be one performing a BUFFER IN statement. Since this I/O statement works with only a starting and ending address, the redefinition of other items in the loop is uncertain.

For later printing purposes, we now scan the text body for an end of statement marker and extract the line number of the head of the loop, (LINE NO).

5.2 Marking Phase (Phase 1)

Phase 1 of invariant elimination scans the body of the loop marking invariant macros. An invariant macro can be one of the following:

1) a load of an invariant variable; i.e. one never stored into within the loop.
2) a constant macro; i.e., set to a constant or a mask of a constant length.
3) a macro all of whose operands are invariant.

Initially, we extract the opcode of the macro and index into the macro information table (MACINF) using it. From this table, we obtain a macro type number.

Presently, there are twelve types which are treated. At this point, processing splits according to the type.

5.2.1 Type 0 (Normal)

This type number is used for all non-special macros and is the default value if no type number is provided for the macro. Processing consists of advancing the scan to the next macro.

5.2.2 Type 1 (Binary Operations)

This type is used in describing macros for which P2 and P3 are operands, P1 the result and no symbols and constants are present. Processing consists of a search to see if both P2 and P3 are among the register numbers in the table of invariant registers. If either is not, the macro is not invariant. Should both operand registers be in the table, we can enter the result register into the table, as a new invariant register. In addition, we will now mark this macro and both macros defining its operands as invariant. The marking of the two defining macros is essential in order to prevent the following occurrence:

```
load A to R1 (invariant)  marked
load B to R2 (variant)    unmarked
R1+R2  to R3 (variant)    unmarked
```

This would happen if we immediately marked invariant loads. Instead, we defer marking loads until we know that the load register will be part of an invariant computation. The defining macro address is kept in the description table along with the register number defined. Marking a macro consists of turning on the sign bit in the header word. Then set the no invariant flag (NO INV) false, indicating an invariant computation has been formed.

5.2.3 Type 2 (Double and Complex Binary Operations)

Presently not treated. Processing consists of advancing to the next macro.

5.2.4 Type 3 (Load)

Extract the symbol table ordinal and search the usage-definition table for a match. If the load macro has a

non-zero RF field, we make no entry in the invariant table. When a match is found, extract the bias field of the variable and check to see if the match in the description table involves an array. For an array match, we can ignore the array if the base-bias of the item being loaded does not fall within the bounds of the array. If these conditions are met, we make two additional checks. First, is the redefinition bit set for the item in the table. This will be set for nonstandard subscript references and forces the load to be marked as variant, even though it may not be. Secondly, we check the store bit. If set, the variable is the object of a store within the loop.

The series of tests for base-bias match, redefinition and store bit described above are performed for each entry in the usage-definition table or until the occurrence of a RDBIT or STBIT test which succeeds.

Finally, if all previous criteria are met, we must make a final check. Is the variable in common and does the loop contain an external reference? If so, we must not mark it invariant.

When all of the above are met, we enter the number of the load register in the description table but do not mark the macro as invariant. Processing then proceeds to the next macro.

5.2.5 Type 4 (Store)

For stores, we make basically the same check as for loads (Type 3). The only difference lies in checking the load bit (LDBIT) instead of the store bit. This ensures that the variable stored into is never used in the loop.

After ascertaining that the store is in fact invariant, we mark both it and its definition as invariant. This is necessary for the case L = 5, where L is invariant since the set register to a constant macro will be unmarked. In addition, we set the no invariants (NO INV) flag to false. Then, we proceed to the next macro.

5.2.6 Type 5 (Set to a Constant)

We enter the register number in the table of invariants and proceed to the next macro.

5.2.7 Type 6 (Initial I/O Call)
Type 7 (Intermediate I/O Call)
Type 8 (Final I/O Call)
Type 9 (Special I/O Call)
Type 11 (Aplist entry)

For each of these, the action is to proceed to the next macro.

5.2.8 Type 10 (Mode Change)

Extract the number of the register to be mode changed and search the invariant table for it. If no match is found, proceed to the next macro. Otherwise, mark both the mode change and the macro defining the operand as invariant, and enter the result number in the invariant table. Then proceed to the next macro.

5.3 Moving Phase (Phase 2)

If no invariants were found (NO INV = .TRUE.), exit to phase 3. During phase 2, we rescan the sequence once more. The first test made is for basic R-list instructions. All of these, except end of statement, are ignored. For an end of statement, we mark its location (LASTEOS) and the present size of the invariant code table.

The second test is for a label. Processing is similar to that of the end of statement except that we put the location of the label into LASTEOS. Both the label and end of statement locating code is concerned with the following case:

```
      A=0.
200   B=0.                    (label 200 inactive)
```

Here, there would be an intervening end of statement and label which are in fact not variant. The extra precautions ensure they are ignored.

Primary checking in phase 2 concerns those macros marked as invariant in phase 1. If a macro is unmarked, we simply advance to the next one. For marked store macros, we perform special processing. Otherwise, we copy the marked macro to the invariant area clearing the mark bit during the copy. In the loop body, the header is left marked and the body of the macro is set to minus

ones. (Later, negative entries will be squeezed from the loop body).

After copying, we perform a check to see if the next macro is also invariant. If so, we just continue copying. Otherwise, we check it for an end of statement and move it to the invariant block if it is one. Then advance to the next macro.

Depending on the last macro seen before the non-invariant macro, we select the appropriate processing.

5.3.1 Binary Operation Last

For this case, we generate a store macro in the invariant table to store the result of the invariant binary operation into an IT. location. The spot in the sequence body where the invariant code was extracted has been set negative. At this point, we use a portion of that space to create a load macro and a transmit operation. The transmit places the result from the IT. load into the expected register number. This process uses up two intermediate R numbers and an invariant temporary location each time it occurs. Then, we advance to the next macro.

5.3.2 Mode Change Last

In some cases, it will not be advantageous to remove computation involving a mode change. An example of this might be:

```
R=D                    D is double, R is Real
```

The part removed would be load of D and a transmit of the upper part. It would be replaced by a load from an IT. and a transmit.

To prevent this, we will not remove a load or a set to a constant and a mode change alone. Unfortunately, at this point the macros are in the invariant block. Therefore, we must remove them from there and restore them to the body of the sequence. Then reposition to the start of the previous invariant macro and revert to the last invariant macro processing phase. If there is no previous invariant macro, advance to the next macro from the mode change.

If the macro whose result is being mode changed is not a load or a set, we generate a store to an IT. in the invariant table and place a load and transmit into the loop sequence.

When the end of sequence is reached, all invariant code has been placed in the invariant table. The places in the main sequence occupied by the code all have the sign bit turned on. To conserve space, we now collapse out all dead entries. The number of words to reduce this sequence by (TEMP, less than or equal to zero) is computed during the squeezing.

5.4 Create Loop Begin Macro (Phase 3)

The entries in the loop begin macro (candiates for register assignment in pass two) are gleaned from the usage-definition table created by USEDEF. If the loop contains external references, the loop begin macro will not be generated. An entry is copied from the usage-definition table to the loop begin macro if:

1) it is not an array reference
2) all members of base redefined bit (RD BIT) is not set
3) there is a load of the variable within the loop

When we copy an entry, we increase the preload-poststore, length (PLPSLEN). This value will be used in MOVINVR to compute how much of the invariant table is really invariant R-list. If we have made entries in the loop begin macro, we append a loop end macro to the exit path of the loop.

Finally, we call UPDOWN with the count in TEMP. This compacts the sequences and eliminates TEMP words of dead space. Then, we exit by a RETURN.

6.0 Table Formats

6.1 Usage-Definition Table

See section on USEDEF

6.2 Loop Begin Macro

```
VFD  12/1751B,18/n,30/line
VFD  8/0,1/R,1/D,1/S,1/L,18/a1,18/CA1,12/H1
          .
          .
          .
VFD  8/0,1/R,1/D,1/S,1/L,18/an,18/CAn,12/Hn
```

line= line no. of loop top
Hi = symbol table ordinal (base)
ai = array size - 1
L = set if used
S = set if defined
D = set if defined before use
R = set if all members of base redefined
CAi = bias for use with equivalenced variables

6.3 Loop End Macro

```
VFD  12/1750B,18/0,30/line
```

line = line number matching that in loop begin macro

MOVINVR

1.0 General Information

Task Description

This routine moves the contents of the invariant table to a point in the R-list preceding the loop from which it was extracted.

2.0 Entry Points

MOVINVR

This is the only entry point. The call has no parameters and exit is via a RETURN.

3.0 Diagnostics and Messages

MOVINVR may issue a series of messages of the following form:

SUMMARY OF CHANGES MADE BY THE OPTIMIZER

nnn WORDS OF INVARIANT R-list REMOVED FROM THE LOOP BEGINNING AT LINE llll

4.0 Environment

4.1 Common Variables

LINV - length of the invariant code table
PLPSLEN - the number of words in the invariant code table which are part of a loop begin macro
LSFLG - set to zero if L=0 option set on FTN card
LINE NO - line number of the top of the loop
LAST - sequence number of loop being optimized
LOCINSQ - marks last word plus one of the sequence holding the loop body

4.2 Local Variables

LFN - file number to write messages on.
HEAD - set non-zero when SUMMARY OF CHANGES... has been printed.

5.0 Structure

Processing

If we have information in the invariant code table other than a loop begin macro, and we have not printed the heading, then print the heading (unless L=0 selected).

Now, we call PUTMS to write the loop body from which the invariant code was extracted. Next we locate the predecessor sequence to the loop body and compute its size (LENGTH). By appending an end of statement marker to the end of the invariant block, we aid sequence partitioning in case of OPT failures in pass 2.

The call to GETMS to obtain the predecessor sequence is made only after we verify that the sequence table will be large enough to hold the additional space added by the invariant table. UPDOWN is then called to move any other sequence down and provide LEN words of space for the contents of the invariant table.

For a DO loop, we must place all invariant code just prior to the DO begin macro. This necessitates moving the DO begin downward LEN words and inserting the invariant table. In all other loop types, we can just append the invariant block to the end of the predecessor sequence.

Finally, we print out the number of words of invariant R-list removed (if L.NE.0) and use PUT MS to rewrite the predecessor sequence which now includes the invariants. Then, we RETURN.

USEDEF

1.0 General Information

Task Description

This routine constructs a table (USE) which describes all uses and definitions of symbols in a sequence.

2.0 Entry Points

USEDEF

This is the only entry point. The call has no parameters and exit is via a RETURN.

3.0 Diagnostics and Messages

None

4.0 Environment

Common Variables

USEB - start of usage-definition table (hereafter UD table)
SEQB - base of sequence for which to construct UD table
ENDSEQ - number of words in the sequence

5.0 Structure

5.1 Setup

Initially, we clear the IO flag (indicator that we are in an I/O list) and the REDEF flag (signifies an abnormal occurrence in the loop such as a BUFFER IN which prohibits further optimization). USELOC (next available location in USE) is set to the base of the UD table and the sequence pointer (LOCINSQ) is set to the start of the sequence.

5.2 Examine R-list Macro

MODE is initialized to the bit value used in setting the load bit in the UD table. We extract the macro opcode and compute the length (LEN). If the macro is a "set R to an address", we make a special check to see if we are inside an I/O list. If not, we proceed to the next macro; otherwise, special processing is required. If the macro is part of a class of macros concerned with I/O, processing is split to several areas according to the macro. For the macros not in this category and with the exception of subscript macros, the following takes place:

5.2.1 Load Processing

Three words are initialized from tables constructed in MACRS. These words describe all loads made in the macro. LOADS contains in successive six bit fields (starting with bit 59) the numbers of IH parameters to the macro involved in loads. The list is terminated by a value of 77B. LOAD CA contains the bias associated with each base member denoted in LOADS. LOAD RF describes which R number parameters are used in an RF field.

5.2.2 Symbol Processing

We now enter a loop which examines successive 6 bit field of the three words (LOADS, LOAD CA, LOAD RF) to determine what should be entered in the UD table. When we hit a 77B, we go to store processing. Using the number from LOADS, we extract the IH field. If the IH field is non-zero (not a symbol table entry), we go to the next LOADS entry. For a LOADS entry of zero, we go to special processing for formal parameter loads or stores. Next, we extract the bias field using the associated LOAD CA entry and form a thirty bit base-bias field. Using that field, we compare against all existing entries in the UD table (including the array size field in the compare to prevent incorrect matches). There are two possibilities which are detailed in the next sections.

5.2.3 Symbol Not Found

Enter the base-bias and MODE bits (initially set for load) into the next entry in the UD table. Then proceed to RF processing.

5.2.4 Symbol Found

Set the bit indicated by MODE into the entry we found. For stores, we go on to RF processing. If MODE indicates we are doing loads, and the entry has no previous load bit, it must have been entered by a store usage.

Therefore, we set the defined before usage bit and continue with RF processing.

5.2.5 RF processing

Using the field from LOAD RF, we extract the RF register number from the macro. If it is zero, we go on to the next LOADS entry. In the case of a non-zero RF, we set the full redefinition bit in the entry from the symbol found or not found processing. Then, we scan all members of the UD table and set the full redefinition for any with the same base ordinal. Then, we go to the next LOADS entry.

5.2.6 Stores Processing

If we have done stores processing, proceed to the next macro. Otherwise, set up the three words from load processing (LOADS, LOAD CA, and LOAD RF) with the corresponding information on stores for the same macro. Change MODE so that we set the store bit and reenter the symbol processing phase.

5.3 Formal Parameter Processing

To obtain the symbol table ordinal, we must use the CA field + 2. Setup LOADS for a termination after the present cycle. In the case of stores to formal parameters, shift MODE left one to denote stores processsing. The only other exception is for the "set register to formal parameter address". This may be a load or a store reference so we set up MODE for both. Then, we go to the symbol search in the symbol processing section.

5.4 Subscript Processing

Initially, we extract number of subscripts and put it in TEXT and maximum size - 1 to MAX. For variable dimensional arrays (PABC field not zero), we set the full redefinition bit. We then form a field using MAX, base and bias and using it search the UD table for a match.

In the case of no match, we enter it with the appropriate load or store bit set. Then, we continue processing as if we had a match. We iterate through the number of subscripts in the macro and set load references for each of them making an entry when necessary. Then, we proceed to the next macro.

5.5 I/O Macros and APLISTS

5.5.1 Set R to Address in I/O List

When this occurs, we set MODE to reflect whether we are in an input or an output list. LOADS, LOAD CA and LOAD RF are set up and we enter normal symbol processing.

5.5.2 Initial I/O Macro

We extract the name of the external being called in the I/O macro from the symbol table. IO is set to one and changed to two if the name starts with an O (same output routine). Then, we set the external references flag to inhibit register assignment and proceed to the next macro.

5.5.3 Intermediate I/O Call

Set the external reference flag and go to the next macro.

5.5.4 Final I/O Call

Clear the I/O list flag and go to the next macro.

5.5.5 Special I/O Call

Set REDEF non-zero (means abandon all optimization) and RETURN.

5.5.6 APLISTS

If the APLIST macro is part of a RETURNS list, set REDEF non-zero and RETURN.

6.0 Table Formats

USE

```
        VFD   8/0,1/R,1/D,1/S,1/L,18/S1,18/CA1,12/H1
        VFD   8/0,1/R,1/D,1/S,1/L,18/S2,18/CA2,12,H2
                    .
                    .
                    .
        VFD   8/0,1/R,1/D,1/S,1/L,18/Sn,18/CAn,12/Hn
```

```
Hi  = base ordinal of symbol
CAi = bias of symbol
Si  = size - 1 of array; zero for variables
L   = on if used in the sequence
S   = on if defined in the sequence
D   = on if defined before used
R   = on if full redefinition
```

ADDROW

1.0 General Information

Task Description

This routine adds a row to the connection matrix and initializes two words of column space.

2.0 Entry Points

ADDROW

This is the only entry point. There are no parameters to the routine and exit is via a RETURN statement.

3.0 Diagnostics and Messages

None

4.0 Environment

Variables

LOC IN RT - common variable containing a dynamic subscript used in referencing the next available location in the row table (R TAB).

LOC IN C - local variable containing an integer which when added to the dynamic base of the column table (CTB) gives the address of the next available word pair in the column table.

5.0 Structure

Processing

If the current row number has already been added, we simply RETURN. Otherwise, the next entry in the row table is set to LOC IN C (the next word pair of the

column table). We increment the row table index and obtain another 100 words if the table is full. We initialize the two column table words to zero and increment LOC IN C by 2. If the column table is full, we obtain another 100 words. Then, we RETURN.

6.0 Table Formats

```
RTAB    VFD  42/0,18/a1
        VFD  42/0,18/a2
                .
                .
                .
        VFD  42/0,18/an
```

The ai represent integers which point to word pairs in the column table (CTAB) relative to the base of the table.

```
CTAB    VFD  6/0,18/C1,18C2,18/C3
        VFD  6/0,18/C4,18/C5,18/link
        VFD  6/0,18/C1,18/C2,18/C3
        VFD  6/0,18/C4,18/C5,18/link
                .
                .
                .
```

The Ci represents the numbers of columns which contain a non-zero entry. The column list for any one row is terminated by a zero field. If there are more than five non-zero entries the sixth field contains a link to another word pair where the column list is continued.

Example:

Connection matrix

	1	2	3	4	5	6	7	8
1	0	0	1	0	1	0	0	0
2	0	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	1	0	1	0	1	1	1	1
5	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

```
RTAB    VFD   42/0,18/1
        VFD   42/0,18/3
        VFD   42/0,18/0
        VFD   42/0,18/5
        VFD   42/0,18/9
        VFD   42/0,18/11
        VFD   42/0,18/13
        VFD   42/0,18/0

CTAB    VFD   6/0,18/3,18/5,18/0
        VFD   60/0
        VFD   6/0,18/2,18/0,18/0
        VFD   60/0
        VFD   6/0,18/1,18/3,18/5
        VFD   6/0,18/6,18/7,18/7
        VFD   6/0,18/8,18/0,18/0
        VFD   60/0
        VFD   6/0,18/8,18/0,18/0
        VFD   60/0
        VFD   6/0,18/7,18/0,18/0
        VFD   60/0
        VFD   6/0,18/8,18/0,18/0
        VFD   60/0
```

SETBIT

1.0 General Description

This routine sets the bit in the connection matrix at the location designated by ROW and COL.

2.0 Entry Point

SETBIT is the only entry point. The subroutine is called with no parameters and exit is via a RETURN.

3.0 Diagnostics and Messages

None.

4.0 Environment

Common Variables

ROW - row number in the matrix
COL - column number in the matrix
RTB - base of the row table (R TAB)
CTB - base of the column table (C TAB)

5.0 Processing

First, we scan each column entry for the row denoted by the contents of ROW. If we find that the column denoted by COL is already marked, we just RETURN. If the column is not marked, we make a new entry in the linked list. When all entries in a word pair (5 entries) have been used, we obtain another pair, link it to the previous one and make the required entry. Then, we RETURN.

6.0 Table Formats

See section 6.0 for ADDROW

TESTBIT

1.0 General Description

This logical function returns true if the connection matrix entry denoted by ROW and COL is set. Otherwise, false is returned.

2.0 Entry Points

TESTBIT is the only entry point. The function is called with a single formal parameter which is not used and exit is via a RETURN.

3.0 Diagnostics and Messages

None.

4.0 Environment

Common Variables

ROW - row number in the matrix
COL - column in the matrix
RTB - base of the row table
CTB - base of the column table

5.0 Processing

We scan the first word of the word pair comparing each of the three entries to COL for a match. A zero entry terminates the list and we RETURN with false. Then, we scan two entries in the second word pair using the same criteria. Then, if necessary, we link to the next word pair and perform the above scans again. If a match with COL is found, we RETURN with true.

6.0 Table Formats

See section 6.0 for ADDROW

PUTMS

1.0 General Description

This routine performs the necessary processing to ensure that a sequence is written to mass storage.

2.0 Entry Points

2.1 PUTMS

This is the primary entry point. The routine is called with two parameters: NUMBER which denotes the sequence number and LENGTH which holds the size of the sequence. Exit is via a RETURN.

2.2 DUMPMS

This entry point is used to force any sequences in memory onto storage. The parameters are the same as for PUTMS. Exit is via a RETURN.

3.0 Diagnostics and Messages

None

4.0 Environment

4.1 Common Variables

ABSB - start of the first sequence in memory
SEQB - start of the current sequence in memory
PEND - number plus one of the present ending sequence in memory or zero if none
CHANGE - set non-zero if we alter a sequence in core, indicates we must write to disk before retrieving another group of sequences

4.2 Local Variables

NSEQ - number of sequences in core
SEQLIM - maximum number of sequences to keep in core

5.0 Processing

5.1 PUTMS

First, we increment the number of sequences in core and compute the relative location of the current sequence. Then, we check to see if this is a put of an existing sequence or a new sequence. For an existing sequence, we clear the number of sequences counter and set CHANGE to the sequence number. A new sequence requires an entry in the index table with the prescence bit on, starting word of sequence in block, and sequence length.

When we have accumulated SEQLIM sequences or the block size exceeds half of working storage, we dump the memory block to mass storage. If conditions for dumping are not met, we move the current sequence base up to the next starting base (LOCINSQ), set the CHANGE flag and RETURN.

5.2 DUMPMS

At the entry, we simply return if CHANGE indicates no need to dump. Otherwise, compute the total block size and use WRITMS to dump the block. We then iterate backward through the index placing the PRU number in each entry with the prescence bit on and clear that bit. Before exiting, the number of sequences, the change flag, and the PEND cell are cleared and the current sequence base reset to the first sequence base (ABSB).

6.0 Table Formats

See section 6.0 for PASS15$

GETMS

1.0 General Description

This routine retrieves (from mass storage if necessary) the sequence specified and sets up the sequence base appropriately.

2.0 Entry Points

GETMS is the only entry point. The call has two parameters: NUMBER denotes the sequence to retrieve and the size is returned in LENGTH. Exit is via a RETURN.

3.0 Diagnostics and Messages

None.

4.0 Environment

Common Variables

ABSB - start of the first sequence in memory
SEQB - start of retrieved sequence in memory
PEND - sequence number plus one of last sequence or zero if none
CHANGE - set non-zero if a sequence has been altered

5.0 Processing

If the required sequence is in core (prescence bit on) we simply compute SEQB using the relative sequence start in the index plus the base of the first sequence. The length is also extracted and then we RETURN.

When the sequence is not in core, we must retrieve it from mass storage. Before doing this, we must dump to mass storage the block which is in memory, if it has been changed. After dumping, we insert the new PRU number and clear the prescence bit for each sequence in the block.

In order to retrieve the block, we must know its size. This can be computed using the last sequence in the block which we locate by scanning the index until the PRU number changes. We then call READMS and bring in the block. Before exiting to the logic to compute SEQB and length, we set residence bits for each sequence in the block just retrieved.

6.0 Table Formats

See section 6.0 for PASS15$

UPDOWN

1.0 General Information

This routine modifies the length of a sequence within a block and moves higher numbered sequences accordingly.

2.0 Entry Points

UPDOWN is the only entry point. The two parameters passed in the call are NUMBER which denotes the sequence number and LENGTH which specifies the amount to modify the sequence length.

3.0 Diagnostics and Messages

None

4.0 Environment

Common Variables

PEND - sequence number plus one of last sequence in memory

5.0 Processing

For a length change of zero, we RETURN. If it is a change for the last sequence in core, we simply adjust the length in the index and RETURN. Otherwise, we compute the start of the sequence in question, modify the starting locations of all higher numbered sequences in the block and compute the destination address. Finally, we compute the number of words to move and call ADJUST to do the move. Upon return, the length of the sequence is modified and we RETURN.

6.0 Table Formats

See section 6.0 for PASS15$

READR

1.0 General Information

These routines perform I/O interface activities for the FORTRAN routines.

2.0 Entry Points

2.1 READR

This entry returns a block of words from a file specified in the call. Parameters in the call are:

NWDS - number of words to read
FILE - file number to read from
RBUFF - area to place the information read

On exit, NWDS holds a count of the number of words requested minus the number received.

2.2 OPENMS

Opens the OPT file for random use. Parameters in the call are:

INDEX - start of the index area
LENGTH - size of the index area

2.3 READMS

This entry provides direct access with numbered index facility to the FORTRAN routines. Parameters in the call are:

LFN - file name, not used
ARRAY - area to read into
NWDS - number of words to read
NUMBER - record number to retrieve

2.4 WRITMS

This entry provides direct access with numbered index facility to the FORTRAN routines. Parameters in the call are:

LFN - file name, not used
ARRAY - area to write from
NWDS - number of words to write
NUMBER - record number to write

3.0 Diagnostics and Messages

OPT FILE MUST BE RANDOM ON MASS STORAGE

This error message is issued to the dayfile, and we abandon all OPT=2 processing by reverting to OPT=1 mode. This condition happens if the file FTNOPT is not on a device capable of direct access.

4.0 Environment

Not applicable

5.0 Processing

5.1 READR

On entry, we save A0 and A1. Then, we take parameters passed from FORTRAN and place them in appropriate registers. We use the compiler routine RDWDS to obtain a block of input R-list. Upon return, we restore the registers A0 and A1 and store the number of words requested minus the number read back into the first parameter. Then, we exit.

5.2 OPENMS

First, we extract the index address and length, combine them in a word and store it into word seven of the OPT FET. Next, we set the random bit in the FET so we can determine if the file is on a direct access device. After issuing an OPEN ALTER with no rewind we set the flag for a numbered index. If the random bit is still set, we just exit. Otherwise, the OPTLVL cell is set to

reflect OPT=1, a dayfile message is issued and we go to pass 2 via CLOPT.

5.3 READMS

Primarily, processing consists of taking the parameters as passed and inserting them into the OPT FET. We extract the appropriate index word and place the PRU number into the sixth word of the FET. The FIRST address is changed to point to the starting data address passed. IN and OUT are set to FIRST and LIMIT to FIRST + NWDS+1. Finally, we issue a READ SKIP with recall and exit.

5.4 WRITMS

The address of the index word in which to place the PRU number is placed into word six of the FET. Then, we modify the FET pointers to reflect the area about to be written out. Finally, an END-OF-RECORD WRITE with recall is made and we return to the caller.

6.0 Table Formats

None.

CHECK

1.0 General Information

This routine checks to see if a subscript exceeds the current bounds of its table. If it does, the size of the table is increased by a specified amount and control returned.

2.0 Entry Points

2.1 CHECK

This is the primary entry point and performs all subscript checking and table adjusting. Parameters are:

SUB - subscript value
TABLE - table name or number
INC - amount to increase table by, if necessary

2.2 ERRORC

Issues a core overflow message, sets the fatal error flag and exits to pass 2.

2.3 ADJUST

Sets up parameters from the FORTRAN call and moves core accordingly. Parameters are:

ORG - origin of the area to move
DEST - place to move it to
NWDS - number of words in the area

2.4 WSSIZE

Returns in X6 the size of working storage at the beginning of pass 1.5.

2.5 MAXCOR

Location containing the size of working storage at the beginning of pass 1.5.

3.0 Diagnostics and Messages

FATAL TO COMPILATION - TABLE OVERFLOW IN PASS 1.5

Issued when there is insufficient storage available to process the program.

4.0 Environment

Not applicable

5.0 Processing

Initially, a check is made on the second parameter to determine if it is a table name or number. If the low eighteen bits are non-zero, we have already located the information about the table. Otherwise, we scan the list of table names for a match. When a match is found, the address of the match plus one is stored into the second parameter. This prevents a search for the name on subsequent calls.

The low 18 bits of a word in the name table point to the first of a group of subscripts used in accessing that table. The first subscript is always the base of the table. Therefore, to verify the subscript, we pick up the first subscript of the next table (its lowest address is held there). If the subscript we are checking is less than that value, we simply exit.

If it is necessary to get more room, we add the increment to the subscripts of all tables above the current one. The list of subscripts is ended with a zero word. Next a check is made to determine if we have exceeded the bounds of working storage. If so, we go to the ERRORC entry. Otherwise, we setup origin destination and number of words and move tables upward.

If the table we increased was not the index table, we just exit. For the index table, we modify the index length in the FET and zero out the new index area before exiting.

6.0 Table Formats

TABS block

```
VFD  42/0L table name1, 18/A1
VFD  42/0L table name2, 18/A2
          .
          .
          .
VFD  42/0L table namen, 18/An
```

where Ai is the address in the block/BASE/where the table base and dynamic subscripts for table number n begin.

/BASE/block

```
VFD  60/base of table 1
VFD  60/subscript of table 1
VFD  60/base of table 2
VFD  60/subscript of table 2
VFD  60/subscript of table 2
          .
          .
          .
VFD  60/base of table n
VFD  60/subscript of table n
VFD  60/subscript of table n
VFD  60/subscript of table n
VFD  60/0
```

DBGPHCT

1.0 General Information

DBGPHCT is comprised of the COMPASS utility routines for PASS 1.4.

2.0 Entry Points

2.1 DBGIPKT

DBGIPKT is called by PH1CTL to control the processing of statements in the Internal Packet.

2.2 DBGINT

DBGINT is called by PH1CTL to process an interspersed Debug statement encountered during PHASE 1 processing.

DBGINTX is the alternate entry for DBGINT and is called by PH1CTL when an interspersed statement follows an unrecognized standard FORTRAN statement.

2.3 DBGEPKT

DBGEPKT is called by PH1CTL to control the processing of statements in the External Packet.

2.4 POINTRS

POINTRS is called by DBGEPKT, DBGIPKT, DBGINT, and PS1CTL to set up COMPASS pointers for use by Debug mode FORTRAN routines.

2.5 DMVWDS

DMVWDS is called by PASS 1.4 FORTRAN routines to provide the COMPASS linkage for a call to MOVE.

2.6 WRTMS1. This entry point opens the file.

2.7 WRTMS2. This entry point writes information out onto the file without adding an end of record. (This allows for multiple writes on the same record).

2.8 WRTMS3. This entry point puts an end of record marker on the file.

2.9 WRTMS4. This entry point closes the file.

2.10 RDMS1

RDMS1 is called by BUGPRO to read information from the Debug random file into Central Memory.

2.11 CFO

CFO checks the first occurrence of a name in a FORTRAN statement when the name has been previously mentioned in a Debug statement.

2.12 DBGCUN is called to issue the error message for illegal mention of a name in a Debug statement.

2.13 DSYMTAB

DSYMTAB is called by BUGCON to set up the COMPASS linkage for a call to SYMBOL.

2.14 FIXPNTR

FIXPNTR is called by SETARR, BUGPRO, and BUGCON to update COMPASS pointers to agree with their new FORTRAN representations.

2.15 DINPH2

DINPH2 initializes the Debug processor for Phase 2.

2.16 DCONV

DCONV enters a constant in the constant table for the Debug option processor.

3.0 Diagnostics and Messages

3.1 There are no fatal to compilation, fatal to execution, or non-ANSI diagnostics.

3.2 Informative

3.2.1 If there is a Debug deck before the program card or immediately following a program unit header card, which is not headed by C$ DEBUG card, the following message results:

'DEBUG STATEMENT MISSING FROM THE START OF PACKET.'

3.2.2 If there are not 200 words available for Debug processing at internal packet time, the following message results:

'MORE CORE NEEDED FOR DEBUG PROCESSING.'

3.2.3 If a Debug input file is specified on the FTN card (D=lfn; default is INPUT), and no Debug information is on that file, the following message results:

'NO DEBUG INFORMATION FOUND ON DEBUG FILE.'

3.2.4 When a loader directive occurs while processing an external packet, processing ceases even if there are further Debug cards and the following message results:

'LOADER DIRECTIVE TERMINATES EXTERNAL PACKET PROCESSING.'

3.2.5 If Debug cards occur between program units, the following message results:

'EXTERNAL PACKET ILLEGAL BETWEEN ROUTINES.'

3.2.6 If there are not 500 words available for setting up tables for packet processing, the following message results:

'MORE CORE NEEDED FOR DEBUG MODE.'

4.0 Environment

4.1 Lists and tables used by COMPASS utility routines and FORTRAN routines. Beginning and ending address relative to blank common, pointers to these lists, and tables are in common block /DBGBLK2/, which is explained in full under FORTRAN subroutine PUT.

4.1.1 Debug Routine List, D.SDRL to D.EDRL, with length LNGDRL.

4.1.2 Debug Variable List, D.SDVL to D.EDVL, with length LNGDVL.

4.1.3 Area List, D.SAREA to D.EAREA.

4.1.4 Fixed Area List, D.SFDIT to D.EFDIT-1.

4.1.5 Symbol Table, D.SSYMTB to D.ESYMTB.

4.1.6 ELIST Table, SELIST to D.ELAST; D.ELIST is the pointer to the E-list item currently being used.

4.1.7 Options List, beginning at D.OPL; D.NESTW points to the current options word; D.NEST points to the field within that word.

4.1.8 CONLIST, constant table for the statement currently being processed, D.CON1 to D.CONL.

4.1.9 Arrays and Stores Information (AASI), beginning at D.SAASI; D.NAASI points to the next available ordinal in this list.

4.1.10 Random Index, beginning at SDBGIND with length LNGIND.

4.1.11 Constant table, beginning at D.SCON1 and at D.STOR for BUGCON.

4.1.12 Reference Map Information, beginning at D.RFMAP.

4.1.13 The end of the DO information is D.DOLAST for the FORTRAN routines.

4.2 Common block /DBGBLK1/

ALLARR	.NE. 0 if option applies to all arrays
ALLCALL	.NE. 0 if option applies to all calls
ALLFUNC	.NE. 0 if option applies to all functions When .GT.0 these items represent frequency counts.
GOTOSFL	.NE. 0 if GOTOS options is on
NOGOFLG	.NE. 0 if no execution despite fatal error
TRACEL	current TRACE level
ALLROU	.NE. 0 if option list applies to all routines

SPIDER	Dummy routine name when no routine names are specified on the C$ DEBUG card in the external packet.

4.3 Common block /NONFTNX/

D.COL	.NE. 0 if external packet already processed
DTYPE	= 0 if next statement not Debug statement
D.NCURU	(2 words) UPDATE ID of next statement
D.NDUKE	binary line number of start of next statement
D.NLABEL	label of next statement
DBGRFMP	last used word of REFMAP or equivalent

4.4 Miscellaneous Internal Symbols

CONTYPE	constant type code
DBGFSTT	= 39, first statement type code of Debug statements
DBGEXTP	= 50, last statement type code of Debug statements + 1

5.0 Structure

5.1 DBGIPKT

5.1.1 Initialization

Clear the "inhibit execution if fatal error" flag and set packet flag to "no information". Initialize lengths of DRL, DVL, random index, index record number, and options list header word for the current routine if there are any Debug cards. If not, set flag to indicate this and go to end of packet processing. Set up tables via RJ DTABLE; set up pointers for the FORTRAN routines via RJ POINTRS; and adjust pointers affected by the insertion of Debug tables. If there is not enough room for the tables, close up the processing which has been started. Otherwise, set the flag to "there is packet information." The Debug information pertaining to this routine is brought in from disk via RJ BUGPRO.

5.1.2 Begin Processing

The flags to indicate internal packet processing and the options list word and field pointers are set up.

FWAWORK must be saved and restored at the end of processing. If the next card, is not a Debug card restore symbol table, E-list, and FWAWORK to original positions, set flag to "no Debug information" and go to end of packet processing. If it is, it is converted to E-list via RJ SCANNER and typed either then or by an additional RJ GETTYPE if necessary. If it is a bad Debug statement (type returned is 0), look at the next card. If the card is not a C$ DEBUG card, the error message "DEBUG STATEMENT MISSING FROM START OF PACKET' results and a C$ DEBUG card is processed and control flows to main processing.

5.1.3 Processing Debug Statements

The necessary pointers are set for the FORTRAN routines and the E-list of the statement is converted to table form via RJ BUGCON. FWAWORK is set to the end of the options list so far and reset to the end of the area list each time a new C$ AREA card occurs. This processing continues until a non-Debug statement is reached. A check is made to make sure there has been some Debug information. If not, control flows through the same path as when this is realized at beginning of processing time. Otherwise, a RJ BUGCLO flushes the information to disk, the symbol and E-list tables are moved back into place, FWAWORK is restored, and the flag is set to indicate that there has been packet information.

5.1.4 End of Internal Packet Processing

A check is made to see if there are 200 words available for interspersed processing. If not, the appropriate diagnostic is issued if it has not already been, and a flag set to indicate this condition (D.NOGO = -1). Otherwise, rearrange tables and pointers for the FORTRAN routine.

Because at this point the options list has either already been sent to disk or there is none to be sent, the D.OPFLG is set to indicate this. D.PACK and D.PADD are set to "interspersed processing." TYPFLAG status is set to "OK", and the random index length is updated.

Control returns to the calling routine.

5.2 DBGINT - Process Interspersed Statements

Under normal entry conditions, SCANNER converts the statement to E-list and if SCANNER has not typed the statement it is typed via RJ GETTYPE. If the alternate entry is made, return conditions are set up first. At this point, the statement is already in E-list and the check to see if it has been typed is made. If not, it is typed via RJ GETTYPE. If it is a bad Debug card, go get the next card.

The options list is built starting at D.OPL which is the end of the DO information. D.NESTW points to the word; D.NEST points to the field within that word. After all pointers are set for FORTRAN routines, a RJ BUGCON is made to put the Debug statement into table form. OFF statements are handled in BUGCON, and interspersed AREA and DEBUG cards are checked for syntax, but not processed. If the statement is any but one of three types there is a RJ TURNON which activates the options.

This process continues until a non-Debug statement occurs after which TYPFLAG reflects "O.K." status, and control returns to the caller.

5.3 DBGEPKT

5.3.1 Initialization

LNGDRL, LNGDVL, LNGIND, RECORD, DISPOW, ROUNAME, NOLIST flag and location of blank common are all set up for FORTRAN routines. The packet flag is set to indicate external packet processing. The bad statement flag is set to "not bad statement". If an external packet is not legal at this point, control goes to section 5.3.4. DTABLE is called to set up the tables for Debug mode, if there is enough room. If not, there is no further attempt to bring information in from disk. If there is room BUGPRO is called to bring any information in from disk and put it in the tables.

5.3.2 Begin Processing

SCANNER is called to put the first statement into E-list. If the statement has not been typed by SCANNER, GETTYPE is called to do this. If it is a bad Debug card, go back and try the next card. If it is not a Debug card, NXTMOVE is called to return E-list and

SYMTAB to their original positions. If there is already information on the disk, the random file is opened, the DRL and DVL are zeroed out to indicate no DRL and no DVL, and the file is closed. If the Debug file is the input file, end processing, since no further possibility exists. If not, then there is no information on the Debug file. The Debug file is then closed and the input file is opened. At this time, an external packet on the input file is still possible, so the procedure is repeated with Debug file = Input file.

5.3.3 Convert to Table Form

If a C$ DEBUG card is not the first card of the packet, the dummy routine SPIDER is entered in the DRL. The packet information will then apply to all routines.

After setting up the necessary pointers, BUGCON is called to convert the statement to table form. When an AREA or DEBUG card occurs, a new options list must be started and FWAWORK is set to the next position in the area list. Otherwise, it is set to the next position in the options list, and building of that list continues.

This process continues until a non-Debug card is sensed.

5.3.4 When External Packet is Illegal

When an External Packet is illegal, TYPFLAG is set to "not OK" status and restored to "OK" status upon occurrence of a program header card. D.NOGO is set to "do not process Debug statements" so that the packet can be checked for syntax, but no lists will be created.

5.4 DTABLE - Set up Tables for Debug packet processing

If DTABLE is called from the external packet processor X2 holds the address of the end of the symbol table; if from internal packet processor X2 holds the LWA of working storage. If there are not 500 words between the address in X2 and the start of blank common, the appropriate diagnostic is issued, the flag is set to indicate this, B6 set to 0, and control returns to the caller.

The symbol table is moved to below the DRL and DVL. Pointers for the DRL, DVL, AREA List, Symbol Table, the random index, and Refmap pointers are set up or adjusted

after the move and control returns to the caller. The Debug Tables are positioned between REFMAP and the symbol table.

5.5 NXTMOVE - Move SYMTAB and E-list to original position

The length of SYMTAB and E-list combined is LWA of working storage minus the start of SYMTAB. Together they are moved back up to the REFMAP. The words moved count is in D.ELAST. Their respective pointers are then adjusted to reflect the move.

5.6 POINTRS - Set up pointers to FORTRAN routine

The FORTRAN pointers are equivalent to their respective COMPASS values adjusted so that they are relative to blank common.

5.7 DMVWDS - Call MOVE for FORTRAN routines

It is through this routine that the macro MOVE can be utilized for FORTRAN routines. The calling sequence is parallel to COMPASS.

CALL DMVWDS (WORDCT, FWA, DESTADD)

In this routine, X1 is set to WORDCT, X2 to FWA, X3 to DESTADD, and the MOVE macro is executed.

5.8 WRTMS1 - Set up FET for Random File and open it

Routines 5.8 through 5.12 are analogous to their FORTRAN library counterparts.

5.9 WRTMS2 - Move words from an area list to Debug file buffer. Moves are not necessarily in complete record increments.

5.10 WRTMS3 - Terminate random record on Debug file

5.11 WRTMS4 - Close Debug random file

5.12 READMS1 - Get a record from the Debug random file

5.13 CFO - Checks to see if the first occurrence of a name is a FORTRAN statement and its use in a previous Debug statement are compatible

5.13.1 Entry Conditions

A0 - A2, X1, X2, X6, X7 are values as set by SYMBOL when the name first occurs.

X0 = 0 if it is a variable or array; X0 = 1 if it is external.

5.13.2 This routine is called from the statement processors. If the IF bit is on and the name is not a variable or array, DBGCUN is called to issue a message. Otherwise, return is made to the calling routine.

5.13.3 Exit Conditions

If there is no conflict, X2 will be word B of the symbol table entry plus the Debug bits. If there is a conflict, the Debug bits field is cleared and the registers remain the same as at entry time.

5.14 DBGCUN - Issue message for illegal use of name in a Debug statement

The P-field and natural type of name is saved before the symbol table entry is updated. ERPROI is called with the necessary information to print the informative diagnostic. Upon return, the registers are restored and symbol table entry updated. Control then returns to the calling routine.

5.15 DSYMTAB - Call SYMBOL for FORTRAN routines

The parameters are set up in the proper registers and SYMBOL is called. If it was the first occurrence in a Debug statement, the type is set to type Debug and the natural type is saved.

If the name is a legal type and has been entered in the symbol table, the ordinal is returned and the symbol table pointer is updated for the FTN routines.

5.16 FIXPNTR

This routine updates from the FORTRAN values, the values of their counterpart in the COMPASS routines.

COMPASS	FORTRAN (relative to / /)
SYM1	D.SSMTB
SYMEND	D.ESMTB
LWAWORK	D.ELAST
SELIST	D.ELIST
LELIST	D.LELST

5.17 DINPH2 - Initialize Debug processor for Phase 2

If there were any constants encountered during Phase 1, a constant table is set up and the constants are entered in it.

The name LABEL. is entered in the symbol table and control returns.

5.18 DCONV - Enter constant in the constant table for Debug mode

5.18.1 Entry Conditions

X1 holds LOCF, pointer to the start of the constant.

5.18.2

If the element passed to this routine is not a constant, X6 is set less than zero to indicate this and control returns. The same is true for a complex constant in illegal form.

5.18.3

For complex constants, the imaginary and real parts are both converted and saved. Type complex is stored in CONTYPE. For non-complex constant, the constant is converted, and the type, taken from the E-list is stored in CONTYPE. The word count for complex and double precision variable is 2, for others 1. Word count -1 is stored in WORDS.

When in Phase 1, the constant is entered in the Debug constant table if it is not already there. When in Phase 2, CONVERT is called again to enter it in CONTAB. If processing a packet, the constant is entered in the global table if it is not already there.

5.18.4 Upon exit, X6 is set to the following, where FLAG = 1 when the constant is entered in the global table:

42/0,3/TYPE/,1/FLAG,14/ORDINAL

5.19 ADDGCON - add a word to the global con table.

A check is made to see if there is enough space (space allocated - length). If there is, the constant passed in X1 is entered, and the length is updated.

If there is not room, an attempt is made to get space from the R-list buffer, in which case the affected pointers must be adjusted. If this fails an error message to indicate "Global Constant Table Overflow" is issued.

5.20 SRCH - search a con table for a match

5.20.1 Entry Conditions

X0 = FLAG (0 or 1514)
X1, X2 = Words 1 and 2 of constant
X4, X5 = FWA and length of con table being searched

5.20.2 A loop is set up beginning at the FWA, as indicated by X4, and ending when a match is found, or the end of the table is reached. In the case of a 2 word entry, the loop is repeated for the second word.

5.20.3 Exit Conditions

Match found
B2.LT.0;X1 = FLAG + ORDINAL
Else
B2.GE.0;TEMP+2 = FLAG + LENGTH OF TABLE

PUT

1.0 General Information

PUT searches a linked line or statement number bound list for a particular bound. If this particular bound is not found, it will be entered into the list upon request.

2.0 Entry Points

PUT is the only entry point in this routine.

2.1 PUT is called by BUGPRO to enter bounds into linked FM bound lists. This is done in order to set up the line and statement number FROM bound lists after the area list has been brought in from disk.

2.2 PUT is called by BUGACT to search for bounds in the FROM bound lists before a statement is processed to see it options should be turned on at that statement and in the TO lists after a statement to see if options should be turned off.

2.3 PUT is called by TURN ON after it activates the options designated by a FROM bound in order to enter the corresponding TO bound into a TO bound list.

3.0 Diagnostics and Messages

There are no diagnostics issued by this routine.

4.0 Environment

4.1 Common block/DBGBLK1/

ALLARR	.NE. 0, if option applies to all arrays
ALLCALL	.NE. 0, if option applies to all calls
ALLFUNC	.NE. 0, if option applies to all functions
GOTOSFL	.NE. 0, if GOTOS option is on
NOGOFLG	.NE. 0, if no execution if fatal errors, despite Debug mode

TRACEL	Current trace level
ALLROU	.NE. 0, if the options list applies to all routines
SPIDER	Dummy routine name when no routine names are specified on the C$ DEBUG card in the external packet

4.2 Common block/DBGBLK2/

4.2.1 The following symbols are used as list pointers in both FTNX and COMPASS Debug modifications.

SDRL	Beginning address of DRL relative to / /
EDRL	Next available ordinal of DRL
SDVL	Beginning address of DVL relative to / /
EDVL	Next available ordinal of DVL
SAREA	Beginning address of AREA list relative to / /
AREAEND	Next available ordinal of the AREA list
DFOPL	Beginning address of the options list relative to / /
SSYMTAB	Beginning address of SYMTAB relative / /
ESYMTAB	Next available ordinal of the symbol table
SAASI	Beginning address of AASI relative to / /
REFMAP	Beginning address of REFMAP or equivalent to / /
ELIST	Current E-list pointer relative to / /
LELIST	Start of E-list for true side of logical IF
ELAST	Ending address of E-list relative to / /
DFNESTW	Current address of the options list
FIDIT	Beginning address of the fixed AREA list relative to / /
EFIDIT	Last ordinal + 1 of fixed AREA list
DFCONI	Beginning address of CONLIST relative to / /
DFCONL	Last used word ordinal of CONLIST
NAASI	Next available ordinal of AASI
SDBGIND	Beginning address of random debug index relative to / /
CONSTOR	SCANNER's CONSTOR for BUGCON
DFSCONI	SCANNER's CONSTOR relative to / /
DOLAST	DOLAST for FORTRAN Debug Routine

4.2.2 The following symbols are used as flags for FTNX and COMPASS modifications.

DFNOGO	.NE. 0 if Debug statements not to be processed
DFPACK	-1, internal pkt.; 0, interspersed; +1, external pkt.
POW	previous options word
DFOPFLG	.NE. 0 if options have to be written out
DFON	0, if turning on options; .NE. 0, if turning off
NUMERR	number of errors in the area list
PADD	0, if interspersed (for TURNON)
FEFLAG	.NE. 0 causes list of FE statement in NOLIST
NOLSFLG	0, if NOLIST option is on
OPENFL	1, if Debug unit is open; 0 if closed
DBGPROG	location 56 for FTN Debug routines
LDEBUG	location of DEBUG (blank common)
RECORD	next available record number
DISPOW	display for options list header word
SUCCESS	0, if external packet not on disk
LTFLAG	.TRUE. if processing on OFF statement
LFLAG	.TRUE. if the OFF statement is interspersed

4.2.3 The following symbols are used to pass information between FTNX and COMPASS Debug routines.

CUR UP DT(2)	Update ID of the current statement
DUKE1	Binary line number of start of current statement
CLABEL	Label of current statement
DFTYPE	Statement type
DFNEST	Current position with DFNESTW
LNGIND	Length of the index of Debug Random File
NO ACT	0, if no packet information; .LT. 0 else
AREAFLG	AREA list flag
D LNG DRL	Length allotted to the DRL
D LNG DVL	Length allotted to the DVL
C F PACK	E, if external packet; I, if internal
UP DT TBL(20)	Table for UPDATE ID's of comment cards
COUNTUP	Number of entries in UPDTTBL
SCNUPDT	.NE. 0 if PUTUPDT entered from SCANNER
FSTCOM	Line number of first comment card; 0, else
INDEXNO	1, if no internal packet information; 3, else

PHSFLAG	.LT.0 if between Phase 1 and Phase 2

4.3 Blank Common

DEBUG is an array beginning with the first location in blank common. It enables FTN Debug routines to reference any program location within the program field length.

4.4 Common blank/DBGBLK3/

NEXT	list pointer varies with various routines
TOFM	0, if a TO list; 1, if a FROM list to be searched
ITEM	line or statement number, right justified for PUT
BDADD	ordinal of bound to be added to area list, 0 if none to
MINZERO	.EQ. 7777B, end of options list indicator
MULT	pointer to field within the options list word
OH	contents of the field in the options list word
AASI	current word in ARRAYS and STORES information table
SYMTAB	used to look ahead in symbol table
FMLIST	ordinals of beginning of FROM bound and UPDATE bounds list
NFIELD	0, if FROM bound; .NE.0 if a TO bound
ASHIFT	indicates for which reason SETARR is called
DFOPL1	saves linkage information from AASI
DFOPL2	saves linkage information from AASI
TEMP	used by several routines as temporary storage for various items
SASI	saves the next operation a variable can be involved in
NXT STR	saves SASI
POINT	ELIST op code for .
TEMP1	used by several routines as temporary storage
ELIST1	saves E-list of the FROM bound
SAV UP DT(2)	temporary storage for CUR UP DT in BUGPRO
TRCADD	word containing ordinals of start of TRACE information
ERRNUM	error number from B-Cell in FIDIT
DLINE	line number from B-Cell in FIDIT

DTYPE	0, if next statement is not Debug statement
ERR MSG(5,4)	text of messages issued by BUGSOUT
NDRL	used to update DRL pointers when BUGCLO moves things
DRL	current ordinal in DRL
OPREC	record number of current options list
M	Debug error number for DEBUGER
SPIDIS	0, if "SPIDER" is in DRL, options apply to all routines
D UNIT	Debug's output file DEBUG
OPDRL	ordinal of the current options list in the DRL
STOSWAP	-1, if stores without relational operators was entered in the options list for this STORES statement 1, if it was stores with relational operators 0, if neither has yet been entered
MISFLAG	1, if at least 1 legal bound exists; 0, if none; -1 if illegal
CUROPBD	.NE.0 while processing a bound
CUR ROUT	routine whose bounds are currently being processed
COW	current options word
FMBOUND	0, if a line no. bound; 1, if statement no.; 2, if UPDATE identifier
TO BOUND	0, if a line no. bound; 1, if statement no.; 2, if UPDATE identifier
BLNG	number of words in the bounds list
DVL	current DVL ordinal
REL OP	relational operator code for options list
NO CONST	ordinal of a constant for options list
LEVEL	trace level
NO OF WDS	number of record sent to disk by BUGCLO
NXT ADD	address of last header word for this routine
ALNG	current length of lists moved to disk
NXTADD1	used to look ahead in header words
LSHIFT	shift factor for TURNOFF depending on interspersed or packet
NXTSTR1	saves NXTSTR while scanning an options list
LEVSHFT	shift factor for TURNOFF depending on interspersed or packet
TAASI	temporary storage for AASI in TURNOFF
TRACED	gets next TRACE level
NOLIN	0, if words being taken from AASI space,

	-1, if from elsewhere
AASIADD	beginning of linked list of freed words
RTAB(9)	relational operator code
NXT ITEM	-1, if TO bound to be added to list; 0, if none; 1 if FROM bound
TO LIST	beginning of line and statement number TO bound linked lists
DBGPIC(9)	E-list for the statement identifiers of the Debug statements
PREDBG(2)	text for prefix to Debug syntax error message
STAR1	.NE.0 if FROM bound is *
STAR2	.NE.0 if TO bound is *
SAMLINE	.NE.0 if FROM bound = TO bound
FSTLINE	to indicate first line when FROM bound is *
LSTLINE	to indicate last line when TO bound is *
STAR	E-list op code for *
COMMA	E-list op code for ,
RPAREN	E-list op code for)
LPAREN	E-list op code for (
SLASH	E-list op code for /
BOUND WD	current bound word
OP WORD	current option word
TLNG	total number of contiguous words sent to disk
BASE ADD	saves SAREA while BUGPRO moves SYMTAB and E-list
FT ADD	counter for words being brought in from AREA list on disk
BDS ADD	used by BUGPRO to set up bounds linked lists
TOT LNG	total length of information being brought in from random file
UP NO	used to calculate UPDATE identifier from its display code
UP DIS	used to calcualte UPDATE identifier from its display code
SYMTAB B	word B of a given symbol table ordinal
SAVE OH	temporary storage for OH
OVRFLOW	.NE.0 if there is constant table overflow
DLFLAG	used while scanning DRL when a new AREA statement occurs
DFSAVE	current node when searching a tree in BUGCON
DFSAVE 1	previous node when searching a

	tree in BUGCON
RELFLG	.NE.0 if the operator is a validity operator
ROU NAME	routine name whose packet information is being brought in
TO SHIFT	= 20 if line number TO list; = 40 if statement number TO list
VALUE	holds returned value of a function
INDEF	E-list representation of 'INDEF'
RANGE	E-list representation of 'RANGE'

4.5 The following conditions are expected upon entry:

NEXT = ordinal (relative to the base of the AREA list) of the beginning of the list to be searched

ITEM = the line or statement number searched for

TO FM = 0 if bounds in TO list

= 1 if bounds in FROM list

BD ADD = the ordinal (relative to the base of the AREA list) of the bound if it is to be added

= 0 if the bound not to be added

4.6 The following global conditions are expected upon entry:

SAREA = the ordinal (relative to the base of the DEBUG) of the beginning of the area list

4.7 The following conditions are changed upon exit:

NEXT = the ordinal (relative to the base of the AREA list) of the bound searched for, if found

= 0 if not found

5.0 Structure

When PUT is called, it is first determined from TOFM whether the bound is a TO or FROM bound, and the

appropriate field of the area list is masked out. This field is compared to ITEM, which is the bound being searched for or being placed in the list.

If ITEM is larger, the next bound compared is taken from the .GT. node. If ITEM is smaller, the next bound is taken from the .LT. node. If there is nothing in the .GT. or .LT. node, the current bound is entered there. When a match is found (ITEM = appropriate field of bound being searched), a check is made to see if it should be added. If not, return is made; otherwise, the bound address is entered the .EQ. node of the next available ordinal.

PUTUPDT

1.0 General Information

PUTUPDT searches the linked UPDATE Identifier bound list for a particular bound. If this particular bound is not found, it will be entered into the list upon request. PUTUPDT also ensures that the identifier is of the form:

.nnnnn where $0 \leq n \leq 9$

Leading blanks are converted to 0, but embedded blanks are not allowed.

2.0 Entry Points

PUTUPDT is the only entry point for this routine.

2.1 PUTUPDT is called by BUGPRO to enter bounds into the UPDATE identifier bound list. This is done to set up the UPDATE identifier bound list after the area list has been brought in from disk and all UPDATE identifier bounds, both FROM and TO, are entered into the list.

2.2 PUTUPDT is called by BUGACT to search for a FROM bound in the list before a statement is processed to see if options should be turned on at that statement and for a TO bound after a statement is processed to see if options should be turned off.

2.3 PUTUPDT is called by SCANNER to search for the UPDATE identifier of any comment card in the routine which appears between two executable statements. If the identifier is found, it is saved in the array UPDTTBL (providing there is enough room in the array; a maximum of ten identifiers may be saved) for later use by BUGPRO.

3.0 Diagnostics and Messages

There are no diagnostics issued by this routine.

4.0 Environment

4.1 The following local conditions are expected upon entry:

NEXT	=	the ordinal (relative to the base of the AREA list) of the beginning of the UPDATE identifier bound list, left justified in bits 59-41
CUR UP DT(1)	=	the name portion of the identifier being searched for
CUR UPDT(2)	=	the numeric portion of the identifier being searched for
NXT ITEM	=	-1 if TO bound to be added to list
	=	0 if no bound to be added to list
	=	1 if FROM bound to be added to list
BD ADD	=	the ordinal (relative to the base of the AREA list) of the bound to be added to the list
SCNUPDT	=	1 if routine called by SCANNER
	=	0 else
COUNTUP	=	next available location in UPDTTBL

4.2 The following global conditions are expected upon entry:

SAREA	=	the ordinal (relative to the base of DEBUG) of the beginning of the area list
FM LIST	=	the word containing the ordinals (relative to the base of the AREA list) of the beginning of the line number, statement number and UPDATE identifier FROM bound linked lists

4.3 The following conditions are changed upon exit:

NEXT	=	the ordinal (relative to the base of the list) of the identifier, if found
	=	0 if not found
N FIELD	=	0 if FROM bound was searched for
	≠	0 if TO bound was searched for
SCNUPDT	=	-1 if identifier found but no room for it in UP DT TBL

5.0 Structure

5.1 When called from SCANNER, the list beginning at the address in the upper 18 bits of NEXT is searched for UPDATE identifier stored in CURUPDT(1) and CURUPDT(2). If a match is found, the identifier is saved in the table UPDTTBL. This table saves up to 10 identifiers. SCNUPDT is set to -1 upon return to SCANNER if there is no room left in the table.

5.2 When called from BUGACT (interspersed mode), the tree structure of the area list is searched for a match using the procedure described in PUT. The nodes for the TO bounds are in complemented form. NEXT returns the ordinal, if found.

5.3 When called from BUGPRO, the given bound, as indicated by BD ADD, is entered in the list.

BUGACT

1.0 General Information

BUGACT is called by PS1CTL to ensure that packet options are activated at their respective FROM bounds and similarly deactivated at their respective TO bounds.

2.0 Entry Points

2.1 DFON = 1

BUGACT is called by PS1CTL before a statement is processed. The FROM bound linked lists are searched for all possible matches and TURN ON is called for every match found.

2.2 DFON ≠ 0

BUGACT is called by PS1CTL after a statement has been processed. The TO bound linked lists are searched for all possible matches and TURN OFF is called for every match found.

3.0 Diagnostics

No diagnostics are issued by this routine.

4.0 Environment

4.1 The following local conditions are expected on entry:

DFON	=	value set as described above
CUR UP DT (1)	=	the name portion of the UPDATE identifier of the first line of the statement being processed
CUR UP DT (2)	=	the numeric portion of the UPDATE identifier of the first line of the statement being processed

DUKE1 = the line number of the first line of the statement being processed

CLABEL = the statement label on the first line of the statement being processed (the label is left adjusted with blank fill in the lower 30 bits of the word)

4.2 The following global conditions are expected upon entry:

FM LIST = the word containing the ordinals (relative to the base of the AREA list) of the beginning of the FROM bound linked lists (the ordinals are positioned below)

```
VFD  2/0,18/Line no.,2/0,18/Statement no.,2/0,
     18/Update no.
```

TO LIST = the word containing the ordinals (relative to the base of the AREA list) of the beginning of the line and statement number TO bound linked lists (the ordinals are positioned below)

```
VFD  2/0,18/Line no.,2/0,18/Statement no.,20/0
```

SAREA = the ordinal (relative to the base of DEBUG) of the beginning of the AREA list

5.0 Structure

BUGACT calls PUT and PUTUPDT to search bound linked lists for matches with the line number, statement number, and UPDATE identifier of a given statement. For every match found, BUGACT calls either TURN ON to activate options or TURN OFF to deactivate options at that statement. The choice of bound linked lists to search and the choice of calling either TURN ON or TURN OFF depend on the entry parameter DFON.

GETOUT

1.0 General Information

Given the position of a 12-bit field in the Options list, GETOUT fetches the next successive 12-bit field.

GETOUT is called to return either

1) an option

2) an option parameter ordinal

3) end of parameter list indicator (0000B)

4) end of options list indicator (7777B)

2.0 Entry Points

2.1 GETOUT is the only entry point for this routine. GETOUT is called by TURNON and TURNOFF. It enables TURNON to traverse the options list and turn on options; and TURNOFF, to turn off options.

2.2 GETOUT is called by BUGPRO after the options lists have been brought in from disk in order to obtain the Debug Variable List (DVL) ordinals of option parameters and replace them with symbol table ordinals.

3.0 Diagnostics and Messages

There are no diagnostics issued by this routine.

4.0 Environment

4.1 The following local conditions are expected upon entry:

DFOPL = the ordinal (relative to the base of DEBUG) of the options list word containing the given 12-bit option field

MULT = the field position in DFOPL of the given field

4.2 The following conditions are changed upon exit:

DFOPL = the ordinal (relative to the base of DEBUG) of the options list word containing the new 12-bit option field

MULT = the field position in DFOPL of the new field

OH = the contents of the new field

5.0 Structure

Get the Contents of the Indicated Field

GET is called with DFOPL pointing to an option word of the form

VFD 12/OH,12/OH,12/OH,12/OH,12/OH

where OH = option code
DVL ordinal
end of option indicator (0000)
end of option list indicator (7777)

MULT points to one of the fields as indicated. When MULT is less than 0, DFOPL points to the next word and MULT is set to 4.

The options word is then shifted left by MULT * 6 which positions the desired field in the lower 12 bits. These 12 bits are masked off and stored into OH.

TURN ON

1.0 General Information

Given an options list, TURN ON activates all legal options in the list. It traverses the list and either sets bits in the symbol table or increases the frequency count in global option flags as each legal option is encountered. An error message is produced for any illegal options found (i.e., CALLS option asked for on a dimensioned variable).

2.0 Entry Points

2.1 TURN ON is the only entry point for this routine. TURN ON is called by BUGACT whenever the line number, statement number, or UPDATE Identifier associated with the next statement to be processed is found in a corresponding FROM bound linked list.

2.2 TURN ON is called by BUGPRO upon the completion of its area and options list processing whenever the line number of a declarative statement is found in the line number FROM bound linked list.

2.3 TURN ON is called by BUGCON during OFF statement processing in order to reactivate the packet options which had been deactivated by the OFF.

2.4 TURN ON is called by DBGPHCT and PS1CTL during interspersed statement processing in order to activate the options set forth in the statement.

3.0 Diagnostics and Messages

3.1 There are no fatal to compilation, fatal to execution, or non-ANSI diagnostics issued by this routine.

3.2 Informative

3.2.1 When trying to turn on the ARRAYS option for a non-dimensioned, non-DEBUG type variable, the following diagnostic results:

"AT THIS POINT THE ARRAYS OPTION IS ILLEGAL FOR..."

3.2.2 When trying to turn on the CALLS option for anything which is not a subroutine, or for the subroutine currently being compiled, the following diagnostic results:

"AT THIS POINT THE CALLS OPTION IS ILLEGAL FOR ..."

3.2.3 When trying to turn on the FUNCS option for anything which is not a function, or for the function currently being compiled, the following diagnostic results:

"AT THIS POINT THE FUNCS OPTION IS ILLEGAL FOR..."

3.2.4 When trying to turn on the STORES option with or without relational operators for anything which cannot be stored into, one of the following diagnostics results:

"AT THIS POINT THE STORES OPTION IS ILLEGAL FOR..."

or

"AT THIS POINT THE STORES OPTION WITH RELATIONAL OPERATORS IS ILLEGAL FOR..."

3.2.5 If there has been no TRACE information saved in the AASI when trying to turn on the TRACE option, the following message results:

"AT THIS POINT THE TRACE OPTION IS ILLEGAL"

4.0 Environment

4.1 The following local conditions are expected upon entry:

DFOPL = the ordinal (relative to the base of DEBUG) of the beginning of the options list

P ADD = 1 if the options list is associated with a packet

= 0 if the options list is associated with an interspersed statement

NEXT = the ordinal (relative to the base of the area list) of the FROM bound associated with the options list.

4.2 The following global conditions are expected upon entry:

SAREA = the ordinal (relative to the base of DEBUG) of the beginning of the area list

SAASI = the ordinal (relative to the base of DEBUG) of the beginning of the Arrays And Stores Information list

ALLARR = the current frequency count for the all arrays checked option

ALLCALL = the current frequency count for the all CALLS checked option

ALLFUNC = the current frequency count for the all FUNCTIONS checked option

GO TOSFL = the current frequency count for the GO TOS checked option

NOGOFLG = the current status of the execute if Fatal Errors option

(> 0 means no execution if Fatal Errors)

TRACEL = the current TRACE option level

TRCADD = the word containing the ordinals (relative to the base of AASI) of the beginning of the packet and interspersed TRACE option information lists

TO LIST = the word containing the ordinals (relative to the base of the area list) of the beginnings of the line and statement number TO bound linked lists

SSYMTAB = the ordinal (relative to the base of DEBUG) of the start of the Symbol Table.

The following global conditions are updated upon exit:

ALLARR, ALLCALL, ALLFUNC, GOTOSFL, NOGOFLG, TRACEL, TO LIST.

5.0 Structure

5.1 Symbol Table bits affected

COUNT (bits 30-19)	ordinal of AASI in DYDIT if NOT = 0 and IF = 0
NOT (bit 31)	0, if a Debug option is not on for this variable
IF (bit 32)	0, if ARRAYS or STORES or both are on for this variable. 1, if either CALLS or FUNCS is on for this variable
SF (bit 33)	1, if either STORES or FUNCS is on for this variable
AC (bit 34)	1, if either ARRAYS or CALLS is on for this variable

5.2 Since when called from BUGACT and BUGPRO options lists derived from packets are involved, a check is made to see whether the given FROM bound has not been preceded by its TO bound. If it has not, the options list is processed and the "No TO Bound" error status is placed into the associated B-Cell. If it has, the "TO Bound occurred before FROM bound" error status is placed into the B-Cell and the list is not processed.

5.3 Processing the List

MINZERO (7777B) indicates end of options list. A zero options field indicates end of option, and processing continues until all options for this options list have been processed.

5.3.1 ARRAYS, CALLS, and FUNCS

If the next options field is zero or MINZERO, it means the option applies to all arrays, all calls, or all functions, depending on the option code, and ALLARR, ALLCALL, or ALLFUNC, respectively, is incremented by one. If the field is MINZERO, control goes to end of list processing. Otherwise, processing of options continues.

If the next field is not zero or MINZERO, it is 2 times the symbol table ordinal to which the option applies. Before turning an option on, checks are made to be sure that an option is valid for a given variable.

5.3.2 GOTOS

To turn on the GOTOS option, GOTOSFL is incremented by one.

5.3.3 NOGO

TO activate the NOGO option, NOGOFLG is set to -1. This option cannot be deactivated.

5.3.4 STORES

5.3.4.1 STORES without Relational Operators

IF the STORES option is legal for the indicated symbol table entry, the option is turned on.

5.3.4.2 STORES with Relational Operators

Again, a check is made for the validity of STORES with relational operators for this symbol table entry, and if valid, the bits are set. The option word containing the STORES with relational operators code is followed by a word containing the information for object time processing in the following form:

ORDINAL	twice the symbol table ordinal for which the option applies
bits 47-45	type
bit 44	global flag (1, if in global table)
bit 43	constant or variable flag, 1 if variable on right side of expression
bits 42-30	ordinal of constant or variable
RO	code for relational or checking operator 0 indefinite 1 less than 2 less than or equal 3 not equal 4 equal 5 greater than or equal 6 greater than 7 out of range 8 valid (0 or 7)
FC	frequency count for the relational expression
LINK	address of the next relational expression involving the variable

The pointers must then be adjusted so that the next expected option is taken from the next word, not the word containing the information for STORES with relational operators.

5.3.5 TRACE

If the indicated TRACE level (default = 0) is greater than the current TRACE level, the new value is stored. Otherwise, the current level remains.

5.3.6 If turning on options due to packet specifications, the "NO TO BOUND" status is set in the B-Cell and the TO bound is put in the TO list if it is a line or statement number.

BUGSOUT

1.0 General Information

BUGSOUT is called by PS1CTL after all of the statements in a routine have been processed. BUGSOUT traverses the area list and checks the status of the range of all bounds pairs. An appropriate diagnostic message is printed out for each bounds pair whose range satisfies one of the following conditions:

a) lies completely outside of the range of the routine

b) does not terminate within the range of the routine

c) terminates but does not begin within the range of the routine

d) is inverted with respect to the routine

NOTE: Only the options associated with (b) are ever activated.

2.0 Entry Points

BUGSOUT is the only entry point for this routine, and is called by PS1CTL during termination of Phase 2 processing.

3.0 Diagnostics

3.1 There are no fatal to compilation, fatal to execution, or non-ANSI diagnostics issued by this routine.

3.2 Informative

3.2.1 If a bound satisfies condition (a), the message printed out is:

"AT DEBUG LINE (number) (E or I) - ((FROM bound), (TO bound)) BOTH BOUNDS OUT OF RANGE OF ROUTINE"

3.2.2 If a bound satisfies condition (b), the message printed out is:

"AT DEBUG LINE (Number) (E or I) - ((FROM bound), (TO bound)) TO BOUND OUT OF RANGE OF ROUTINE"

3.2.3 If a bound satisfies condition (c), the message printed out is:

"AT DEBUG LINE (number) (E or I) - ((FROM bound), (TO bound)) FROM BOUND OUT OF RANGE OF ROUTINE"

3.2.4 If a bound satisfies condition (d), the message printed out is:

"AT DEBUG LINE (number) (E or I) - ((FROM bound), (TO bound)) TO BOUND OCCURS IN ROUTINE BEFORE FROM BOUND"

where:

number = packet line number

E or I = External or Internal packet,

4.0 Environment

4.1 The following conditions are expected upon entry:

FIDIT = the ordinal (relative to the base of DEBUG) of the start of the fixed area list (brought in by BUGPRO with the appearance of the first executable statement)

EFIDIT = the ordinal (relative to the base of DEBUG) +1 of the end of the fixed area list.

4.2 The following conditions are changed upon exit:

FIDIT = EFIDET

NUMERR = the number of bounds satisfying one of the conditions (a) through (d) above.

5.0 Structure

The error field is masked out of the B-Cell. If an error exists, the information necessary to issue the diagnostic (packet type, line number, and type of bound) is masked out, and the diagnostic indicated by the code in the error field is issued.

The area list is traversed until all B-Cells have been checked.

BUGCON

1.0 General Information

BUGCON is called to extract DEBUG information from the type and E-list representation of a DEBUG statement and then use this information in the construction and manipulation of DEBUG lists.

2.0 Entry Points

2.1 BUGCON

2.1.1 BUGCON is called by DBGEPKT for every statement in the external DEBUG packet.

2.1.2 BUGCON is called by DBGIPKT for every statement in the internal DEBUG packet.

2.1.3 BUGCON is called by DBGINT and PSICTL for every interspersed DEBUG statement.

2.2 DMVCON

BUGCON is entered through DMVCON when called from BUGCLO in the case that BUGCLO must have the Debug tables moved.

3.0 Diagnostics and Messages

3.1 There are no fatal to compilation, fatal to execution, or non-ANSI diagnostics issued by this routine.

3.2 Informative

3.2.1 If it is found that there is no room to finish processing a packet, after processing has already begun, the following message results:

"DEBUG: MORE CORE IS NEEDED, FURTHER PACKET DEBUG STATEMENTS WILL BE CHECKED FOR SYNTAX BUT NOT PROCESSED".

3.2.2 When there are no more options list numbers, the following diagnostic results:

"DEBUG: NO MORE OPTIONS LIST NUMBERS"

3.2.3 If BUGCLO has been called to release space, cannot release any and returns through the alternate exit, BUGCON issues the following message:

"DEBUG: MORE CORE IS NEEDED FOR DEBUG PROCESSING".

3.2.4 Errors in BUGCON are diagnosed via DEBUGER, depending on the error code "M".

4.0 Environment

4.1 The following conditions are expected upon entry:

DFTYPE = the type of the DEBUG statement

ELIST = the ordinal (relative to the base of DEBUG) of the start of E-list for the statement

DFPACK = 1 if the statement is in the external packet
= -1 if the statement is in the internal packet
= 0 if the statement is interspersed

4.2 The following conditions are expected on entry and are updated upon exit:

SDRL = the ordinal (relative to the base of DEBUG) of the start of the Debug Routine List

EDRL = the ordinal (relative to the base of DEBUG) of the end of the Debug Routine List

SDVL = the ordinal (relative to the base of DEBUG) of the start of the Debug Variable List

EDVL = the ordinal (relative to the base of DEBUG) of the end of the Debug Variable List

AREAEND = the ordinal (relative to the base of DEBUG) of the end of the area list

DFNESTW = the ordinal (relative to the base of DEBUG) of the word currently being constructed in the options list.

DFNEST = the current available 12-bit field position in DFNESTW

REFMAP = the ordinal (relative to the base of DEBUG) of the beginning of REFMAP information

DBGPROG = the ordinal (relative to the base of DEBUG) of the word containing the name of the routine currently being compiled

D LNG DRL = the global length of the Debug Routine List

D LNG DVL = the global length of the Debug Variable List

ELAST = the ordinal (relative to the base of DEBUG) of the last word of E-list

SSYMTAB = the ordinal (relative to the base of DEBUG) of the start of the Symbol table

ESYMTAB	=	the ordinal (relative to the base of DEBUG) of the end of the Symbol table
DFNOGO	=	0 if enough room is available for lists
	=	-1 if not enough room is available
MISFLAG	=	1 if at least one legal bounds pair exists in an area statement
	=	0 if no bounds pairs exist
	=	-1 if only illegal bounds pairs exist
DFOPFLG	=	0 if no options lists is sitting in core
	=	-1 if an options list is in core and must be sent out to disk
AREAFLG	=	0 if a DEBUG statement was the first statement following a group of options statements
	=	1 if it was an AREA statement
ALL ROU	=	0 if a DEBUG statement has parameters
	=	1 if a DEBUG statement has no parameters (means no restrictions on routines mentioned in an AREA statement)
CUR ROUT	=	name of routine whose bounds are currently being processed
	=	0 if no bounds are being processed
POW	=	the number of the options list being processed
DISPOW	=	POW in display code
DFSCON1	=	the ordinal (relative to the base of DEBUG) of the beginning of SCANNER's constant table
CONSTOR	=	the actual address of the beginning of SCANNER's constant table
RECORD	=	the last used record on the Debug random file

5.0 Structure

The list construction and manipulation tasks are described in their order of appearance in BUGCON.

5.1 Processing the Debug Statement DEBUG

5.1.1 External Packet

All routines mentioned in the parameter list are entered into the DRL and designated as active. If no parameters appear, ALLROU is set to 1, the dummy routine SPIDER is placed into the DRL (if it's not already in), and all routines whose area lists are still in core are activated. (NOTE: A routine is activated by setting bit 59 of the second word of its DRL entry.)

Bounds and options associated with SPIDER will be applied to every routine in the job being compiled.

5.1.2 Internal Packet

If no parameters appear, the routine name is entered into the DRL and activated. Only the routine name is legal in a parameter list for the DEBUG statement. If any other name appears, an appropriate diagnostic message will be printed out.

5.1.3 Interspersed

The statement* will be checked for syntax but no list processing will be done.

5.2 Sending Options to Disk

Either a DEBUG or an AREA statement can terminate the information to be entered into an options list. Therefore, each time a DEBUG or an AREA statement is sent to BUGCON, DFOPFLG is checked to see if there is an options list in core. If so, the list is terminated with an end of options list 12-bit field (7777B) and the list is sent to disk.

5.3 Processing the Debug Statement AREA

5.3.1 Routine Name in Statement

For the External Packet, the DRL is searched for the activated entry of the routine name appearing in the statement. If ALLROU is set and the routine name is not found in the DRL, the name is entered amd activated. If ALLROU is not set and the routine name is not found or no routine name is in the statement an appropriate diagnostic message is printed out. (CUR ROUT = the routine name, if found).

For the Internal Packet, if the routine name appearing in the statement is not the name of the routine, an appropriate diagnostic message is printed out. If no routine name appears in the statement, the routine name is assumed to be the name intended. (CUR ROUT = the name of the routine)

For an interspersed statement, either Internal or External Packet syntax is correct since no list processing takes place.

5.3.2 Set up Options List

If this is the first AREA statement in a group of one or more AREA statements, DISPOW is set up for the associated options list. A diagnostic is issued if no more numbers exist.

5.3.3 Bounds Syntax

The E-list representation of a bounds pair is checked for correct syntax. If the syntax is incorrect, the appropriate diagnostic is issued.

5.3.4 Header or Options Word

A header or an options word or both are set up for the routine name found in the DRL.

5.3.5 B-cell

The syntactically correct bounds pair is put into B-cell form if enough room is available in the area list. If not, the appropriate diagnostic is issued.

5.4 Options Statement

If this is the first options statement in a group of one or more, AREAFLG is checked to see if an AREA statement appeared after the last DEBUG statement. If not, a dummy AREA statement with the bounds pair (*,*) is provided for all activated routines in the DRL. (If ALL ROU is set, dummy AREA is provided only for SPIDER).

After the AREAFLG check, a record number is obtained for the upcoming options list and placed into the Area List entry for all associated B-cells.

5.5 ARRAYS, CALLS, or FUNCS Options

Since the ARRAYS, CALLS, and FUNCS options are in the same form, they are processed with the same code. The options information is entered into the options list.

5.6 GO TOS

The GO TOS option information is placed into the options list.

5.7 NOGO

The NOGO information is placed into the options list. If the option is in the Internal Packet of the routine, the option is turned on.

5.8 STORES

The STORES option information is placed into the options list. This is where the differentiation is made between STORES with and STORES without Relational Operator. The differentiation is based on the E-list representation of the parameter list of the statement.

5.9 TRACE

The TRACE option information is placed into the options list.

5.10 OFF

If the statement is interspersed, all options mentioned in the parameter list are turned off. If there is no parameter list, all options are turned off and then the TO bound linked lists are traversed in order to turn on those options derived from packet statements which should still be on.

NOTE: Each E-list representation is scanned for information until either an End of Statement (EOS) or syntax error is reached. Since information is placed into a list as soon as it is scanned and found syntactically correct, all information preceding a syntax error is utilized. In other words, a syntax error acts as an EOS, but no scanning is performed on the E-list past the error and a diagnostic is produced.

TURN OFF

1.0 General Information

Given an options list, TURN OFF deactivates all legal options in the list. It traverses the list and either clears bits in the symbol table or decreases the frequency count in global option flags as each legal option is encountered.

2.0 Entry Points

TURN OFF is the only entry point for this routine.

TURN OFF is called by BUGACT whenever the line number, statement number or UPDATE identifier associated with the last statement processed is found as a TO bound in a linked bound list.

TURN OFF is called by BUGPRO upon the completion of its area and options list processing whenever the line number of a declarative statement is found as a TO bound in a linked bound list.

TURN OFF is called by BUGCON during OFF statement processing in order to deactivate the options specified as parameters in the OFF statement.

3.0 Diagnostics and Messages

There are no diagnostics issued by this routine.

4.0 Environment

4.1 The following local conditions are expected upon entry:

DFOPL = the ordinal (relative to the base of DEBUG) of the beginning of the options list

PADD = 1 if the options list is associated with a packet

= 0 if the options list is associated with an interspersed statement

NEXT = the ordinal (relative to the base of the Area list) of the TO bound associated with the options list

4.2 The following global conditions are expected upon entry:

SAREA = the ordinal (relative to the base of DEBUG) of the beginning of the Area list

SAASI = the ordinal (relative to the base of DEBUG) of the beginning of the Arrays And Stores Information list

ALLARR = the current frequency count for the all ARRAYS checked option

ALLCALL = the current frequency count for the all CALLS checked option

ALLFUNC = the current frequency count for the all FUNCTIONS checked option

GOTOSFL = the current frequency count for the GO TO's checked option

TRACEL = the current TRACE option level

TRCADD = the word containing the ordinals (relative to the base of AASI) of the beginning of the packet and interspersed TRACE option information lists

SSYMTAB = the ordinal (relative to the base of DEBUG) of the start of the Symbol Table

4.3 The following global conditions are updated upon exit:

ALLARR, ALLCALL, ALLFUNC, GOTOSFL, TRACEL, TRCADD

5.0 Structure

5.1 When called by BUGACT and by BUGPRO, options lists derived from packets are involved. In these cases, a check is made to see whether the given TO bound is accompanied by an activated FROM bound. If so, the options list is processed and the no bounds errors status is placed into the associated B-Cell. If not, the error code for "NO FROM bound" is entered in the B-Cell and the list is not processed.

5.2 Turn off the Options

When a line number, statement number, or UPDATE identifier is found to be a TO bound, the associated options list is scanned, using the same procedure as in TURNON. Instead of setting the list to 1, however, it is turned off, set to 0. In the case of all arrays, all calls or all functions to be affected, ALLARR, ALLCALL, or ALLFUNC, respectively is decremented to a minimum of 0. None will ever be negative.

GOTOS FL is decremented if the GOTOS option occurs in the list. The NOGO option is ignored, since it cannot be deactivated after it has been activated.

When STORES option is being turned off due to the occurrence of a TO bound, only those variables which appear in the list are affected. When the STORES option occurs in an OFF statement, there is no variable list associated with it, so all variables are affected.

DEBUG ER

1.0 General Information

DEBUG ER is called by BUGCON to print out error messages specified by the entry parameter M.

2.0 Entry Points

DEBUG ER is the only entry point for this routine.

3.0 Diagnostics and Messages

3.1 There are no fatal to compilation or execution, or non-ANSI diagnostics issued by this routine.

3.2 Informative Diagnostics

All error messages (unless otherwise designated) are preceded by the phrase "DEBUG SYNTAX ERROR LINE (Statement line number)"

3.2.1 M = 1

DEBUG ER is called by BUGCON when an expected delimeter is not found in the E-list representation of a Debug statement.

"EXPECTED /, (, ;, or), MISSING"

3.2.2 M = 2

DEBUG ER is called by BUGCON when characters are found beyond the closing right parenthesis of the parameter list of a Debug statement.

"CHARACTERS PAST)"

3.2.3 M = 3

DEBUG ER is called by BUGCON when a name is not present where expected in the E-list representation of a Debug statement.

"EXPECTED SYMBOL, MISSING"

3.2.4 M = 4

DEBUG ER is called by BUGCON during AREA statement processing when a name is not found in the Debug Routine List. (Only syntax checking is done on bounds and options associated with a routine not found in the DRL).

"ROUTINE DOES NOT APPEAR IN DEBUG ROUTINE LIST"

3.2.5 M = 5

DEBUG ER is called by BUGCON during AREA statement processing when a bounds pair which is syntactically correct is not semantically correct.

"BAD BOUND"

3.2.6 M = 6

DEBUG ER is called by BUGCON during TRACE statement processing when an integer is expected, but not found, in the parameter list.

"LEVEL INTEGER EXPECTED, MISSING"

3.2.7 M = 7

DEBUG ER is called by BUGCON during NOGO statement processing when characters appear beyond the option word NOGO in the E-list representation of the statement.

"ILLEGAL CHARACTERS PAST NOGO"

3.2.8 M = 8

DEBUG ER is called by BUGCON during OFF statement processing when an option appearing in the parameter list is not syntactically correct.

"LEGAL OPTION EXPECTED, MISSING"

3.2.9 M = 9

DEBUG ER is called by BUGCON during OFF statement processing when the OFF option appears as a parameter in another OFF statement.

"AN OFF STATEMENT SHOULD NOT APPEAR IN AN OFF STATEMENT"

3.2.10 M = 10

DEBUG ER is called by BUGCON during GOTOS statement processing when characters appear beyond the option word GOTOS in the E-list representation of the word.

"ILLEGAL CHARACTERS PAST GOTOS"

3.2.11 M = 11

DEBUG ER is called by BUGCON during TRACE statement processing when a character other than a left parenthesis is the first character beyond the option word TRACE in the E-list representation.

"ILLEGAL CHARACTERS PAST TRACE"

3.2.12 M = 12

DEBUG ER is called by BUGCON during STORES statement processing when the determination of the type of STORES option (STORES or STORES with relational operation) cannot be made from the E-list representation.

"EXPECTED,,), OR .OP., MISSING"

3.2.13 M = 13

DEBUG ER is called by BUGCON during AREA statement processing when the meaningless bound (*) appears in the E-list representation.

"BOUND FORM (*) MEANINGLESS, IGNORED"

3.2.14 M = 14

DEBUG ER is called by BUGCON during TRACE statement processing when the TRACE level is too large or during STORES with Relational Operators processing when the E-list representation of the constant associated with the

relational operator cannot be converted into the binary representation of a constant.

"ILLEGAL CONSTANT"

3.2.15 M = 15

DEBUG ER is called by BUGCON during STORES with Relational Operators processing when the Debug Operator Constant Table is full. (The table used during external packet processing, is 1000B words long, and contains the binary representation of the constants associated with relational operators).

"CONSTANT TABLE OVERFLOW"

(The proceding phrase here is "DEBUG ERROR ON LINE (statement line number)")

3.2.16 M = 16

DEBUG ER is called by BUGCON during DEBUG statement processing in an internal packet when a name in the parameter list is not the name of the routine being compiled.

"ILLEGAL ROUTINE IN DEBUG LIST"

4.0 Environment

The following conditions are expected upon entry:

M	=	a value as described above
DUKE1	=	the line number of the first line of the statement containing the error (the one currently being compiled)

5.0 Structure

Depending on the entry parameter M, a branch is made to print out the appropriate diagnostic. If the "no list" option is on, the line number of the statement containing the error is also printed.

BUGPRO

1.0 General Information

BUGPRO is called to bring the Debug Routine List (DRL) and the Debug Variable List (DVL) in from disk for two reasons:

(1) Before the first statement of a packet is processed so that the DRL and the DVL can be added to and then sent back to disk.

(2) At the appearance of the first executable statement so that the links of all Debug lists pertinent to the routine being compiled can be set up for the successful operation of the Debug mode packet options.

2.0 Entry Points

BUGPRO is the only entry point for this routine.

2.1 The first reason as described above is used in two places.

2.1.1 BUGPRO is called by DBGEPKT just before processing the first statement in the external packet on the input file if an external Debug input file existed and contained Debug information (D = lfn on FTN card).

2.1.2 BUGPRO is called by DBGIPKT just before processing the first statement in the internal packet.

2.1.3 BUGPRO is called by PS1CTL at the appearance of the first executable statement.

3.0 Diagnostics

3.1 There are no fatal to execution, fatal to compilation, or non-ANSI diagnostics issued by this routine.

3.2 Informative

If not enough room is available to bring the Area and Options lists into Central Memory and set up the links necessary for the successful operation of the Debug mode packet options, the following message will be issued:

'DEBUG: MORE CORE IS NEEDED FOR DEBUG PROCESSING'.

4.0 Environment

4.1 The following conditions are expected upon entry:

SDBGIND = the ordinal (relative to the base of DEBUG) of the start of the Debug random file index

DOLAST = the ordinal (relative to the base of DEBUG) of the first word of working storage (FWAWORK)

ELAST = the ordinal (relative to the base of DEBUG) of the last word of working storage (LWAWORK)

4.2 The following global conditions are expected on entry and will be updated upon exit:

SSYMTAB = the ordinal (relative to the base of DEBUG) of the beginning of the Symbol Table

ESYMTAB = the ordinal (relative to the base of DEBUG) of the end of the Symbol Table

ELIST = the ordinal (relative to the base of DEBUG) of the start of E-list

LELIST = the ordinal (relative to the base of DEBUG) of the start of the E-list of the statement associated with a logical IF statement

FIDIT = the ordinal (relative to the base of DEBUG) of the start of the fixed Area list (set to the start of the Symbol Table on entry)

SAASI = the ordinal (relative to the base of DEBUG) of the start of the Arrays and Stores Information Table

INDEXNO = 1 if the global DRL and DVL are to be brought in from disk (from DBGEPKT, DBGIPKT and if no internal packet exists)

= 3 if the routine DRL and DVL are to be brought in from disk (an internal packet exists)

FM LIST = 0 upon entry

= the word containing the ordinals (relative to the base of DEBUG) of the beginning of the line and statement number FROM bound linked lists and the beginning of the UPDATE Identifier bound linked list.

TO LIST = 0 upon entry

= the word containing the ordinals (relative to the base of DEBUG) of the beginnings of the line and statement number TO bound linked lists.

DUKE1 = the line number of the first executable statement.

5.0 Structure

The tasks of BUGPRO are listed in the order of their appearance.

5.1 BUGPRO called to set up DRL and DVL for packet processing. The DRL and DVL, if they exist, are brought in from disk and placed into Central Memory just above the Symbol Table. Control is then passed back to the calling routine.

5.2 BUGPRO called to set up lists for option processing.

5.2.1 DRL checked for existence of routine

The DRL, if it exists, is brought in from disk and placed just above the Debug random file index.

The DRL is then searched for occurrences of the name of the routine and the dummy routine SPIDER. The total lengths (room necessary for Area and Options lists associated with the routine) of all such occurrences found are added together and a test is made to see if enough room is available to bring in all of the associated Area and Options lists as well as the DVL. If enough room is not available, an appropriate diagnostic is issued and control returns to the calling routine.

5.2.2 The Symbol Table and E-list representation of the first executable statement are moved down in core to free room above the Symbol Table for the Area and Options lists.

5.2.3 Bring in Area Lists

The Area lists for all of the occurrences of the routine name in the DRL are placed in Central Memory one below another and then the Area lists for all of the occurrences of SPIDER following in the same manner.

5.2.4 Options and Bounds Linked Lists

Starting with the last Area list placed into Central Memory and preceeding upward until the end of first is reached, the Area lists are searched for Options words, FROM bounds, and UPDATE Identifier TO bounds. For each Options word found, the associated Options list is brought in from disk and placed just below the last list brought in. FROM bounds and UPDATE identifier TO bounds are entered into appropriate bounds linked lists.

5.2.5 Substitution of DVL Ordinals

The DVL, if one exists, is brought in from disk and placed just above the Debug random file index. E-list is moved down in Central Memory to allow for the growth of the Symbol Table resulting from the addition of any variables in the DVL which are not already in the Symbol Table. The variables are then added starting with the

first option field of the last Options list placed into Central Memory and preceding upward through each option field until the last option field of the first Options list placed. A search is made for DVL ordinals. All DVL ordinals found are then replaced with their corresponding Symbol Table ordinals.

5.2.6 Activating and Deactivating Options

Since it is conceivable that a FROM or TO bound count be associated with a line number smaller than that of the first line of the first executable statement, all line numbers prior to that of the first line of the first executable statement are checked for existence in either the FROM or TO line number bound linked lists or both. If the line number is found in the FROM bound list, TURNON is called to perform the appropriate action and, if found in the TO bound list, TURN OFF is called. (Note that only line number bounds are being checked here.)

PUT IN

1.0 General Information

Given the contents and the position of a 12-bit field in the options list, PUT IN places the contents into the position. PUT IN returns the next successive 12-bit field position if room is available for it. If no room is available for the position, the options list is terminated at the entry position and the list is sent to disk.

2.0 Entry Points

2.1 PUT IN is called by BUGCON while forming the options list in order to enter options, option parameter ordinals, and end of parameter list indicators into the options list.

2.2 PUT IN is called by BUGCLO when an options list has to be sent to disk in order to enter an end of parameter and options list indicator into the options list before sending it.

3.0 Diagnostics and Messages

There are no diagnostics issued by this routine.

4.0 Environment

4.1 The following local conditions are expected upon entry:

DFNESTW = the ordinal (relative to the base of DEBUG) of the options list word to contain the 12-bit to be entered

DFNEST = the field position in DFNESTW of the 12-bit field

OH = the contents of the 12-bit field to be added

4.2 The following global conditions are expected upon entry:

ELAST = the ordinal (relative to the base of DEBUG) of the last word of E-list

OPENFL = -1 if the Debug random file is open

= 0 else

4.3 The following conditions are changed upon exit:

DFNESTW = the ordinal (relative to the base of DEBUG) of the options list word containing the next successive 12-bit option field, if available

= the entry value if no room available

DFNEST = the field position in DFNESTW of the next successive 12-bit field, available or not

POW = 0 if no room available for the next 12-bit field

5.0 Structure

5.1 Put the contents of OH into the options word. The contents of OH are OR-ed into the field designated by DFNEST, after the field has been cleared out. DFNEST is updated to designate the next available field. DFNESTW is updated when all the fields are used.

5.2 If there are no more options words (you have extended down in core to ELAST), BUGPRO is called. BUGPRO will release space if it can. If the Debug random file is open, it is assumed BUGPRO has put something in it. The next available word is cleared and control returns to the caller.

If BUGPRO could not get more space, the list is terminated. The lists set up so far are kept intact, and BUGPRO is called again, this time to send all lists to disk.

SETARR

1.0 General Information

SETARR is called to obtain the ordinal of a particular word in the Arrays And Stores Information list. The word desired is indicated by the entry parameter ASHIFT. If the word is not available, return is made to the alternate return.

2.0 Entry Points

SETARR is the only entry point for this routine.

3.0 Diagnostics and Messages

3.1 There are no fatal to execution, fatal to compilation, or non-ANSI diagnostics issued by this routine.

3.2 Informative

3.2.1 If the frequency count is too large, the message printed out is:

'TOO MANY STORES OR ARRAYS OPTIONS REFERENCES FOR (Symbol Table entry name).'

3.2.2 If an AASI word is not available, the message printed out is:

'REQUEST FOR AASI WORD DENIED. ASK FOR MORE CORE.'

4.0 Environment

4.1 The following local conditions are expected upon entry:

ASHIFT = one of the values as described above

OH = the Symbol Table ordinal (times 2) of the Symbol Table entry to be worked with

4.2 The following global conditions are expected upon entry:

SAASI = the ordinal (relative to the base of DEBUG) of the beginning of the Arrays And Stores Information list

NAASI = the ordinal (relative to the base of the AASI list) of the last successive word used in the AASI list

SSYMTAB = the ordinal (relative to the base of DEBUG) of the start of the Symbol Table

ESYMTAB = the ordinal (relative to the base of DEBUG) of the end of the Symbol Table

ELAST = the ordinal (relative to the base of DEBUG) of the last word in E-list

DOLAST = the ordinal (relative to the base of DEBUG) of the first word available of working storage (FWAWORK)

FIDIT = the ordinal (relative to the base of DEBUG) of the beginning of the fixed area list

DBGPROG = the ordinal (relative to the base of DEBUG) of the word containing the name of the routine currently being compiled

NO LIN = 0 if successive words are being taken from the space available in the AASI list

= -1 if no successive words are available in the list and new words must be obtained either through garbage collection or by increasing the length of the AASI list

AASIADD = the word containing the ordinal (relative to the base of the AASI list) of the beginning of the linked list of freed AASI words

4.3 The following conditions are changed upon exit:

AASI = the ordinal of the desired word in the Arrays And Stores Information list (meaningless if the alternate return is made)

5.0 Structure

5.1 ASHIFT = -1

SETARR is called by TURN OFF to obtain the AASI ordinal of a Symbol Table variable. If the ordinal does not exist, the alternate return is made.

5.2 ASHIFT = 0

SETARR is called by TURN ON while processing an interspersed statement in order to obtain an AASI word for either STORES with relational operators or TRACE options information. If no word is available, the alternate return is made.

5.3 ASHIFT = 8

SETARR is called by TURN ON during ARRAYS options processing to obtain the AASI ordinal contained in a Symbol Table entry. If no such ordinal exists, an AASI word is obtained and cleared and its ordinal is returned. If an AASI word is not available, the alternate return is made. If the AASI ordinal is found in the Symbol Table entry, the frequency count of the ARRAYS option in the word pointed to by the ordinal is checked before return. If it is too large, the alternate return is made.

5.4 ASHIFT = 38

SETARR is called by TURN ON during STORES option processing to obtain the AASI ordinal contained in a Symbol Table entry. The action performed by SETARR is the same as for ASHIFT = 8 except that the frequency count check is made for the STORES option.

5.5 ASHIFT = 10

SETARR is called by TURN ON during STORES with relational operators option processing to obtain the AASI ordinal contained in a Symbol Table entry. The action performed by SETARR is the same as for ASHIFT = 8 except that the frequency count check made is a dummy and the frequency count is never too large.

BUGCLO

1.0 General Information

BUGCLO is called to send lists from Central Memory to disk. Depending on the entry parameter POW, either all lists are sent in order to terminate packet processing by flushing all Debug information to disk or only some area lists are sent in an effort to release some space for additional list construction.

2.0 Entry Points

BUGCLO is the only entry point for this routine

2.1 POW = 0

The alternate exit is never taken.

BUGCLO is called by DBGEPKT at the end of external packet processing to flush all Debug information still in Central Memory to disk.

BUGCLO is called by DBGIPKT at the end of internal packet processing to flush all Debug information still in Central Memory to disk.

BUGCLO is called by BUGCON while attempting to make an entry into the DRL if no room is available in the DRL for the entry and there is no room for the DRL to expand. (Packet processing is terminated prematurely: All debug information already in list form in Central Memory is flushed to disk but further Debug statements will only be checked for syntax and not placed into list form.)

BUGCLO is called by PUTIN (or BUGCON) when another word is needed for the options list under construction but another word is not available. (Again premature termination. The options list is terminated and will exist for its associated area lists in truncated form).

2.2 POW > 0

BUGCLO is called by BUGCON when another word is necessary for the construction of either an Area of an Options list and another word can only be obtained if some Area lists are sent to disk and the space they occupied released.

BUGCLO is called by PUTIN when another word is necessary for the construction of an Options list and the word must come from released Area list space.

(The alternate entry is taken here whenever no Area list can be sent to disk and therefore no space is released).

2.3 POW = 1

The alternate exit is always taken.

BUGCLO is called by BUGCON during AREA statement processing if a call to BUGCLO with POW > 0 has just returned through the alternate exit. Premature termination occurs here just as above with POW = 0 except that any Area lists begun since the termination of the last Options list are destroyed rather than sent to disk. (No room for an Area list is taken to mean no room for its associated Options list. An Area list cannot exist without an Options list and so the Area list is destroyed).

3.0 Diagnostics and Messages

There are no diagnostics issued by this routine.

4.0 Environment

The following conditions are expected upon entry:

POW	=	a value as described above
SDRL	=	the ordinal (relative to the base of DEBUG) of the start of the DRL
EDRL	=	the ordinal (relative to the base of DEBUG) of the end of the DRL

ELAST = the ordinal (relative to the base of DEBUG) of the end of E-list

DFNESTW = the ordinal (relative to the base of DEBUG) of the current word being constructed in the Options list

AREAEND = the ordinal (relative to the base of DEBUG) of the last word of the Area list

LNGIND = the length of the DEBUG random file index

RECORD = the number of the last record written out on the Debug random file

5.0 Structure

The tasks of BUGCLO are described in order of appearance.

5.1 Options List Sent to Disk

If an Options list exists in Central Memory and all lists are being sent to disk, the 12-bit termination of Options list mark (7777B) is added to the list and the list is sent to disk.

5.2 Destroy Unwanted Area Lists

If some Area lists could not be completed because of lack of room, they are destroyed. This is done by searching for and then clearing all DRL entries associated with Area lists which were not sent to disk. (At this time no Options list is still in Central Memory and all completed area lists would have already been sent to disk).

5.3 Get Record for DRL Entry Area List

If there are no more record numbers left in the Debug random file index, a test is made to see if there is enough room to increase the length of the index by 50. If no room is available, a test is made to see if any Area list have already been sent to disk. If no lists have been sent to disk, it is assumed that not enough room is available to finish processing any more Area and Options list, POW is set to -1 and control goes back to

5.2. If Area lists have been sent to disk, the amount of space released is calculated and the remaining lists are moved down to start just above the end of the index. Control now returns to the test to see if the length of the index can be increased by 50. If there is enough room to increase the length by 50, the Area and Options lists still in core are moved up 50 words, the length of the index is increased and the new record number is valid.

5.4 Send Area List to Disk

The DRL is searched for the next entry with associated Area lists still in Central Memory. If no such entry exists, control is passed to 5.5. The Area lists associated with the DRL entry are then sent to disk one behind the other in the same record and control is transferred to 5.3. In the case when POW > 0, all Area lists except those associated with the Options list then under construction are sent to disk.

5.5 Send DRL and DVL to Disk

If the Area lists were to be sent to disk to release space for the further construction of lists, control is passed to 5.6. Otherwise, the DRL entries are sorted in alphabetical order and the DRL and DVL are sent to disk.

5.6 All lists or portions of lists still in Central Memory are moved down in Central Memory by the number of words in the Area lists sent to disk. The list pointers in the DRL and Area lists remaining are updated to reflect the move and control is transferred to the calling routine.

R-list Language Description

1.0 R-list is an intermediate language which is intended to be both easy to generate and convenient to produce good assembly code from. The language is register independent as much as possible; there shall be a provision for specifying specific registers. A limited macro facility shall be provided to facilitate the generation of frequently occuring code and reduce the size of the intermediate R-list. Generally, the operation code in R-list will correspond to the operation code of the associated machine instruction.

1.1 General Description

There are four formats for R-list instructions to permit maximum description of the operation requested in the minimum amount of space. In all cases, the following definitions will hold.

First word will have the same format:

Bit 59 = 0

Bit 58 = 1

Bit 47-57 - Operation Code (OC)

It is formed by placing the operation code in a B register and appending it to the rest of the information in the first word of the R-list instruction by using a PACK instruction. Any negative words encountered in the R-list will be ignored.

1.2 R Fields

All R fields will be sixteen bits long. All R fields generated during pass one will have the high order bit equal to zero; intermediate R's resulting from macro processing will have that bit set.

1.2.1 RI Fields

RI indicates the result of the operation indicated by OC or an operand of a branch instruction. This field will always be rightmost in the first word whenever it occurs.

1.2.2 RJ Fields

These fields indicate the first operand of a triple address instruction.

1.2.3 RK Fields

These fields indicate the second operand of a triple address instruction.

1.2.4 RF Fields

These indicate the index function to be added to a base address or may be used to indicate an operand of a branch.

1.3 CA Fields

CA fields are eighteen bits long and usually contain the constant addend to be added to the reference to an array element. They may also contain some other constant.

1.4 I and H Fields

The I field is twelve bits long and identifies the table for which H is an ordinal. The HI field is eighteen bits long. The defined values for I are as follows:

0 = Symbol Table	5 = APLIST ordinal
1 = unused	6 = 1 Branch if label (GL)
2 = DO Range temporary	7 = Requests a negative zero entry in aplist
3 = Vardim temporary	10 = Optimizing Temporary
4 = unused	11 = Invariant Temporary

If the HI field is negative or zero, both I and H will be ignored. If the H2 field is non-zero, it is interpreted as an ordinal in the symbol table and the final symbol generated will be H1-H2. All H1 are interpreted as ordinals in the list specified by I.

1.5 IN Field

IN fields are eighteen bits long and contain constants.

1.6 SO Fields

SO fields are fourteen bits long and modify the operation code. The high order bit of the SO Field is used only in macro descriptions. The next two bits specify respectively, a sequence terminating and unlocking instruction, a locking instruction. A locking operating reserves a register for the RI of the operation, this reservation holds until an instruction with the next to high order bit in the SO field is encountered. The next bit is set to indicate a register specification is not to be held until the end of a sequence. The low order two octal digits are used to specify a class of registers or a particular register. The right hand two bits specify a register class as follows:

1 - X

2 - B

3 - A

If the next bit is set, the other octal digit specifies the particular register of the class. This specification stipulates which register the RI of the instruction is to be placed in, i.e., the destination register of the operation. Currently only X's and B's may be specified as destination and the particular one must be stated.

2.0 Type I R-list

2.1 Type I Format

All type I instructions occupy one word and consist of operation code, RJ, RK, and RI in that order.

2.2 Type I Operations

Description	R-list Code	Machine Code
Transmit	10	10,22
AND	11	11
OR	12	12
Exclusive or	13	13
Transmit Complement	14	14
Stroke	15	15
Implication	16	16
Equivalence	17	17
Floating Add	30	30
Short Add (after all other uses of RI)	2	53
Short Difference Load	3	57
Short Difference Store	7	57
Floating Subtract	31	31
Double Floating Add	32	32
Double Floating Subtract	33	33
Integer Add	36	36
Integer Subtract	37	37
Floating Multiply	40	40
Double Floating Multiply	42	42
Floating Divide	44	44
Rounded Add	63	34

Rounded Subtract	64	35
Rounded Multiply	65	41
Rounded Divide	66	45
Short Add	46	6N, 7N n=3,4,6
Short Subtract	67	67, 77
Short Load	41	57
Short Store	45	56, 57
Index Right Shift	23	23
Index Left Shift	22	22
Normalize	24	24
Round and Normalize	25	25
Unpack	26	26
Pack	27	27
Shift Transmit	62	10,22
Unpack into B0	34	26

3.0 Type II R-list

3.1 Type II R-list Format

All Type II instructions occupy one word and consist of operation code, IN field, SO field, RI field, in that order.

3.2 Type II Operations

Description	R-list Code	Machine Code(s)
Form Mask	43	43
Clear Register	47	13,43
Set	61	61, 71
Define	53	NA
Register Store	52	NA

The Type II set operation may only be used for placing an 18 bit constant into a B or X register.

The Define operation is used to set an initial condition for the sequence of which it is a member. A Define states that RI is in the register specified by the SO field. The intended use of the Define is to specify the whereabouts of results of function references on returning.

The register store is used to stipulate the destination register of the previous operation of which RI was the result. In effect, this is used to permit the appending of an SO field to a Type I instruction. Once an R is placed in a register by an SO or register store specification, the register is no longer available until the next sequence.

Neither Define nor Register Stores result in the production of any executable code.

Define and Register Stores differ from locks in that locked definitions hold until the end of DO loop jump is encountered, which may be several sequences. Define and register store specifications have meaning only within their own sequences.

If an SO field specifies a reserved register, the reservation will be overridden. Moral: garbage in, garbage out.

4.0 Type III R-list

4.1 Type III R-list Format

All type III instructions occupy two words. The first word consists of the operation code, CA field, SO field, and an RI field in that order. The second word contains a RF field, I field, and an H field right justified.

4.2 Type III Operations

Description	R-list Code	Machine Code(s)
Load	50	50 - 57
Store	51	50-57
Jump if RI=RF	70	04
Jump if RI .NE. RF	71	05
Jump if RI .GE. RF	72	06
Jump if RI .LT. RF	73	07
Zero X Jump	74	030
Non-zero X Jump	75	031
Positive X Jump	76	032
Negative X Jump	77	033
Indexed Jump	102	02
Set	104	60 - 77
APLIST	56	NA
Left Shift (CA)	20	20
Right Shift (CA)	21	21

APLIST entries will specify the address of the actual parameter in the I, H1, and CA fields. The RI field will contain an ordinal which will say which parameter list this entry is a member of. Although entries for a given

argument list (identical RI fields) need not be consecutive APLIST entries, they must be in order.

Any R which is operated on by either a 20 or 21 operation must be defined by shift transmit immediately prior to the shift operation.

5.0 Type IV R-list

5.1 Type IV R-list Format

All type IV instructions occupy one word, consisting of an operation code, CA, I and H fields.

Description	R-list Code	Machine Code(s)
Unconditional Jump	54	0400
Return Jump (60 bit)	101	01
Entry	55	NA
End of Statement	57	NA
End of Sequence	100	NA
Label	60	NA
End of R-list	103	
Return Jump (30 bit)	105	01

6.0 MACROS

6.1 Macro References

Macros are referenced using a Type II format. The OC field contains the macro ordinal and is formed by placing the complement of the macro ordinal (MO) in a B-register and concatenating it to the rest of the information by use of a PACK instruction. The IN field contains the number of words which follow which contain parameters.

6.2 Actual Parameters

The actual parameters immediately follow the macro reference in the following order:

1. symbols, if any
2. R's, if any
3. constants, if any

6.2.1 Actual symbolic parameters are specified two to a word, the first in the lower half of the word, the second in the upper half, and so on. Each half word consisting of a right justified I field and H field.

6.2.2 Actual R parameters are three to a word, as is Type I format, with the first being rightmost.

6.2.3 Actual constant parameters are specified three to a word in 18 bit fields right justified.

6.3 Macro Descriptors

The descriptor locates the macro text and specifies its length and the number of parameters of each type. It occupies one word and has the following format:

Bits 0- 17	MA	beginning address of the macro
Bits 21- 35	ML	length of the macro in words
Bits 36- 41	NI	number of generated intermediates
Bits 42- 47	NK	number of integer constant parameters
Bits 48- 53	NR	number of R parameters
Bits 54- 59	NS	number of symbolic parameters

6.4 Macro Text

The macro text is written in normal R notation with the following modifications:

1. If the high order bit of an RI, RJ, or RK field is not set and is greater than 1, it is interpreted as a formal reference to an R; the remainder of the

field specifies the ordinal of the actual parameter list.

2. If the H field is zero, the IH pair is interpreted as a formal reference with the I field containing the ordinal of the actual parameter in the parameter list.

3. IN and CA fields are interpreted as containing ordinals of actual parameters if the high order bit of the SO field is set. The use of type IV format instruction requiring a CA field as a formal parameter is not allowed.

7.0 Four types of R-list Entries

TYPE I		OC	RJ	RK	RI
	2	10	16	16	16
TYPE II		OC	IN	SO	RI
	2	10	18	14	16
TYPE III	2	10	18	14	16
		OC	CA	SO	RI
		H2	RF	I	H1
		14	16	12	18
TYPE IV		OC	CA	I	H1
	2	10	18	12	18

8.0 Defining R-list Macros

All macros are contained in the routine MACROX which is part of Pass 2. A set of COMPASS macros has been constructed to facilitate the definition of R-list macros. The following describes their use.

Structure: Each macro definition must begin with an RMACRO macro reference followed by the text of the R-list macro followed by an ENDR macro reference.

RMACRO. The RMACRO reference has the following form:

OC - RMACRO

Address field - NOSY, NOR, NOK

Where:

NOR - the number of R's in the macro reference.
NOSY - the number of IH fields in the macro reference.
NOK - the number of constants in the macro reference.

ENDR. The ENDR reference has the following form:

OC - ENDR

Text. Text is written with R-list mnemonic operation codes in the OC field; a list of them is attached. The variable field will contain the arguments to the operation in the following orders:

```
TYPE I    RI, RJ, RK
TYPE II   RI, IN, SO
TYPE III  RI, RF, CA, SY, SO
TYPE IV   CA, SY
```

Omitted fields are interpreted as zero.

R-fields. All R-fields (RI, RJ, RK, and RF) must either begin with an I or P or must consist of 1 or zero. If it begins with an I or P, the rest of the field must be a decimal quantity giving the ordinal of the R in the associated list.

I - the list of R's generated by the macro processor. These must be numbered greater than 0.

P - the list of R's in the parameter list referencing the macro.

0 and 1 are the constant R's located in B0 and A0 respectively.

IN and CA fields. These fields must either be empty or contain an X or a B followed by a digit in the range 0-7 followed by a period. If an X or B is specified, that specification may optionally be preceeded by a T to indicate a temporary register assignment.

SY fields. These must be either empty or a constant, in which case, it is interpreted as being an ordinal in the list of IH parameters.

Example:

```
ABSF    RMACRO      0,2,0
        SKT         I1, P1
        KRS         I2, I1, 59
        XOR         P2, I2, P1
        ENDR
```

I/O Calling Sequences

1. This sections describes the I/O calling sequences for the FORTRAN common object library. Parameters are passed to the object routines via actual parameter lists (APlists). Generation of the I/O aplists is described in the section on I/O statement processing.

1.1 Common Parameter List Flag Bit Definitions

Three flag bits have been defined for modifying the context of 18- or 24-bit address fields passed through APlists to the common object time I/O routines. The flag bits are located in the parameter word that they modify, in bit positions 58 and 59. The address field subject to modification is normally right-justified in bits 0-17 or 0-23. Flag bit meaning is defined as follows:

Flag Status	Context
All flags = 0	The address field contains the direct address of the parameter being passed; the parameter resides in SCM.
LCM =1	The address field contains the direct 24-bit parameter address; the parameter resides in LCM.
VAR = 1	The address field points to a variable.
FP = 1	The address field contains an ordinal to a formal parameter.

1.2 APlist Parameter Element Expansions

1. FIT pointer VFD 1/VAR, 1/FP/ 40/0, 18/FITADR

   where FITADR = vector to FIT word 1

2. LIST item VFD 1/LCM, 1/FP, 1/FP, 1/IND, 3/0, 6/TYPE, 18/NBREL, 6/0, 24/ITEM

   where ITEM = vector to an I/O list item

TYPE = type of this list item. Possible values are:

0 - reserved
1 - logical
2 - integer
3 - real
4 - double
5 - complex
6-63(10) - reserved

If IND = 0, NBREL = number of contiguous elements in the list item

If IND = 1, NBREL contins the SCM address of the list element count

3. APEND VFD 60/END

where END = +0 to denote the end of an I/O list; suitable terminal action shoud be taken.

= -0 to denote an intermediate interruption in an I/O list; subsequent list elements will be requested on the next call(s). (This can occur, for example, when a list element has a function call as a subscript).

4. RECORD length VFD 42/0, 18/WDCNT

where WDCNT = number of words in a record.

5. FORMAT pointer VFD 1/LCM, 1/FP, 1/VAR, 33/0, 24/FMT

where FMT = vector to a FORMAT specification.

6. MODE pointer VFD 42/0, 18/SCM address of word that contains the file mode.

7. BUFFWA VFD 1/LCM, 1/FP, 1/VAR, 33/0, 24/FWA

where FWA = vector to FWA of an I/O buffer area.

8. BUFLWA VFD 1/LCM, 1/FP, 1/VAR, 33/0, 24/LWA

where LW = vector to LWA of an I/O buffer area.

9. SRTING pointer VFD 1/LCM, 1/FP, 1/VAR, 33/0, 24/DATSTR

where DATSTR = vector to a data string for DECODE input or ENCODE output.

10. COUNT pointer VFD 1/LCM, 1/FP, 1/VAR, 33/0, 24/CNT

where CNT = vector to a word containing a character count.

11. TRACE VFD 12/LINECNT, 18/IDADR

where LINECNT = line number of source program statement that caused this call to be compiled, in binary. If LINECNT = 0, no line number will be printed in error traceback listings (to accomodate RUN compiler output).

IDADR = SCM address of a word containing the program unit identification for traceback. The ID word will contain the program unit name in bits 59-18, in left-justified display code with blank (55B) fill, and the address of the program unit entry point in bits 17-0. If the program unit has more than one entry point, the address will be that of the first or main entry point.

2.0 Restrictions

1. The FWA and LWA of an I/O area designated by a BUFFER IN or BUFFER OUT statement must both reside in either LCM or SCM. The compiler will have diagnosed any attempt to mix LCM and SCM addresses.

3.0 Functional Identification of Input/Output Routine Entry Points

Input/Output Type	Initial Call (via APlist)	Intermediate or Final CAlls via APlist
Binary (unformatted)	INPBI./OUTBI.	INPBR./OUTBR.
Coded (formatted)	INPCI./OUTCI.	INPCR/OUTCR.
Buffered I/O	BUFIN./BUFOUT.	
NAMELIST	NAMIN./NAMOUT.	
Core-to-core transfers	DECODI./ENCODI.	DECODR./ENCODR.

Subroutine Linkage and Formal Referencing

All references to formal parameter in the following circumstances will be direct, i.e., address substitution will be performed at subroutine initialization time:

1. Array element reference.

2. Entry in an actual parameter list.

3. Reference within a DO.

4. Reference to a formally defined subprogram.

All remaining references will be via indirect references. The address of the actual parameter list will be maintained in A0 throughout the execution of the text of the execution of the text of the subprogram. The previous A0 will be saved in local memory.

SUBROUTINE LINKAGE

Calling Sequence:

```
SA1 APLIST

RJ  SUBR
```

Where APLIST is the address of the first element of the actual parameter list. The list has the address of the actual parameters stored in order one per word in the low order position with zero fill. The list is terminated by a full word of zero. SUBR is the subroutine name.

Substitution List Format:

Each non-zero entry has the following format:

```
VFD  3/2,9/S,10/0,18/CA,2/0,18/IA
```

Where:

IA is the address of the instruction to be modified.

CA is the constant addend.

S is the number of bit positions to be shifted over to place the address field of interest in the low order 18 bits, i.e., 0, 30, or 45.

G is unused.

The list is ordered in sequence of the associated formal parameters. Following all entries referencing a given formal parameter is a zero word followed by the entries associated with the next formal parameter.

Restriction:

Consecutive entries in the substitution list may not reference the same instruction word.