B1

INTERNAL MAINTENANCE SPECIFICATION

FORTRAN V5

PROPRIETARY INFORMATION

This document is part of a software product which is the property of Control Data Corporation and is proprietary to it. Distribution is restricted to customers having a valid license for the use of the software product; use and disclosure of information in this document are governed by the terms of the license.

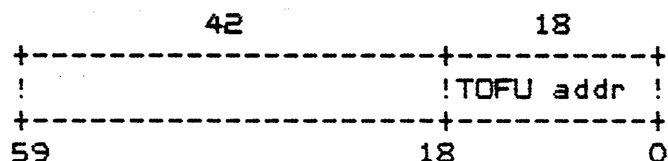
77987506A

0148P-0030A/0103P

These registers are more appropriately discussed within the in-code comments for COMCBUB, COMCBUN, and COMCTOK. See the preambles to these COMDECK's for details. The most important thing to know at this point is that TOK is extremely register driven.

TOK=MN - MAIN DRIVER

As was previously stated, TOK=MN is responsible for reading the next binary TOGEL instruction to execute/interpret. It knows which TOFU is to process a particular instruction because the opcode in bits 0 thru 17 of each instruction is actually the address of the appropriate TOFU.



(compare with figures 3.19 and 3.20)

Bits 59 thru 18 contain information that the appropriate TOFU needs in order to do its job, and varies with each instruction.

TOK=GS - GROUP.....SQZ

TOK=GS is the TOFU that is invoked when the following binary TOGEL instruction occurs.

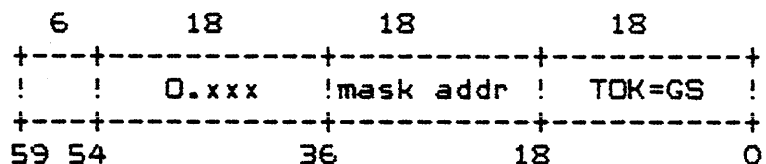


Fig. 3.23. GROUP,...,SQZ TOGEL instruction format

This "instruction" performs a TOGEL GROUP with blank squeeze. TOK=GS calls BUB (Burst/Build with Blank Squeeze) to perform the actual GROUP. The "mask addr" in bits 18 thru 35, defined by symbols TG.MXAP and TG.MXAL, is the address of a shift mask that defines the characters that BUB is to GROUP together.

An understanding of how BUB and BUN work is essential to an understanding of how bits 18 thru 35 are used. Briefly stated, this shift mask contains a bit position for every character in the character set. BUB will GROUP all the characters that have their respective bit in the shift mask ON. In this way, the programmer is able to selectively specify to BUB which characters are to be GROUPed. See 3.3.3 Design/Supporting Details/BUB-BUN Character Access Method.

"O.xxx" in bits 36 thru 53, defined by TG.TOTP and TG.TOTL, is the token type (i.e. O. symbol value) that BUB is to associate with this GROUP.

For example, the TOGEL macro:

GROUP (0..9),CONS,SQZ

would assemble as:

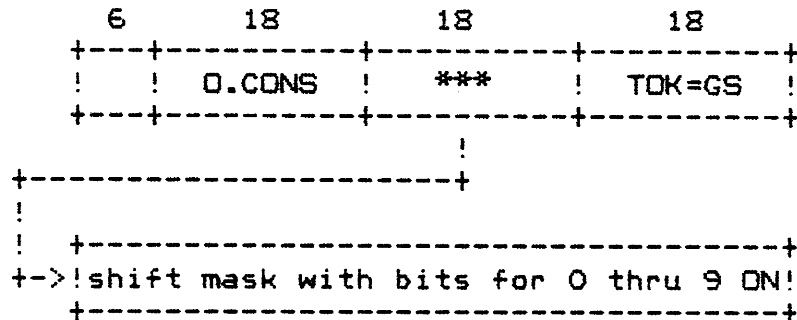


Fig. 3.24

TOK=GN - GROUP.....NSQZ

TOK=GN is invoked for the following binary TOGEL instruction.

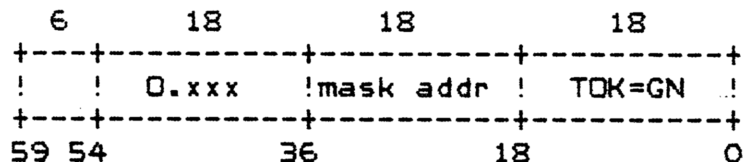


Fig. 3.25. GROUP,...,NSQZ TOGEL instruction format

This instruction performs a TOGEL GROUP with no blank squeeze, and is very much the same as TOK=GS except that it calls BUN (Burst/Build with No Blank Squeeze) to do the actual GROUPing.

TOK=GO - GOTO

TOK=GO is invoked when TOK=MN encounters the following binary TOGEL instruction.

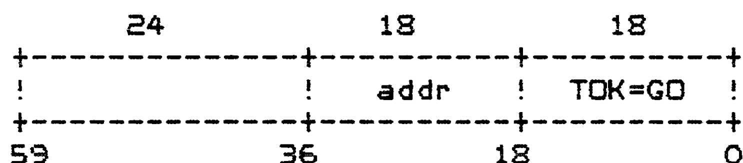
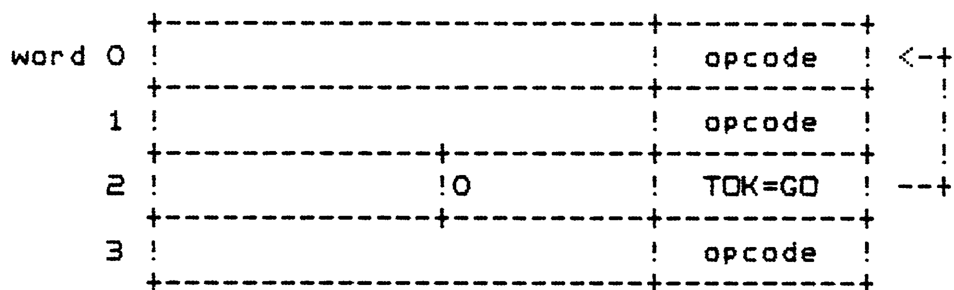


Fig. 3.26. GOTO binary TOGEL instruction format

TOK=GO merely transfers control of the interpreter to the binary TOGEL instruction at "addr" by setting TOK's pseudo P register (register A0) to "addr".

For example, given:



and assuming that P = word 3, meaning that TOK is executing/interpreting the instruction at word 2*, then TOK=GO will set P = word 0. This will force TOK=MN to "read" the instruction at word 0 as the next instruction to execute/interpret.

* Remember that TOK's pseudo P register points to the next instruction to execute. Therefore, while within a TOFU, the instruction being executed is at P-1.

TOK=COF - CASEOF (...)

TOK=COF is invoked to process a CASEOF-TOKEN-ENDC structure when the following TOGEL instruction occurs:

```
     12      12         18          18
+------+-------+----------+----------+
! COA  !  COZ  !   TAD    ! TOK=COF  !
+------+-------+----------+----------+
59    48      36         18          0
```

where:

COA = 1st character in CASEOF range (represented in host character set, i.e. display code for FTN). E.g. for CASEOF (A..Z), COA would be "A".

COZ = last character in CASEOF range in a "rotated character set" representation. E.g. for CASEOF (A..Z), COZ would be "Z" (but not a host character set "Z"; see below).

TAD = address of TOKEN character map for this CASEOF.

Fig. 3.28. CASEOF binary TOGEL instruction format

Because CASEOF processing is so detailed, this discussion is approached at two levels. The first is an overview that introduces the internal structures involved in CASEOF-TOKEN-ENDC processing. It is assumed that the reader understands the TOGEL for this (i.e. that the reader understand what we are trying to do). The second level is considerably more detailed and discusses how these internal structures are used by TOK=COF.

Overview

TOK=COF uses the CASEOF-TOKEN-ENDC structure to generate a single token for the character that the token generator is pointing to when TOK=COF is called.

For a CASEOF-TOKEN-ENDC, the TOGEL macros (in COMATOK) generate a binary TOGEL instruction as in figure 3.28 to the TOM, and also generate 2 auxiliary "tables" that are used to determine what token to generate for a character within the CASEOF range.

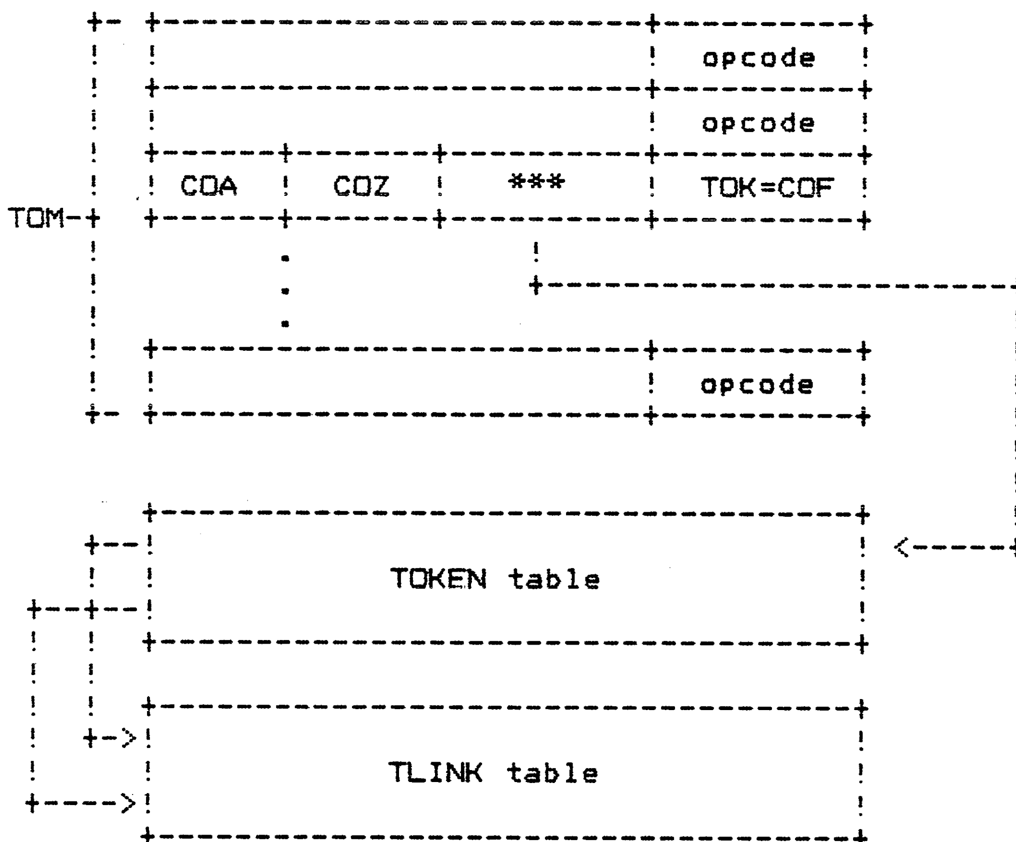


Fig. 3.29. TOM-TOKEN-TLINK structural relationship

The first table, the TOKEN table, is merely a character map that contains a token situation for each character in the CASEOF range. In the simplest case, the character to generate a token for is used as an ordinal into the TOKEN table to pick up a token skeleton that is copied to the token buffer*.

The second table, the TLINK table, is used to determine whether or not the character to entoken is involved in a multiple token sequence. The TLINK table provides the information necessary to know whether a multiple token sequence has occurred and what to do if one does occur.

It should be noted that the TLINK table is only generated by COMATOK for TOKEN entries that are describing multiple token syntaxes.

---

* See figure 3.17a and related text.

For example, for the TOGEL:

```
CASEOF (+../)
  TOKEN PLUS                   +    <--+
  TOKEN MINUS                  -    <--+-------TOKEN
  TOKEN STAR                   *    <--+
    TOKEN EXP,(STAR,STAR)      **      !   <--+
  TOKEN SLASH                  /    <--+      +-TLINK
    TOKEN CAT,(SLASH,SLASH)    //          <--+
ENDC
```

COMATOK will generate a TOKEN table consisting of PLUS, MINUS, STAR, and SLASH; and a TLINK table consisting of EXP and CAT. If the EXP and CAT entries in the above TOGEL had been left out, then no TLINK would be generated.

It can also be seen that TOKEN and TLINK are conceptually different structures: TOKEN is a character_map, i.e. it is used to map a single character into a token. TLINK is a token_map, i.e. it is used to map a sequence of tokens in the token buffer into a new token.

TOKEN and TLINK are related as follows. After TOK=COF has used the character to entoken as an ordinal into the TOKEN table and picked up an appropriate token skeleton, it checks the LNK field in this skeleton* to see if this token is involved in a multiple token sequence. If so, the LNK field points to the beginning of a linked_list in TLINK that describes all the multiple token syntaxes/sequences that this token is involved in.

TOK=COF will move through this linked_list checking to see whether the token buffer contains any of the syntaxes/sequences that are described here. If it finds a match, then a replacement token is extracted from TLINK and copied to the token buffer. See figure 3.29.

The previous paragraphs have (it is hoped) given the reader a cursory overview understanding of the structures involved in processing the CASEOF-TOKEN-ENDC structure. The primary intent of these paragraphs is to clearly distinguish in the reader's mind the nature of the difference between the TOM, the TOKEN character map, and the TLINK syntax table; and to briefly describe how these structures relate to the TOGEL that generated them.

---

* Defined by symbols TK.LNKP and TK.LNKL.

3.3.2.5 CST and Statement Classification

CST determines the nature of the current statement and sets flags for the front end main loop. CST must handle two classes of statement, semantically defined (IF,DO, assignment, etc) and keyword statement. During the entokening process, several cells are maintained which keep the status of nesting level of parenthesis, existence and location of '=' and ','. These cells are used to determine the nature of the semantically defined statements.

3.3.3.1 BUB/BUN Character Access Method

The BUB/BUN character access method consists of 2 subroutines, each of which exists as a common comdeck: COMCBUB - Burst/Build Characters with Blank Squeeze, and COMCBUN - Burst/Build Characters with No Blank Squeeze. Together, they constitute a general purpose discipline for efficiently accessing display code (6 bit) characters.

a. GROUPS_OF_CHARACTERS

As I saw it, one of the fundamental problems encountered during the manipulation of characters was the need to be able to transform characters from their "operating system" format of 10 characters per word into groups that are meaningful to any particular host program. For example, in the following:

HELP IS ON THE WAY

the "groups" are the individual English words . . . the blanks are only meaningful as word-separators.

Whereas, in the FORTRAN:

NO HELP = FROM + HERE

the "groups" are the individual variable names and operators . . . the blanks are meaningless to a FORTRAN compiler.

This "grouping" of characters is the process of bursting characters from their raw 10 character per word format and building them into meaningful, or perhaps said more appropriately: useful, groups of characters.

b. LEFT_TO_RIGHT

Given an input source line in packed (10 character/word) format and a starting character position in that line, a call to BUB or BUN will form a single group of characters. BUB will burst/build characters, ignoring blanks; and BUN will burst/build characters, ignoring nothing.

This introduces a heretofore unmentioned, implicit assumption in most lexical scanning: left to right. BUB and BUN burst and build characters from left to right.

For example, given:

```
THE MANAGERS ARE COMING, THE MANAGERS ARE COMING . . .
    !
```

a call to BUN specifying the alphabetic group, would produce:

```
THE MANAGERS ARE COMING, THE MANAGERS ARE COMING . . .
            !
```

and return to its caller, the group:

```
    MANAGERS
```

Those of you who have already been there will notice a startling similarity between a BUB/BUN group and a TOGEL GROUP. See 3.2.4 Design/Executives/TOK and Token Generation/Learning TOGEL.

3.3.3.2 C$ Statement Processing

This section discusses C$ (compiler directive) statement processing, as it relates to LEX. The actual "parsing" or statement processing for C$ statements occurs in the deck CDDIR, and is not discussed here.

a. THE_PROBLEM

The first compiler directives came along relatively late in the evolution of both FTN (TS and OPT) compilers. They could be used for dynamically turning on and off the compiler-generated source listing.

If the compiler detected:

```
C/    LIST(NONE)
```

on a source line, source listing was suppressed (unless, of course, a statement was found to be in error, etc, etc). And if:

```
C/    LIST(ALL)
```

occurred on a source line, source listing would be turned back on.

One of the major challenges of this scanner design was to lick the "C/ LIST"ing logic problem.

b. THE SOLUTION

The solution to the compiler directive problem in lexical scanning involves the following: first, the *external* specification of compiler directives was chosen *very* carefully, and second, LEX was structured in such a fashion so as to easily accommodate listing logic pathologies.

Compiler directive lines were made to "look" like comment lines for ANSI portability, but they very purposely do not *act* like comment lines. Compiler directives may *not* occur within a continuation sequence. That is, the following is illegal:

```
 CALL ING  SOAT1$    LIST(NONE)
+ (ALL, CARS)
```

3.4 HEADER: Process Program Unit Header Statements

<u>Abstract</u>: HEADER contains statement processors for the PROGRAM, FUNCTION, SUBROUTINE and BLOCK DATA statements; for LOADER directives and code to manufacture a header for program units that don't begin with a header statement.

<u>Interfaces</u>: HEADER routines are called by the front end main loop and indirectly interface the system with console and dayfile messages. HEADER resides on the (0,0), (1,0) and (2,1) overlays.

<u>Data Structures</u> Header defines no data structures, but utilizes several of the managed tables.

<u>Routine Descriptions</u>

a. <u>BKD</u>: BLOCK DATA statement processing. Entered from FEC main loop. BKD provides a default name if the BLOCK DATA block is unnamed. A call is made to DCM for processing and exit is to the front end controller.

b. <u>FCT</u>: FUNCTION statement processing. Entered from FEC main loop. If the function statement is not of the form 'type FUNCTION name ( )', FCT calls STY to set the implicit type. Then DCM and TSB are called to process the statement. Exit is to the front end controller.

c. <u>PPG</u>: PROGRAM statement processing. Entered from FEC main loop. PPG calls DCM and PPA to process the program statement.

d. <u>LCC</u>: Embedded loader directive processing. Handles embedded OVCAP and OVERLAY directives. The entokened directives are recombined in Hollerith form and saved on a managed table. A turple is output to the parse file indicating the existence of the directive. (The directive will later be output as part of the loader input file.) Any errors found will be output. Exit is to the front end controller.

e. <u>PSF</u>: Process special first statement. Called by FEC main loop when the first statement of a program unit is not a header statement. The program unit will be treated as a program with name 'START.'. A warning diagnostic to this effect is published. A pseudo token buffer is dummied to allow the default files 'INPUT' and 'OUTPUT'. PSF calls DCM and PPA to process the 'statement' and exit is to the front end controller.

f. <u>SUB</u>: SUBROUTINE statement processing. Entered from FEC main loop. SUB calls DCM and TSB to process the statement. Exit is to the front end controller.

g. <u>DCM</u>: Display Compiling Message (and a lot more). DCM enters the program unit name in the symbol table, including the attributes known at header processing time. Errors and inconsistencies are diagnosed. An entry in the managed entry point table is made and a header turple is output. Finally, the console/dayfile 'COMPILING type name' message is output. (NOTE: This routine probably ought to be renamed.)

h. <u>PBM</u>: Process buffer length or maximum record length. Called by PPA to process the various constant forms which can represent the above lengths. Provides a binary value of the translated constant and an error indication as required.

i. <u>PPA</u>: Process Program Arguments. Called from PPG and PSF. PPA provides a syntactic processing of the program statement file list, diagnoses errors, enters the named files into the symbol table (including translated buffer and record length information, as applicable) and outputs a file turple for each file. The number of files are counted and, if the system limit is exceeded, a diagnostic is output. At completion of file processing, entry code turples are output and some compiler generated symbols are defined for system externals.

j. <u>PSA</u>: Process Subprogram Arguments. PSA is a general argument processor used to process subroutine, function and entry statement argument lists. PSA handles the building and resolution of the various parameter lists allowed by ANSI77. First it calls SAL to scan the current argument list. This will result in the arguments being placed in the managed argument table. The argument table is then scanned, building a local argument list table. NCM is called to scan the local argument list into the entry point table (managed). If the entry is unique, the entry point table is updated and, in any case, the symbol table entry for the function, subroutine or entry name is updated with a pointer to the associated parameter list.

k. <u>SAL</u>: Scan Argument List. SAL is called by PSA to provide syntactic processing of the argument list and to provide a list of names of the arguments. SAL processes each argument in turn, adding symbols to the symbol table where necessary and diagnosing syntactic and semantic errors which occur. The name is added to a local parameter list and duplicates are diagnosed. If the argument didn't appear in a previous formal parameter list, a formal parameter information table (managed) entry is made. A cross reference table entry is made. Alternate return parameters are noted.

l. <u>TSB</u>: Translate Subprogram Begin. Called from FCT and SUB. Calls PSA to process the argument list. Outputs start of executables turple and makes a symbol table entry for the system exit.

m. <u>WSA</u>: Wrapup Subroutine Arguments. WSA makes symbol table entries as needed for system externals pertaining to formal parameters used as actual parameters.

3.5 KEY: Keyword Statement Processing

Abstract: KEY contains statement translating routines and subroutines for the ASSIGN, CALL, CONTINUE, END, ENTRY NAMELIST, PAUSE, RETURN, STOP, GOTO, IF, ELSEIF, ELSE and ENDIF statements.

Interfaces: KEY is a front end deck and resides on overlays (0,0), (1,0) and (2,1).

Data Structures

KEY defines no data structures, but has communication cells and utilizes several of the managed tables.

Routine Descriptions

a. AGN: ASSIGN statement translation. Since the ASSIGN statement is syntactically awkward, ASL is called to extract the expected label, which is processed by ISL for semantic correctness. The imbedded keyword TO is tested and stripped by ASK and the assign variable is semantically tested by TRV. Any syntactic or semantic errors result in diagnostics. An ASSIGN turple is output and an entry is made into the ASSIGN managed table, combining the assign label and assign variable. Exit to front end controller via PSL.

b. CLL: CALL statement translation. The subroutine name is checked for semantic validity, and a symbol table entry is created (or modified as necessary). The presence of an argument list is determined, and if present, a call to PAR is made to process the list. Upon return from PAR, the turple for the subroutine is output, and if alternate return labels occurred, a GOTO turple for the returns is output. Exit is to the front end controller via PSL.

c. CRL: Call statement return label processing. CRL is a subroutine used by the parser when an alternate return label is encountered in a parameter list. The alternate return label(s) had been kept on the statement label argument table, pending completion of the translation of the call parameter list. CRL makes a generated label for the returns and adds it and all the alternate return labels to the managed argument table.

d. CON: CONTINUE statement translation. CON diagnoses trivial uses of the CONTINUE statement (e.g., as object of a logical if) and exits to the front end controller via PSL.

e. END: END is entered from the FEC main loop and provides some end of program unit functions. If the listing (control statement) options were modified by a universal C$ listing directive (LIST,NONE), attributes and references are turned off. If flow was into the END statement, a return or end turple is output, as applicable. If formal parameters were present, a call to WSA completes their processing. MND is called to materialize namelist dimensions and exit is to the front end controller.

f. ENT: ENTRY statement translation. ENT checks the entry name for semantic correctness and the statement for legal position within the program unit (e.g., not in a program, not in a do loop). The entry name is added to the entry managed table and an entry point turple is output. The argument list is processed by PSA and exit is to the front end controller, via PSL.

g. NAM: NAMELIST statement translation. NAM translates a NAMELIST statement into an entry in the namelist managed table. The group name is syntactically checked and entered in the symbol table. The group header is put on the namelist table. Each variable on the namelist list is tested for validity and entries are made on the namelist table (including dimension information for arrays). Exit is to the front end controller.

h. PAU: PAUSE statement translation. PAU merely calls SPR to translate any appended message and output the proper turple. Exit is to the front end controller, via PSL.

i. RTN: RETURN statement translation. RTN is a semantic testing routine. It checks the type of program unit being compiled to determine if an alternate return form is legal, and if not (and an alternate return as specified) corrective action is taken. If an

ordinary return, a return turple is output. If the return was the appended statement of a logical if, RIT is called to restructure the if (front end optimization). If a legal alternate return (in a subroutine) is present, PJX is called to parse the return clause and output the turple. Exit is to the front end controller, via PSL.

j. STP: STOP statement translation. STP calls SPR to translate any appended message and output the proper turple. Exit is to the front end controller, via PSL.

k. GOT: GOTO statement translation. GOT (with GOA and GOC) translates syntax and semantics of the three types of GOTO statement. Upon entry from the FEC main loop, the nature of the GOTO is determined. GOT processes unconditional GOTOs. ISL is called to semantically test the label and if the GOTO is appended to a logical if, RIT is called to reset the target. Processing of the goto turple is deferred, via the HANGER mechanism. Upon return from the hanger processor, the goto turple is output or a diagnostic reflecting the lack of need for the turple is issued. GOA processes assigned GOTOs. The goto variable is semantically tested and if a label list is present, a loop is used to test syntactic and semantic (via ISL) legality. An assigned goto turple is output and exit is to the front end controller via PSL. GOC processes computed GOTOs. The label list is scanned (but not translated) counting the number of labels. The goto expression is then parsed (via PJX) and the computed goto turple output. Then the label list is translated, using ISL for semantic testing and for each label, a goto label turple is output. Exit is to the front end controller, via PSL.

l. ELS: ELSE/ELSEIF statement translation. Entered from FEC main loop on either ELSE or ELSEIF. Calls FIB to finish off the previous block if 'arm'. If the statement was ELSEIF, exit to ELF. Otherwise, mark the current block structure entry that ELSE occurred and exit to the front end controller, via PSL.

m. ELF: Finish processing ELSEIF statement. ELF calls PAR to translate the logical clause and checks the presence of the keyword THEN. The relational expression returned by PAR is semantically tested and CIM is called to produce an IF megaturple. Exit is to the front end controller via PSL.

n. EIF: ENDIF statement translation. EIF provides semantic processing for the close of a block IF. An illegal nesting of block IF/DO loop is detected, and the offending structure(s) removed (in this case, any unterminated DO loops) by RBE. FBS is called to finish off the block structure and the required generated label turples are output. The block if entry is removed from the block structure table. Exit is to the front end controller via PSL.

o. IFS: IF statement translation. IFS calls PAR to translate the if clause. Upon return from PAR, the nature of the if is determined. Arithmetic ifs are processed by IFL and block ifs by IFT. If the if clause is logical (and the if is not a block if) the appended statement is classified by CST and tested for legality as an if appendage. If legal, CIM is called to output the if megaturple and exit is to the front end controller.

p. IFL: Label (arithmetic) if. IFL checks the label list for correct syntax and semantic (ISL). TP. format operands for the labels are saved and processing is deferred via the hanger. Upon return from hanger processing, the label list is analyzed and the proper (optimal) if turple is selected. Selection is based upon repetition/uniqueness of labels and the fall through possibility provided by the hanger label check. Exit is to front end controller via CUS.RET (hanger processor).

q. IFT: IF ( ) THEN (block IF) processing. IFT calls CIM to output an if megaturple and adds an entry to the block structure table (managed) for the block. Exit is to the front end controller, via PSL.

r. CIM: Construct if megaturple. The subroutine CIM determines if the expression returned by PAR was a simple relational (e.g., IF(A.EQ.B)). If so, that relational expression is used as the first turple of the if megaturple (modified in place on the parse file). If not, the first turple is made and output. In either case, the second turple is made and output, completing the megaturple.

s. FIB: Finish if block. FIB is called by ELSE/ELSEIF statement processor to provide some semantic tests and to update the block structure table entry for the current block. ELSE/ELSEIF statements not in a block IF structure are diagnosed. Illegal nesting of block IF/DO loops are diagnosed and any unterminated DO loops are removed from the block structure table. FBS is called to process the block structure table entry. Generated label and branch turples are output as required.

t. CEM: Check entry point mode. CEM is a routine called by ENT to determine the semantic correctness of an entry point name. A function subprogram type may not differ from its entry type if either are type character. Further, if both are type character, the character length must agree. CEM diagnoses deviations from these rules.

u. MND: Materialize namelist dimensions. MND is called from the END processor. A scan of the namelist table is made. Each namelist group name which appeared in an I/O statement has its list scanned. For each array in the corresponding namelist list, the dimension table is marked to be materialized. This will cause run time dimension table information to be output with the binary.

v. PJX: Parse jump expression. PJX calls PAR to process an expression used to compute an index into a list of labels.

w. C=PJX: Parser interface routine for PJX. Tests the parsed expression for integer, and if not, calls CMR to coerce the mode to integer. The indexed jump operator and the operands are set up for standard turple output and return to PAR is at POP.STD.

x. RIT: Reset if target. RIT is called when a RETURN or unconditional GOTO is appended to a logical IF statement. The branch condition is reversed and a branch is thereby eliminated. Conversion is in place on the parse file. This is a front end optimization.

y. SER: Compile end instructions. SER is called by END (and RTN when in a program). Sets up for end routine processing and calls SRS for compilation of instructions and AP list.

z. SPR: STOP/PAUSE compilation. SPR is called by PAU and STP to translate the message argument, if present. The constant argument is tested for legality and a TP. format operand is set up for SRJ which is called to output the RJ and AP list turples.

aa. SRJ: Select RJ. SRJ is called by SER and SPR to output turples for the stop/pause/end functions. EAL is called to output required AP list turples and the proper system routine name is selected for the RJ turple which is output.

3.6 CDDIR: C$ Directive Processing

<u>Abstract</u>: CDDIR translates the C$ COLLATE, DO, IF, ELSE, ENDIF and LIST directives.

<u>Interfaces</u>: CDDIR, based upon the status of control statement options and translated C$ directives, modifies working copies of the option control cells. This can affect source listings, compilation of portions of a program unit, code generation and character collating weights. A front end routine, CDDIR resides on overlays (0,0), (1,0) and (2,1).

<u>Data_Structures</u>

CDDIR defined data structures consist of tables used to translate the directive parameters and are described by the relevant routines.

<u>Routine_Descriptions</u>

a. <u>COL</u>: Translate COLLATE directive. COL verifies the syntax of the COLLATE directive and outputs a collate directive turple and updates the working copy of the collate control cells. Exit is to the front end controller.

b. <u>DO</u>: Translate DO directive. Calls TCP to translate the directive parameters. Upon return, the status of the parameters is tested, and the relevant working status cells are updated. A do directive turple is output. Exit is to the front end controller. The table FW.DO contains the directive parameter texts and return status cells.

c. <u>IFD</u>: Translate IF directive. IFD defines the format of the C$IF table. IFD processes the FTN5 conditional compilation directive IF. The FEC bypass stage is tested to determine the status of conditional compilation. (If in bypass mode, the directive still must be stacked, but marked inactive so that else and end if directives associated with it will not improperly start or stop compilation.) If the directive is active, PKX is called to parse the if clause. The expression must be reduced to a logical constant. If the expression is .true. (or in bypass mode already) exit is to the front end controller. If the expression is .false., compile mode is set to bypass and exit is to the front end controller.

d. ELSE: Translate ELSE directive. ELSE checks the syntax and semantics of the ELSE directive. If the C$IF table entry is marked inactive, nothing is done. If the group is active, the opposite active taken by the corresponding IF directive is performed. If currently in bypass mode, the previous stage is restored and compilation resumes. If currently compiling, bypass is evoked. Exit is to the front end controller.

e. ENDIF: Translate ENDIF directive. ENDIF checks the syntax and semantics of the ENDIF directive. The C$IF table entry for the group is removed. If the current group is inactive, exit is to the front end controller. If the compiler is in bypass mode, the previous stage is restored and compilation continues. Exit is to the front end controller.

f. LIST: Translate LIST directive. LIST calls TCP to translate the LIST directive parameters. A table, FW.LIST, containing directive parameter texts and return status cells is used by TCP. Upon return from TCP, the list parameter status cells are tested and the corresponding working copies of the control statement cells are updated. For any LIST directive to be honored, the corresponding control statement must have been selected. Some of the options require action beyond updating the working cells. LIST(O) requires an object list directive turple. LIST(R) requires reset of the ERT switch. LIST(S) may require listing the deferred buffer.

g. GDL: Get directive label. GDL is called by the conditional compilation directives to obtain the label, if present. The label is made available for the C$IF table entry.

h. TCP: Translate C$ parameters. Using the parameter table provided, TCP translates the parameter list for a C$ directive. The parameters requested are tested for legality, and if of the P=C form, PKX is called to translate the parameter constant. Irregularities result in diagnostics. Processing continues until the parameter list is exhausted.

3.7 DATA: DATA Statement Processing

Abstract: DATA provides routines for the translation of the DATA statement.

Interfaces: DATA uses the IO list processing routines to translate the data variable list. DATA is a front end routine and resides on overlays (0,0), (1,0) and (2,1).

Data Structures

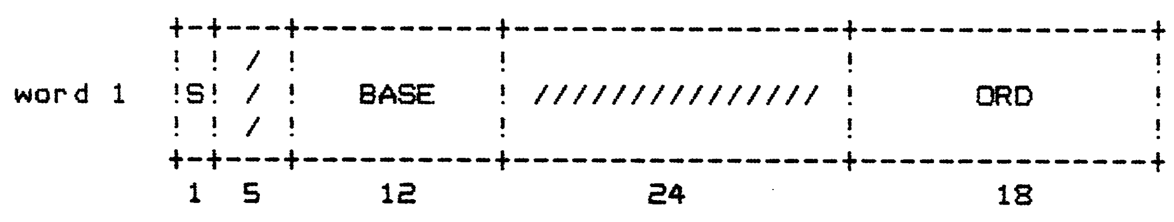
a. MACROS/MICROS: DATA defines local macros and micros to aid the translation of the data statements. The macros and micros relate to token interpretation.

b. Cells: The DATA cells contain information flags and values pertaining to the DATA processing status.

c. DL.: Describe/define for the data initialization list pointer table. Contains the pointers to the start of a data variable list and its associated data constant list.

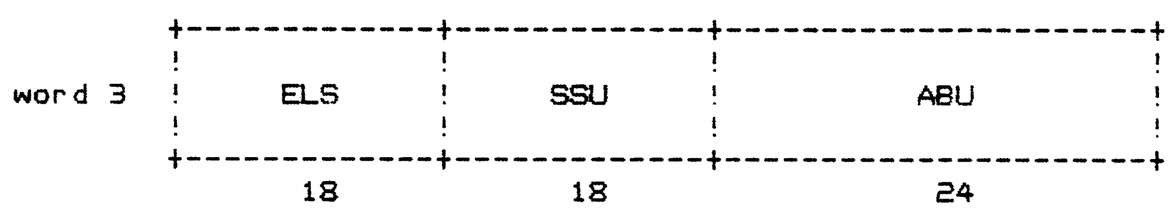
d. DVI: Data variable information. As each data variable item is processed, a three word entry in DVI cells is made with the following format:

word 1



word 2 WB. symbol table word (see global data structures)

word 3



S	-	single/double precision flag
BASE	-	ordinal of base member
ORD	-	ordinal of symbol table entry
ELS	-	element size
SSU	-	substring size
ABU	-	additional bias

e. S.: Construction of output block control words.

a. DATA.E: DATA processor error exit. DATA.E provides generalized error recovery from diagnostics noted during processing a DATA statement. The relevant data information tables are trashed and any constants generated for the data statement in error are removed from the constant table. If any implied do's were generated, their block structure table entries are removed. Exit is to the front end controller.

b. DATA: DATA statement translation. DATA is the main loop for translating a data statement. First, a call to BDL groups the statement into pairs of data variable list - data constant list. Then DATA loops to process each pair of lists, first translating the constant list (BIT), then translating the variable list (PVL), then publishing the information to the intermediate language file via PDI. Exit is to the front end controller (via the cleanup function of DATA.E).

c. BDL: Build data list pointers. BDL scans the DATA statement, via STD, using the constant list slashes as delimiters. The slashes are replaced by end of statement tokens (in the token buffer) to aid list processing routines. A data initialization list pointer table entry is made for each variable-list constant list pair. Gross syntax errors are detected (mismatch of variable and constant list, null lists, etc.). BDL is called once per data statement by DATA.

d. STD: Scan to delimiter. STD is called from BDL to provide the actual scan of the lists. STD examines each token in the token buffer, until the desired delimiter is found (or end of statement, an error). The address of the delimiting token is returned to the caller. If a left parenthesis token is encountered, STD skips to the matching right parenthesis token, speeding the scan.

e. BIT: Build data item table. BIT builds a constant pointer list (on the data item table) for a single data constant list. Each item in the constant list is translated into the proper binary form. (Hollerith and character constants are already in binary form, so the pointer is merely extracted from the token.) The constants are scanned into the constant table, and the pointer to the binary is used to form the data constant table entry (mode and length are the other fields). The constant delimiter is tested and processing continues or terminates (comma or end of statement marker). Unsigned integer constants receive special processing. When found, the delimiter is tested for a replication factor token. If such a token is present, the constant is used to form a replication entry in the data constant list. BIT calls several subroutines to process the variety of constants which may appear in the constant list.

f. PVL: Process data variable list. PVL is a subroutine that controls translation of the data variable lists. Called from DATA once for each data variable list present. PVL calls CVL (IO, 3.11) to process the variable list. The resultant list turples are then interpreted by SED (CONRED, 3.13) to simulate execution. This latter step is required by the general form of implied loops allowed in data variable lists.

g. C=DVL: Parser interface for data variable list items. C=DVL determines the variable legality (must be scalar, array or array element; cannot be blank common, etc.). If the list item is within an implied do, collapse information is merged in, as applicable. A data variable turple is output to the parse file.

h. EDI: Emit data initializations. EDI is called by the CONRED simulation routines. Calls SDV to set up the data variable item and NIC to organize linear progressions.

i. NIC: Output linear index pattern. NIC is called by EDI to provide the actual simulation of data variable list implied do's. NIC is entered with the implied loop's trip count determined. This will be the number of data items consumed by the loop. NIC analyzes the implied loop, trying for contiguous blocks of storage, and in conjunction, tries for replicated data items. If such conditions exist, NIC will reform the replications to fit the contiguous storage. The end result will be data table entries to correspond to the loop and its associated data. The output (to the data table) formatting is performed by subroutines described below.

j. OSH: Output scalar header. OSH updates the S. cells for data header initialization. If a previous data header is present, it is finished by a call to UPH. The length value associated with the data variable item is converted from elements to storage units (words for all but type character, which is character count). The header information known is output to the data information table (DA. format).

k. ORH: Output replication header. ORH calls OSH to provide the DA. header word and then creates the DB. header for replication information. The S. cells are updated to indicate the status of the block in progress.

l. OVI: Output value of item. OVI enters the binary of a data item list constant into the data information table. If mode coersion is required, CMV performs this service.

m. SDV: Set up data variable. SDV analyzes the symbol table entry for a data variable list item and sets up the DVI cells accordingly.

n. UPH: Update previous header. UPH tests the S. cells to determine if the previous data block was complete. If so, no action is taken. Otherwise, the information in the S. cells is used to update the header word(s) on he data information table. The S. entry is marked as complete.

o. ADC: Add constant to data item table. ADC is called by BIT to process constants encountered in a data constant list. ADC process already converted constants (Hollerith and character) by extracting the constant table pointer from the token. Arithmetic constants are converted by a call to TNK and scanned into the constant table by ASI.

p. ASC: Add symbolic constant to data table. ASC is called when a symbolic constant (parameter) is encountered in a data constant list. The symbol table entry for the constant is anlyzed and a DI. format data item table entry is produced.

q. ASI: Add scalar item. ASI takes the binary of a converted constant and scans the entry into the constant table. ASI returns a DI. format data item table entry.

r. CFC: Check for complex constant. CFC is called when a left parenthesis is followed by a token which can be part of a constant. CHC is called to convert the first half, comma delimiter is tested and CHC is called to convert the second half of the constant. The resultant binary of the real and imaginary parts is returned (or a failure indication).

s. CHC: Convert half of complex constant. CHC is a coroutine of CFC. CFC guides the constant conversion process, keeping track of sign, decimal points, etc. DEC is called to convert a constant, CSC to fetch the value of a symbolic constant and KCV to convert an integer to real. If nonconstant elements are found, exit is to the CFC failure processor.

t. CMV: Coerce mode of value. CMV is called by data variable list processing and PARAMETER statement processing to convert the constant mode to the mode of the associated variable. Inconsistent conversions are daignosed (e.g., logical - nonlogical). The new constant is entered in the constant table and a DI. format entry is returned.

u. CPR: Complex parameter reference output. Called by CFC to output cross reference information when a PARAMETER is used as half of a complex constant. Needed for timing of the reference output.

v. <u>CRL</u>: Close out replication list. CRL is called when BIT notes the end of a list of replicated data items. The data item table replication header is updated to reflect the length of the list of data constants.

w. <u>CRC</u>: Check repeat constant. CRC is called by the data constant list processor when a replication constant is noted. The constant is semantically tested for type and value and irregularities are diagnosed.

x. <u>CSC</u>: Check for symbolic constant. CSC is called by BIT when a variable token is encountered in a data constant list. A symbol table scan is made, and if the name is not that of a symbolic constant, failure is flagged. If the name was that of a symbolic constant (parameter), CSC returns information pertaining to the constant value.

y. <u>GNI</u>: Get next item. GNI is called by NIC when processing the data variable list match up with the data item list. GNI tests the current replication state, and returns the proper constant pointer (from the data item table) and decrements the replication count, as necessary. If the next item is not in a replication group, the item is returned and the table count is adjusted accordingly. If replication was in effect, and the current GNI call exhausts the list, that is noted.

3.8 DECL: Declarative Statement Processing.

Abstract: DECL contains statement translating routines for the COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, INTRINSIC, LEVEL, PARAMETER and SAVE statements. DECL also contains subroutines used for end of declarative phase processing.

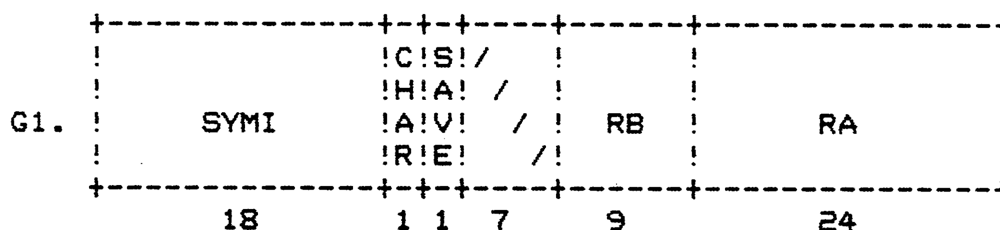
Interfaces: DECL is a front end deck and resides on overlays (0,0), (1,0) and (2,1). The front end controller interfaces with DECL to provide declarative/executable phase transition.

Data Structures:

DECL uses many of the global data structures. The usage is described in the relevant routine descriptions. Data structures defined by DECL are:

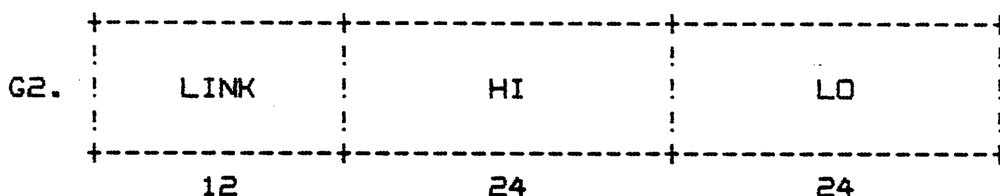
a. DIMI: A storage area containing enough space for a dimension header and dimension descriptors up to the maximum dimensionality allowed. This area is used as scratch storage by routines not involved in dimension processing.

b. G/E Table: Galler/Fisher equivalence table. Used at end of declarative time equivalence processing.



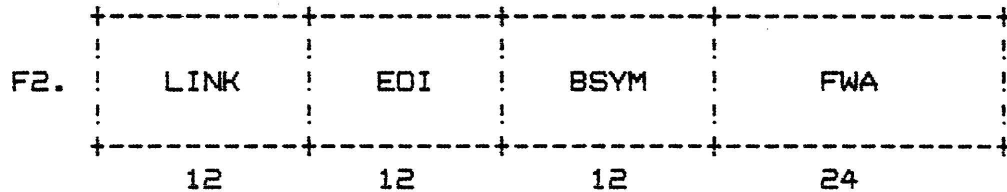
```
     +--------------+-+-+----+------+---------------+
     !              !C!S!/   !      !               !
     !              !H!A! /  !      !               !
G1.  !     SYMI     !A!V!  / !  RB  !       RA      !
     !              !R!E!   /!      !               !
     +--------------+-+-+----+------+---------------+
            18       1 1  7     9           24
```

SYMI: Index of symbol table WB. entry
CHAR: Character entity
SAVE: Saved variable
RB : Relocation block
RA : Relative address



```
     +------------+----------------+----------------+
     !            !                !                !
G2.  !    LINK    !       HI       !       LO       !
     !            !                !                !
     +------------+----------------+----------------+
           12             24               24
```

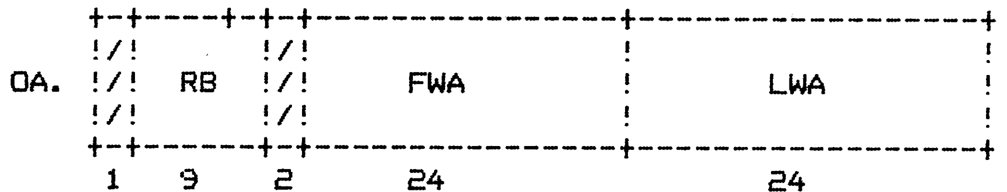
LINK: Index in equivalence table
HI : Space needed above root
LO : Space needed below root



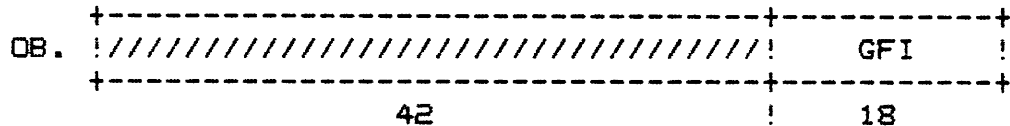
LINK: Index in equivalence table
EOI : Index in equivalence overlap table
BSYM: Index of base member ST WB.
FWA : FWA of equivalence class

c. Equivalence Overlap Table:

Used in end of declarative equivalence processing.



RB : Relocation base of equivalence class
FWA: FWA of class
LWA: LWA of class



GFI: Index of class root in equivalence table

Routine Descriptions:

a. CMN: Common statement translation. Entered from FEC main loop, CMN processes the common statement, checking for correct syntax and some semantics (doubly declared in COMMON, dummy argument). As necessary, entries are made in the common block name table (T.BLK). The common variables are entered in the common item table (T.COMM). Arrays declared in a common statement are processed via a call to DIR. Exit is to the front end controller.

b. DIM: Dimension statement translation. DIM is entered from the FEC main loop when a dimension statement is encountered. DIM merely loops on the array names encountered, calling DIR to process the dimensions. Exit is to the front end controller.

c. <u>DIR</u>: Process dimensioned variable. DIR is called by the common, dimension and type declaration statements to process the dimensionality of a variable. The variable name is tested for semantic legality. The DIMI (dimension holding area) is initialized and DIS is called to process the dimensions. Upon return from DIS, the dimension entry is finished and the dimension table entry is scanned into T.DIM.

d. <u>DIS</u>: Assemble dimension subscripts. DIS is called by DIR to control processing of dimension subscripts. DIS loops through the dimension list, calling CDB to process individual dimension bounds. Each loop will produce a D1. and D2. entry for the relevant dimension. If an explicit lower bound is present, two CDB calls are necessary. If the bounds are constant, the span is calculated and the product of spans field is updated. If either (or both) bounds are variable, the necessary turples for span calculation are output to T.VDIM and a T.VDI entry is made. The header is marked accordingly.

e. <u>CDB</u>: Compile dimension bound. Called by DIS, CDB processes dimension bounds by calling PAR to parse the expression. Upon return from PAR, the TP. form of the expression is converted to D2. dimension bound format.

f. <u>C=DDB</u>: The parser interface routine for dimension bound expression processing. Called when a comma or right parenthesis is encountered during bounds expression parse (the ARGMODE function, see PAR, 3.12). If the bounds expression was constant, exit is to PAR with no processing. If an error was noted, a dimension bound constant one is made (so that processing can continue). If the dimension bound was a variable only, a VD. store turple is made (on T.VDIM) and that TP. form is returned. If the bound was an expression, the turples are moved from T.PAR to T.VDIM and a VD. store turple is added to T.VDIM. Exit is to PAR.

g. <u>OVP</u>: Output vardim product of spans. OVP is called by DIR when adjustable dimension bounds occurred in a dimension declaration. OVP makes multiply turples, as needed, to provide the size of the array (in elements). The turples are added to T.VDIM and a VD. store turple ends the sequence. The partial product of spans is replaced by the T.VDIM pointer. A T.VDI entry is made, as necessary.

h. OVS: Output vardim store turple. OVS constructs the store turple into the VD. cell. All the turples making up the vardim entry are analyzed to eliminate duplicates. If the current entry is unique, the store turple is output and a T.VDI entry is made.

i. OVT: Output vardim turple. OVT outputs an add or subtract turple required in calculating the variable span. The result of the turple is returned in TP. format.

j. EQS: EQUIVALENCE statement translation. EQS is called from the FEC main loop to provide syntactic and some semantic translation of equivalence statements. EQS enters equivalenced items on T.EQUS for further processing at end of declaratives. The classes are as defined by the user. Merging of classes and common interface is deferred until end of declaratives.

k. EXT: EXTERNAL statement translation. EXT is entered from the FEC main loop to provide syntactic and semantic analysis of the EXTERNAL statement. Symbol table entries are made/updated as required. Exit is to the front end controller.

l. INT: INTRINSIC statement translation. INT is entered from the FEC main loop to provide syntactic and semantic analysis of the INTRINSIC statement. Functions named in an INTRINSIC statement must be defined by FTN5 to be intrinsic and if previously explicitly typed, the type must be confirming. Symbol table entries are made/updated as necessary. Exit is to the front end controller.

m. LVL: LEVEL statement translation. LVL is entered from the FEC main loop to provide syntactic and some semantic analysis of the LEVEL statement. The level number is tested for legality and the associated list is analyzed. Common block names are marked with level information (and T.BLK entries are made as necessary). Variable names cause the relevant symbol table entry to be made/updated. Exit is to the front end controller.

n. PRM: PARAMETER statement translation. PRM is entered from the FEC main loop to provide syntactic and semantic analysis of the PARAMETER statement. The parameter name is semantically tested and the constant expression is translated via PKX. If necessary, the resultant constant is converted to the mode of the parameter. The constant is entered in T.CON and the symbol table entry for the parameter is updated to reflect the location of the constant. Processing continues for the entire list. Exit is to the front end controller.

o. SAV: SAVE statement translation. SAV is entered from the FEC main loop to provide syntactic and some semantic analysis of the SAVE statement. Common block name table entries are updated with save information. Symbol table entries for dummy arguments and local variables are updated. Exit is to the front end controller.

p. PCD: Process close of declaratives. PCD is called from the FEC main loop when the current statement entokened by LEX is the first non-declarative statement of the program unit. PCD consists of subroutine calls to end of declarative semantic routines and trashes some tables which are no longer needed.

q. APT: Assign pointer tags. APT is called from PCD to process common block table (T.BLK) entries. When APT is called, the sizes of the various common blocks is known (equivalence information has been processed) and level information at the block level is known. APT assigns blocks of levels 2 and 3 to LCM/ECS depending on the machine configuration. The length of the blocks is semantically tested. All blocks except the program block are processed.

r. ASL: Assign level usages. ASL is called from PCD to process symbol table variables as to level and whether ECS/LCM is to be assigned. If so, the corresponding attribute bit is set.

s. CCC: Check character common block. CCC is called from PCD. If character variables were declared, the common block table (T.BLK) is scanned to determine if any common blocks consist of character variables. If so, the CB. word of the T.BLK entry is finished and if mixed character and non-character variables were present, a diagnostic is output.

t. CCL: Coordinate common/level information. CCL is called from PCD to propagate the T.BLK level information to the members of each common block. The level information for each block is determined and then the T.COMM chain is scanned, and the symbol table entry for each member is updated.

u. CCS: Convert character symbol. CCS is called by PCD to convert the character offset to a word offset and beginning character position (WC.).

v. DCS: Diagnose common/save variables. DCS is called by PCD to diagnose redundant specific SAVE declarations when universal save was declared.

w. DSRT: Sort double entry table. A shell sort (JACM 1960) algorithm. Called by EQU to sort the equivalence overlap table (T.EOT). Provided by comdeck COMFDST.

x. EQU: End of declarative equivalence processing. EQU is called from PCD to provide final equivalence processing. The T.EQUS entries, formed during translation of the EQUIVALENCE statement(s) are reformatted for processing. Substring references and subscript references are resolved and converted to offsets. The equivalence classes are formed into a tree structure (via the Galler-Fisher algorithm [CACM 7:5, 301-303]) with the final root being the base member, and other members being represented as offsets from the base. The classes are then checked for common block membership (and for semantic errors in that relation) and, where applicable, the individual equivalence classes are merged to form larger classes. This will result in modification of the base member and offset for one of the classes. Upon completion of the merge, address (relative) are assigned and the relevant symbol table entries are updated.

y. ACV: Assumed length character vardim processing. ACV is called by PCF to output T.VDIM and T.VDI entries for formal parameters of type CHARACTER and assumed (passed) length. ACV tests the formal parameter for assumed length character and if so, outputs a GPL turple and a store turple, via OVT and OVS.

z. MCA: Make relative common assignments. MCA is called from PCD to provide initial end of declarative common block processing. The common items on T.COMM are scanned, based on the starting chain in T.BLK. Relative addresses are assigned to each member, and the size of each common block is determined. Semantic tests are made for CHARACTER/non-CHARACTER variables assigned the same block, and a null common block declared in SAVE/LEVEL is diagnosed. The relevant symbol table entries are updated with the block relative addresses.

aa. MFR: Mark function as referenced. MFR is called from PCD to set the relevant VALUE. symbol table entry as must be defined. For function subprograms only.

bb. PCF: Process CHARACTER/formal parameter interaction. PCF is called by PCD to provide special processing of type CHARACTER formal parameters. A function subprogram of type character has its associated VALUE. treated as a formal parameter. In fact as formal parameter number 1. The remaining formal parameter numbers are adjusted (the ordinals) and then tested (via ACV) for assumed length. As necessary, new T.VDIM and T.FPI entries are made.

cc. PSC: Propagate save bit through common. PSC is called from PCD after completion of equivalence processing (all common block members are known). TSC scans the symbol table, and all members of a common block which appeared in a SAVE statement have their symbol table entry updated with the save indication.

dd. PKS: Parse constant substring. PKS is called by EQS when a substringed variable occurs in an equivalence statement. The constant substring expression is parsed (via PIX) and the results are returned as first element, last element. Syntax errors are diagnosed.

ee. SAS: Scan array sizes. SAS is called from PCD upon completion of all other declarative processing. The symbol table is scanned and all arrays are tested for legal size, in context of the array assignment in storage (LCM/ECS arrays can be larger than SCM arrays). Errors are diagnosed.

ff. VDP: Variable dimension processing. VDP scans the symbol table, looking for variables which appeared in dimension bound expressions. If such a variable is in common or a formal parameter, the VDS bit is unset. The variable is tested for type integer, and if not, a diagnostic is issued. The T.VDI table is processed to eliminate front end information.

3.9 TYPE: Explicit and Implicit Type Declarations

Abstract: TYPE contains statement translators for the explicit type statements BOOLEAN, LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX and CHARACTER, and for the IMPLICIT statement.

Interfaces: TYPE is a front end deck and resides on overlays (0,0), (1,0) and (2,1). The header statement processor interfaces with TYPE to process statements of the form: type FUNCTION name().

Data Structures:

TYPE utilizes the symbol table structures but defines only one small structure:

TASK: TASK is the subkeyword table for the implicit type declaration processing:

```
+--------------+-----------+-------+---------------+
!     MODE     !///////////!  LEN  !      KEY      !
+--------------+-----------+-------+---------------+
       18           17         7           18
```

MODE: Value of mode (M.type)
LEN : Keyword length (bits)
KEY : Pointer to literal representing keyword

a. TYP: Process explicit type. The individual keyword processors are entered from the front end controller main loop. The M.type value associated with the relevant type is set (and the length for type CHARACTER) and all types are processed by TYP. If the explicit type statement is the first statement of the program unit, a test is made to determine if the statement is a function header. If so, exit is to KW=FUNC (with the type information) and TYP is done. Otherwise, PSF is called to 'make' a header and control returns to TYP. Each item in the type list is processed as follows: If the symbol isn't in the symbol table, it is entered, with the relevant type information. If the item is dimensioned in the type declaration, DIR is called for that processing. If the item is already in the symbol table, semantic checking is performed to insure that a symbol is not doubly typed, that the type of an intrinsic function isn't altered and that a symbol which can't be typed (e.g. program name) is trying. If the symbol being typed is the function subprogram name, the main entry symbol table information is modified. Type CHARACTER lists require special processing, because the length can

be overridden for individual symbols. After each CHARACTER symbol is translated, its associated length is translated, its associated length is translated via CCL. Syntax errors are diagnosed, as well as the semantic checks above.

b. IMP: Translate the IMPLICIT statement. IMP is called from the FEC main loop to provide syntactic and semantic checking of the IMPLICIT statement. Implicit processing consists of converting the letters and ranges of letters in the implicit declaration into a bit mask and then merging that information into the NAT.TYP table (in FEC, see STY[3.1]). Duplicated implicit declarations are diagnosed, as are invalid range declarations. For type CHARACTER, the associated length is stored in NAT.LEN.

c. CCL: Check character length. CCC is called when processing CHARACTER symbols to translate the constant length expression (or assumed length indication). Illegal lengths are diagnosed as well as syntax errors.

d. CSK: Crack subkeyword. CSK is an interface routine for the implicit processor. It is called to determine that the implicit type defined is in fact, a legal type. CSK returns the type value (M.type) and character length when relevant.

3.10 FMT: Format Statement Processor

Abstract: FMT contains routines to translate the FORMAT statement.

Interfaces: FMT is a front end deck and resides on overlays (0,0), (1,0) and (2,1).

Data_Structures:

FMT uses T.FMT as its output and defines the following data structures:

a. FMT=TOK: The token/character mapping and branch table.

```
+-------------------------+---+-+--------------+
!                         !/  !F!              !
!           DPC           ! / !I!     ADDR     !
!                         !  /!X!              !
+-------------------------+---+-+--------------+
            36                 1        18
```

DPC : Display code for the token
FIX : Indicates fixed character string
ADDR: Address of token processor

b. FMTJT: Jump table for alphabetic characters (gleaned from an O.VAR token). A packed table of offsets, four per word.

c. ES.: Edit status word. The edit status word (EDSTA) is used to guide semantic processing of a FORMAT statement. Tests of EDSTA can determine what portions of an edit descriptor have already been processed and thus, what is legal or required subsequently.

```
    +------------+-+------------------------+------+
    !            !T!                        !      !
ES. !     TB     !G!          ATTR          !  SC  !
    !            !P!                        !      !
    +------------+-+------------------------+------+
          18      3           33               6
```

TB : Token buffer pointer to initial token
TGP : Tab group (see below)
ATTR: Attributes (see below)
SC : Status codes (see below)

TGP (Tab group bits). On indicates current descriptor is:

T : T
TL : TL
TR : TR

ATTR bits. On indicates current descriptor:

HOL:	is a Hollerith/character descriptor
SF :	contained a scale factor
SP :	allows a scale factor
SGN:	contained a sign (+ or -)
RPT:	was preceded by a repeat count
WF :	had a width field
WR :	requires a width field
DES:	has been fixed (the descriptor letter occurred)
EXP:	has an exponent
EF :	has an exponent field
EP :	allows an exponent
PER:	contained a period
PP :	allows a period
PR :	requires a period
ERR:	contained an error
FIN:	can have no further fields

SC (state code) bits. Semantic bits which are set to inform the processors what portion of the edit descriptor has been processed.

IS :	initial token state
SS :	scale factor state
RS :	repeat count state
DS :	edit descriptor state
WS :	width field state
MS :	M or D field state

d. T.FMT: Entries on T.FMT consist of the blank padded format statement number (DPC) followed by the blank squeezed format descriptor (as many words as necessary). The last word is blank filled. All Hollerith and character descriptors are converted to nH format.

Routine Descriptions

a. FMT: FMT is entered from the front end controller main loop to process FORMAT statements. FMT processes the statement label (or diagnoses its absence), initializes T.FMT and the format build area and initializes the FMT master loop for processing the FORMAT descriptors. Exit is fall through to FMT=NX.

b. FMT=: FMT= is a group of small routines used to process T.TB token which make up the FORMAT statement descriptors. A brief description of the routines follows:

FMT=NX: The controller for the FMT master loop. Fetches the next token, determines the nature of the token and evokes the proper processing routine(s), using FMT=TOK.

FMT=COM: Process a comma. Diagnoses extraneous commas and finishes processing of any pending edit descriptor via AED.

FMT=COL: Process a colon. Finish processing of any pending edit descriptor via AED. Exit to FMT=SL to attempt removal of legal, but unnecessary comma.

FMT=SL: Process slash. (Also the concatenation symbol, which is treated in FORMAT as two slashes). Finish any pending edit descriptor via AED and restart the record length count. Attempt removal of legal, but unnecessary comma.

FMT=PER: Process a period. A semantic check of the legality of a period in the current edit descriptor (existence and placement) and detects the multiple use of a period in a single edit descriptor.

FMT=PL:
FMT=MI: Process a plus or minus. Test the legality of a sign in the current edit descriptor.

FMT=LP: Process a left parenthesis. Finishes pending edit descriptor via AED. Updates parenthesis level (and tests limit) and establishes a repeat count for the level just started.

FMT=RP: Process a right parenthesis. Finishes the pending edit descriptor via AED. The just completed group has its length multiplied by the group repeat count. Length is tested via CRL. Parenthesis level is decremented and the total length of the just completed inner level is merged into the next outer level.

FMT=QHO: Process quoted Hollerith string token. Diagnoses as non-ANSI, then treats as character/Hollerith.

FMT=RLC: Process R"" or L"" token. Formats a diagnostic string and takes error exit.

FMT=CHA: FMT=HOL: Process character or Hollerith token. The lexical scanner has already merged the value of these constants into T.CON. FMT=CHA/HOL determines the number of relevant characters and builds an nH prefix (all such constants in a FORMAT statement are treated as Hollerith) and then copies the content to the format build area.

FMT=EOS: End of statement processing. Diagnoses badly ending statements. Blank pads the last word of the FORMAT descriptor, as necessary. The completed FORMAT descriptor is copied to T.FMT and T.CON is reset to its initial state at the start of FORMAT processing (to eliminate constants which occur only within the FORMAT descriptor). Exit is to the front end controller.

FMT=ILL: Output a diagnostic for a token which is not legal in a FORMAT descriptor.

FMT=CON: FMT=VAR Process a constant or variable token. This is an interface to FMT's inner loop, which bursts the constant/variable information carried in the tokens. The string is added to the FORMAT build area via PFC and the edit descriptor status is updated as necessary. Exit is fall through to FMT.NX.

c. FMT.: FMT. is a group of routines which process the edit descriptors contained in a variable/constant token. A brief description of these routines follows:

FMT.NX: The inner loop controller. FMT.NX extracts the current character to be processed (or exits to FMT=NX when no characters remain) and determines the proper routine using FMTJT.

FMT.ED: General processing of edit descriptors. Updates the edit descriptor status words and diagnoses irregularities. Exits to the relevant processor.

FMT.R: FMT.O FMT.Z FMT.A FMT.D FMT.F FMT.E FMT.G FMT.I FMT.L These routines merely set the proper edit descriptor status bits for updating the status word. All of these descriptors allow repeat counts (but do not require) and thus require no special processing.

FMT.B: Process the BN or BZ descriptor, when encountering a B as edit descriptor. Diagnose irregularities or accept and mark as finished.

FMT.P: Scale factor processing. Determine legality of the scale factor syntax and mark the existence in the status word.

FMT.S: Process the S, SP or SS edit descriptors. Determine the edit descriptor involved and diagnose illegal conditions.

FMT.T: Process the T, TL or TR edit descriptors. Determine legality and set status word accordingly.

FMT.X: Process the X edit descriptor.

FMT.INV: Any alpha character which is not a legal edit descriptor evokes this routine to issue a diagnostic.

FMT.DIG: This routine processes numeric portions of the FORMAT descriptor. The DPC form of the constant is converted to binary. The edit descriptor status word is inquired as to the nature of the current constant, and when determined, the status word is updated, diagnostics are issued as required, and the value of the constant is saved, as necessary.

d. AED: Analyze edit descriptor. AED is called when an edit descriptor is completed. Based upon status bits set by the various processors of FMT= and FMT., the edit descriptor is tested for completeness and lack of extraneous components. The record length is updated by CRL, as required.

e. CRL: Check record length. Determines if described record exceeds device capabilities. Issues diagnostic as required.

f. PFC: Process format character(s). The characters represented by (or contained in) a token are merged into the current build word. When a build word fills, it is added to the build area and a new word is started.

g. RED: Restart edit descriptor status. Called when a lack of punctuation occurs. Resets the edit descriptor status to the initial state.

3.11 IO: Input/Output Statement Processing. **D10**

Abstract: IO contains routines for translation of the input/output keyword statements, and parser interface routines to process I/O lists.

Interfaces: IO is a front end deck and resides on overlays (0,0), (1,0) and (2,1). The routines in DATA (3.7) use the IO list processing routines to process DATA statement variable lists.

Data Structures:

IO uses the following managed tables heavily: T.PAR, T.IOARG. For list collapse, special T.TB token formats are used (see A.2.1). Additionally, IO defines the following data structures:

a. Cells: Various status cells are kept by IO, relating to the type if I/O statement involved.

b. S.IOCALL: A table of names of FCL routines which will provide the required run time I/O functions. The table is ordered, and is accessed based upon combinations of the cell values set during the course of I/O statement translation.

c. FW.CTL: Table of I/O control codes. FW.CTL is created by rewriting the ICDEF macro and using comdeck COMSIOC. Format is:

```
+-----------+-------------+-------+----------+
!   ADDR    !/////////////! LEN   !   KEY    !
+-----------+-------------+-------+----------+
     18           17      !   7        18
```

ADDR: Address of control code processor
LEN : Length of control code (name)
KEY : Pointer to literal representing control code.

Routine Descriptions:

The routines in IO are organized as follows: I/O statement translators, translation support routines, parser interface routines (for translation), I/O list processing routines, implied do routines.

a. BACKSPACE: Entered from the front end main loop when a BACKSPACE statement is encountered. Sets the S.IOCALL indicator to 'backspace' and exits to FPS.

b. CLOSE: Entered from the front end main loop when a CLOSE statement is encountered. Sets the S.IOCALL indicator and the control code legality mask to 'close' and exits to FMS.

c. ENDFILE: Entered from the front end main loop when an ENDFILE statement is encountered. Sets the S.IOCALL indicator to 'backspace' and exits to FPS.

d. INQUIRE: Entered from the front end main loop when an INQUIRE statement is encountered. Sets the S.IOCALL indicator and the control code legality mask to 'inquire' and exits to FMS.

e. OPEN: Entered from the front end main loop when an OPEN statement is encountered. Sets the S.IOCALL indicator and the control code legality mask to 'open' and exits to FMS.

f. REWIND: Entered from the front end main loop when a REWIND statement is encountered. Sets the S.IOCALL indicator to 'rewind' and exits to FPS.

g. FMS: Translate file manipulation statements (OPEN, CLOSE, INQUIRE). FMS provides common processing of the control lists for the file manipulation statements. PKC is called to translate the control item list (using the provided legality mask). OST is called to output a default skip turple, as necessary. SFP is called to set the file property bits. A call to TSX enters the relevant I/O routine name into the symbol table. The I/O routine call is output by IOJ. If necessary, a skip label turple is output and exit is to the front end controller, via PSL.

h. FPS: Translate file positioning statements (BACKSPACE, REWIND, ENDFILE). FPS translates the various forms of the file positioning statements, calling CUD and PKS as required to translate the control list items. OST is called to output a default skip turple, as necessary. TSX enters the relevant I/O routine name in the symbol table. IOJ is called to output the I/O routine call turple. A default skip label is output, as required, and exit is to the front end controller, via PSL.

i. PRINT: Entered from the front end controller main loop when a PRINT statement is encountered. Sets the S.IOCALL indicator, the direction indicator and the default file name to 'output'. Exits to PIC.

j. <u>PUNCH</u>: Entered from the front end controller main loop when a PUNCH statement is encountered. Sets the S.IOCALL indicator and the I/O direction indicator to 'output'. Sets the default file name to 'punch'. Exits to PIC.

k. <u>READ</u>: Entered from the front end controller main loop when a READ statement is encountered. Sets the S.IOCALL indicator, the I/O direction indicator and the default file name to 'input'. Depending on the form, exits to PIC or PEC.

l. <u>WRITE</u>: Entered from the front end controller main loop when a WRITE statement is encountered. Sets the S.IOCALL indicator and the I/O direction indicator to 'output'. Exits to PEC.

m. <u>PIC</u>: Process implied control list. PIC processes formatted (including free format) I/O statements which use a default file. The supplied default file name is placed to T.CON via NCS. The first item in the I/O list is assumed to be the format designator and is processed by PFN. The remaining list items are processed by LST, the exit.

n. <u>PEC</u>: Process explicit control list. PEC processes the control list for READ and WRITE statements. Since the unit and format specifiers may be keyword or positional format within the control list, the nature of the control list is determined. The possibility of character expressions (for internal files or format designator) must also be tested. This is accomplished by parser interface routines. When the control list has passed to keyword format, PKC is called to translate the remainder of the control list. Upon completion of control list processing, the control items specified are analyzed and the I/O routine to be used is determined. Semantic analysis of the control list determines if irregularities occurred, and diagnostics are issued as necessary. Exit is to LST to process the I/O list.

o. <u>BUFFER</u>: Entered from the front end controller main loop when a BUFFER IN or BUFFER OUT statement is encountered. The nature of the statement (IN or OUT) is determined and TSX is called to enter the proper I/O routine name into the symbol table. CUD is called to process the unit designator and SFP sets the relevant file property bits. Successive calls to PAR translate the mode designator, the FWA and the LWA. A call to IOJ issues the call turple. Any error results in termination of processing. Exit is to the front end controller via PSL.

p. <u>DECODE</u>: Entered from the front end controller main loop when a DECODE statement is encountered. Sets the I/O direction indicator to 'output' and exits to NDC.

q. <u>ENCODE</u>: Entered from the front end controller main loop when an ENCODE statement is encountered. Sets the I/O direction indicator to 'input' and falls through to NDC.

r. <u>NDC</u>: Process ENCODE/DECODE arguments. The record length argument is translated by a call to PAR. PFN is called to process the format designator. PAR is called again to process the string address. Upon return from translating the string address, the I/O direction indicator is reversed and exit is to LST to process the I/O list.

s. <u>CUD</u>: Compile unit designator. CUD is a source statement subroutine and is called by various translators to process the unit designator. If the unit designator is implied (UNIT=*), the proper unit (INPUT or OUTPUT) is selected. Otherwise, the unit designator is compiled by PAR. If the unit designator (TP. format) returned by PAR is type character, the legality of the value and the legality of an internal file is tested. A constant unit designator is processed by UDP. OUT is called to output the control turple for the unit designator. Fatal errors result in an error entry return.

t. <u>ICK</u>: I/O control keyword check. ICK is called by PKC to scan a supposed control item keyword against the table of legal keywords. If a match is found, the keyword table entry is returned, otherwise, a failure indication.

u. <u>IIC</u>: Initialize I/O control. IIC performs some general initialization of cells, flags and pointers required by various I/O statement translators.

v. <u>IOJ</u>: Compile I/O jump. IOJ is called whenever an I/O routine call turple is needed. IOJ outputs the proper I/O call turple (determining if the current call is initial or restart). The control and list item turples are flushed to T.PAR prior to issuing the call turple.

w. <u>OUT</u>: Output unit turple. OUT is required because the FCL requires the unit designator to be the first IOAPL item. Thus, for all I/O statements which use a unit designator, space was reserved on T.IOARG. The unit designator turple is output to T.SCR and then copied into the proper position on T.IOARG.

x. <u>PFN</u>: Process format or namelist designator. PFN is called to process the format/namelist designator from the positional or keyword control list processors. The S.IOCALL indicator is updated to reflect the various possible values. If free format (FMT=*) is specified, only the S.IOCALL indicator is updated. If a statement label is specified, ISL is called to process it. A call to SSY determines if the specifier is a namelist name. Otherwise, PAR is called to translate the designator. Upon return from PAR, the designator (TP. format) is analyzed semantically. Irregularities are diagnosed and the proper control list turple is output, via OCT.

y. <u>PKC</u>: Process keyword control items. PKC is called by the I/O statement translators when the control list is in keyword mode. IDK is called to determine the existence of a supposed keyword verb. If the verb is invalid, a diagnostic is issued and processing halts. A legal verb is tested against a mask of context legal verbs and again, if invalid processing halts on a diagnostic. Multiple occurrences of an individual verb is diagnosed at this time. If the verb is legal, the address of the relevant processing routine is extracted from KW.CTL and processing continues with the individual routines. PKC= routines are grouped by the legal values of the arguments. Processing of the individual arguments converges back in PKC and the proper control line turple is output.

The occurrence mask is updated to prevent multiple specification and keyword processing continues until a closing right parenthesis is noted.

z. <u>SFP</u>: Set file property bits. SFP is a routine called by various I/O statement translators to set file property bits for the map, attribute and cross reference listings. If the listings are suppressed, no action takes place. Otherwise, the I/O statement is analyzed and the proper symbol table entry for the relevant file is updated with the proper attribute bits.

aa. <u>UDP</u>: Unit designator processing. UDP is called when a constant unit designator is specified. VUD is called to validate the constant (if not arithmetic). Arithmetic constants are validated and a file name is added to the symbol table, as necessary.

bb. <u>VUD</u>: Validate unit designator. VUD is called by UDP to semantically test a Boolean constant as a unit designator. VUD returns an error exit or valid indication, as applicable.

cc. <u>A=BMOD</u>: Parser interface for BUFFER IN/OUT mode designator. Provides semantic check of the mode designator. Invalid operands are diagnosed. Exits to BUFFER via PAREXIT.

dd. <u>A=BLWA</u>: Parser interface for BUFFER IN/OUT LWA. Provides semantic tests for the BUFFER LWA. Determines, when possible, the relationship of the FWA to LWA and diagnoses invalid conditions. Exits to BUFFER via PAREXIT.

ee. <u>C=BFWA</u>: Parser interface for BUFFER IN/OUT FWA. provides semantic tests for the BUFFER FWA. Diagnoses invalid conditions. Exits to BUFFER via PAREXIT.

ff. <u>C=CNT</u>: Parser interface for ENCODE/DECODE string count. Diagnoses invalid operands. Exits to NDC via PAREXIT.

gg. <u>C=FMT</u>: Parser interface for format designator. When a format designator is translated by PAR (not statement label, namelist name or free format), C=FMT provides semantic tests for the validity of the format. If no errors are noted, the format control turple is output. Exit is to the original PAR caller, via PAREXIT.

hh. A=FOU: Parser interface for unit/format designator. In some cases, it is impossible to tell by examining one or two tokens whether the first element in a control list is representing the unit designator or the format designator. (Parenthesized expression). A=FOU allows PAR to parse the expression, then determine the nature of the operand. Exit is to A=FMT or A=UNT.

ii. C=ICC: Parser interface for I/O control items which must be character variables. Provides semantic tests of the parsed operand. Exits to PKC, via PAREXIT.

jj. C=ICCX: Parser interface for I/O control items which must be character expressions. Provides semantic tests of the parsed operand. Exits to PKC, via PAREXIT.

kk. C=ICI: Parser interface for I/O control items which must be integer variables. Provides semantic tests of the parsed operand. Exits to PKC, via PAREXIT.

ll. C=ICIX: Parser interface for I/O control items which must be positive integer expressions. Provides semantic tests of the parsed operand. Exits to PKC, via PAREXIT.

mm. C=ICL: Parser interface for I/O control items which must be logical variables. Provides semantic tests of the parsed operand. Exits to PKC, via PAREXIT.

nn. C=IOL: Parser interface for I/O list items. C=IOL determines the I/O direction (in, out) and if input, calls VAI to determine the legality of the item. If the I/O list item is an array item reference, C=IOL determines if it is the target of a partial implied do collapse, and modifies the item as necessary. If the item is an entire array, the size of the array is determined and the length operand is set. Assumed size arrays are flagged as errors. Finally, an IOAPL turple is output to T.IOARG. Exit is to IOL..RTN.

oo. A=STR: Parser interface for ENCODE/DECODE string address operand. Provides semantic tests of the parsed operand. Exits to NDC, via PAREXIT.

pp. C=UNT: Parser interface for unit designator. This routine is a stub, used for conformity with the parser's ARGMODE schema. Semantic tests are performed by CUD.

qq. CML: Check for match in T.ILI. CML is called when processing an input list when a subscript variable is encountered. During processing of the list, all variables and array elements are added (base/bias form) to T.ILI. Whenever a variable occurs in a subscript expression (or as an implied do induction variable) it is test by CML to determine if it is on T.ILI. If so, a restart call must be issued in order that the variable will be properly defined.

rr. KWE: Keyword parameter error setup. KWE is called by the I/O control item parser interface routines to set up the FILL. calls properly for diagnostic output.

ss. OCT: Output control turple. OCT is called from the various I/O control item translators to format and output an I/O control turple to T.IOARG.

tt. OST: Output skip turple. OST is called by the I/O statement processors to determine the necessity of a default skip turple. If necessary, a generated label is made, the skip turple is output, via OCT, and a flag is set to alert the necessity of outputting the generated label turple.

uu. VAI: Validate addressable item. VAI is called when an operand must be addressable (can be stored into) (e.g. variable, array element). A call to DOA determines the addressability. If a valid store item and the current reference defines the item, the proper symbol table bits are set, and DDR is called to determine if the definition redefines an active do control index.

vv. AII: Add input item to T.ILI. AII is called to add an input list item to T.ILI for future test by CML. AII converts the item to base/bias form and adds it to the table.

ww. FII: Format input item. FII is called by AII to properly format the item for inclusion on T.ILI.

xx. LST: Process I/O list. List is entered from the I/O statement translators which can have I/O lists. It also processes DATA statement variable lists. The transition from control list to I/O list is performed (syntax checks) and OST is called to output a default skip turple, as necessary. The control items are finished and initialization is performed prior to calling CVL

to actually process the list. Upon return from CVL, the I/O statement compilation is completed. A call to IOJ flushes T.IOARG to T.PAR, the terminal I/O call turple is issued and the generated label for the default skip is issued, if needed. A call to OIL provides the necessary sequence break and exit is to the front end controller, via PSL.

yy. CVL: Compile input/output/data variable list. CVL sets status calls pertaining to I/O list processing and calls IOD to mark any implied dos. After the preliminaries, CVL loops on the list items, calling PAR to process simple items, and transferring to special I/O routines to process do begin, conclusion and collapse items. Upon completion of I/O list processing, CVL tests for unterminated implied dos and cleans up any loose ends.

zz. DCB: Do collapse begin. DCB is entered from the CVL loop when a do collapse begin token (O.DCBI) is encountered. If the final collapse level was variable, a multiply turple is issued for the block move size operand. All collapse affected subscript variables are set to the initial value by issuing store turples. All turples issued are to T.PAR. Return is to the CVL loop, with the token cursor pointing at the object array token.

aaa. DCC: Do collapse conclusion. DCC is entered from the CVL loop when a do collapse conclusion token (O.DCCI) is encountered. DCC merely resets the token buffer pointer past the final O.DCCI token and returns to the CVL loop.

bbb. DOB: Do begin. DOB is entered from the CVL loop when an implied do token (O.DOBI) is encountered. DOB first calls IDC to perform collapse analysis. If complete collapse was performed, exit to the CVL loop is immediate. Otherwise, the token buffer pointer is set to the implied do control variable and CDI is called to translate and compile the implied do. Upon return from CDI, the do conclusion token is made and pointers are adjusted to provide later conclusion processing. Return is to the CVL loop.

ccc. DOC: Do conclusion. DOC is entered from the CVL loop when an implied do conclusion (O.DOCI) token is encountered. IOJ is called to provide the necessary restart call. PDT is called to compile the necessary do conclusion turples. Return is to the CVL loop.

ddd. IDC: Implied do collapse. IDC is called from DOB to attempt collapse of some or all levels of nested implied dos. If array bounds checking is in effect, no collapse is performed. If the current do structure has already been checked, exit is immediate, with no processing. The token buffer is then scanned, counting O.DOBI tokens, until a non do begin token is encountered. This establishes the nesting level for the current implied do structure. If the maximum allowable dimensionality is exceeded, collapse is abandoned. The next tokens must be O.VAR and it must represent an array or collapse is abandoned. Information about the array (dimension pointer, number of dimensions, etc) is saved and PCI is called to translate the subscripts of the array. Upon return from PCI, the subscripts have been parsed and are saved in an index table (base/bias form or an indicator that the subscript was not a simple variable. The implied do induction and control variables must follow immediately after the subscripted array, or collapse must be abandoned. Next comes a loop in which the control and induction variables are examined, testing for collapse possibility. If the control variable matches the corresponding subscript (index table) collapse is possible. The induction variables are tested via a call to PCI. Nonunity increment disqualifies the current level for collapse. If the initial and limit variables (constants) match the lower and upper bound for the corresponding dimension, the current level collapses. The tokens representing the control and induction variables for a collapsed level are rewritten to provide collapse information. This information includes the O.DCCI token, the initial value of the subscript, the collapse size (which is accumulated from level to level), variable size multiplier (or null) and a starting bias (when the last collapse level initial value is greater than the lower bound for that dimension, or is variable). Processing of the control and induction variables continues until the entire structure is collapse or until an index marked 'not collapsible' is met. If collapse

was incomplete, the O.DCBI (and probably O.DOBI) tokens must be processed. Exit is to DOB, with the token buffer positioned properly and the do collapse information set up. If collapse was complete, the size of the I/O move is calculated (issuing arithmetic turple if necessary) and an IOAPL turple is issued. If the last collapse level initial value is variable, an array load turple is generated which is then referred to in the IOAPL turple. The token buffer pointer is reset past the entire implied do structure and exit is to the CVL loop, via DOB.

eee. IOD: I/O do. IOD is called from CVL to mark occurrences of implied dos. The token buffer is scanned, keying on parentheses and equal signs. The relevant tokens are modified from O.LP to O.DOBI and the pointer fields are modified to point to the induction/control variables. IOD is called once per I/O list.

fff. A=DOCI: Parser interface for implied do collapse induction variables. Counts the induction variables as they are translated by PAR. When the list of induction variables is complete, exit is to IDC via PAREXIT and PCI. If too many induction variables are present, collapse is terminated and the do processor will handle the diagnostic.

ggg. A=DOCS: Parser interface for implied do subscript processing. The parsed subscripts are analyzed and if a simple variable, its base/bias form is stored as the index for collapse processing. Otherwise, a collapse invalid mark is stored. If the array dimensionality is exceeded, collapse is abandoned. Upon completion of subscript translation, exit is to IDC via PAREXIT and PCI.

hhh. PAX: Prepare array cross section. PAX issues turples to define do induction variables for implied NOS. Also determines necessity of restart calls due to data interference. This information is gleaned from T.ILI. Adds the information currently processing to T.ILI for future PAX calls.

iii. PCI: Parse collapse items. An interface to the front of PAR. Sets up special processing tokens to allow PAR to special process the items in question. Upon return from PAR, the modified tokens are restored. Called from IDC.

3.12 PAR: Expression Translation

Abstract: PAR is a general purpose expression translator. It is used to parse virtually all expressions which can occur in the FTN5 language. PAR consists of analysis and synthesis routines for the various possible operators and many subroutines necessary for production of the IL.

Interfaces: PAR is a front end deck and resides on overlays (0,0), (1,0) and (2,1). PAR interfaces with every front end deck. The major communication is provided by the PARMODE cell and the ARGMODE, ARGMIS and ARGCOMA cells, which control the parser behavior in many situations.

Data Structures:

The major concern of PAR is the output of IL turples and thus, T.PAR is the major data structure. In addition, PAR defines the following structures:

a. **Cells:** PAR defines several cells for use as flags, pointers and scratch storage.

b. **CONO:** The operator/operand connotation table. Used to determine the legality of the current token in combination with the next upcoming token. Format is:

```
+--------------------------------+-------------+
!              CONO              !    ADDR     !
+--------------------------------+-------------+
               42                      18
```

CONO: Legality bits, corresponding to O. values on = legal successor.

ADDR: Address of PAR. analysis routine

The CONO table is ordered as the O. token values. Successor element is tested by a shift of the token value.

c. **POPNX:** A vector of synthesis (pop) addresses for operators which are popped from OSTACK.

d. **PR.SET:** A set of stack priority values for operators which occur in normal expressions. Used by the PRIOR table.

e. PRIOR: Operator priority table. Ordered by O. token values, PRIOR has entries for all token generated operators. Format is:

```
+--------------+----------+--------+-----+-----+
!     SKEL     !   ATTR   !////////! STP ! TBP !
+--------------+----------+--------+-----+-----+
       18          13         11      9     9
```

SKEL: Skeleton index (or special proc. routine)
ATTR: Front end attributes
STP : Stack priority
TBP : Token buffer priority

Note that this structure is a special case of the TH. structure (described in A-2-2).

f. SETOP: An unordered table of operators which are not strictly token driven. These are accessed by name as required for turple output. The format is as the PRIOR table above.

g. SETARM: An unordered table of ARGMODE values, used by PAR to process lists of items separated by commas and terminated by right parenthesis or end of statement. Format is:

```
+---------+---------+-------------+-------------+
!   REF   !  ATTR   !     COM     !     PAD     !
+---------+---------+-------------+-------------+
    12        12          18            18
```

REF : Cross reference character for list items
ATTR: Item attributes (conditions)
COM : Address of comma interface routine
PAD : Address of parenthesis interface routine

h. Macros: All of the above tabular structures are defined using appropriate macros.

i. Other: Small tables and vectors are defined for the processing of specific operators. These structures will be described with the pertinent routines.

Routine_Descriptions:

PAR is an operator precedence parser, with built in deviations, mainly the ARGMODE structure. The deck structure is: parser front end interface routines (to set up various flavors of parse), the analysis routines (operator precedence parse), the synthesis routines (including parser post analysis interface and special purpose subroutines, driven by the ARGMODE values) and general subroutines concerned with IL output. Routine descriptions follow.

a. CNF: Compile normal formula. CNF is entered from the front end main loop when an assignment statement is encountered. Translation and compilation is performed via a call to PAR. Exit is to the front end controller via PSL.

b. PIX: Parse integer constant expression. PIX is a PAR front end interface routine called when an integer constant expression is required. The ARGMODE and PARMODE values are set to filter non integer, non constant values as error conditions. The supposed integer constant expression is translated by a call to PAR and tested by LCT. Any expression which did not reduce to an integer constant is diagnosed, and a 'made' constant '1' is returned. C=PIX provides the post analysis parser interface.

c. PKX: Parse constant expression. Operates like PIX except any constant value is an acceptable result (LCH is the constant test routine) and the PARMODE value is supplied by the calling routine.

d. PAR: Parser. PAR is the subroutine name of the parser. Exit can be effected from many points within the parser, even from low level subroutines and interface routines. In general, exit is normal. The parser operates with two stacks OSTACK (for operators, including nested argmode information) and ESTACK (for operands). Two lock registers are of note: B5 points to OSTACK, B6 to ESTACK. These locks must be preserved from the initialization in PAR to the final exit. (B4 is still locked to the token buffer, but that is standard in front end routines). PAR performs the aforementioned initializations and tests for legality of the initial token of the expression. That done, fall through is to the PAR. loop and analysis routines.

e. PAR.: The PAR.xxx routines make up the analysis phase of the parser. Descriptions of these routines follow:

PAR.NX: The analysis loop. The current token is fetched from the token buffer. Its value is used as an index to CONO. The PAR. jump address is extracted and the next token in the token buffer is fetched. Its value (the second token) is used as a shift count for the CONO legality bit pattern. If legal, branch to the relevant routine, if not, exit to PAREX for diagnostic and recovery.

PAR.TRU:
PAR.FAL: Entered from PAR.NX when O.TRUE or O.FALSE is encountered. Determines if logical constant is legal in current context. If so, a TP. form logical constant is added to ESTACK and exit is to PAR.NX. Otherwise, a diagnostic is output, via DBE, and an integer 1 is stacked.

PAR.OCT: Entered from PAR.NX when O.OCT or O.HEX is encountered. The relevant token is converted to binary by OCT and the result is processed at TNK.DBL.

PAR.DEC: Entered from PAR.NX when O.CONS or O.PERIOD is encountered. The constant represented by the token is converted to binary by DEC and the result is processed at TNK.DBL.

TNK.DBL: TNK.DBL is the processing routine for translated constants. Upon return from OCT or DEC, the tokens representing a constant have been translated to binary, and the mode has been determined. The binary is entered into T.CON (as necessary) via NBC and a 'follow function' test is made to determine the legality of the token following the last token of the constant and a constant. If legal, return to PAR.NX. Otherwise, exit to PAREX for diagnostic and recovery.

TNK.PARM: Entered from PAR.VAR when a symbolic constant is encountered. The syntax and semantics of the PARAMETER usage is tested, and irregularities are diagnosed. If all is in order, the TP. format of the constant represented by the PARAMETER is added to ESTACK.

PAR.HOL: Entered from PAR.NX when O.HOL is encountered. The O.HOL token is converted to TP. format. Diagnostics (Hollerith is non-ANSI) are issued as required. The constant is added to ESTACK.

PAR.CHR: Entered from PAR.NX when O.CHAR is encountered. The context legality of the character constant is tested, and if legal, the character constant is processed by ECC. Otherwise, a diagnostic is issued and an integer 1 is added to ESTACK.

PAR.VAR: Entered from PAR.NX when O.VAR is encountered. This is the semantic analysis routine for all symbolic names explicitly referenced in expressions by the user. First, SSY is called to determine if the symbol is in T.SYM. If the symbol is in the table, the properties assigned are tested. If PARAMETER, exit to TNK.PARM for processing. The remaining possibilities are array, variable (scalar) or some kind of function. For each possibility, indicators will be set/extracted and the process will converge at TREX. If the symbol is an array, semantic tests are made (is the array followed by subscripts?, is the array leveled?, etc) and diagnostics are issued as necessary. Exit to TREX. If the symbol is a variable, semantic tests (fewer) are again made, and exit is to TREX. Now the symbol must be some kind of function or there is an error. The various function possibilities are tried (intrinsic, statement, external, etc) and when a 'hit' is found, the relevant properties are extracted for TREX (and beyond) and any semantic checks required are made. If nothing good has happened, the nature of the symbol is determined and the relevant diagnostic is output. In some cases, a symbol will be unclassified (appears only in a type declaration). When these symbols are encountered in an expression, followed by an argument list, the usage defines a function. The type of function (intrinsic or user external) is determined, and the symbol table entry is updated to reflect the classifying usage. When the symbol is not in T.SYM, it may be only a variable or a function, and if a function, it may only be intrinsic or external user. The determination is made by the presence or absence of an argument list. Exit to TREX.

TREX: TREX is entered from various parts of PAR.VAR when the nature of a symbol is determined. Further semantic tests are made here, pertaining to existence of the symbol usage in an input list or in a dimension bounds expression. If all is well, a TP. form operand is made and added to ESTACK. Exit is to the processor designated by the PAR.VAR process (PAR.NX for variables, unsubscripted arrays and functions without argument lists).

PAR.STFD: Entered from PAR.NX when O.STFA is encountered. An O.STFA token is manufactured when a statement function definition is translated (see STMTF, 3.14). The token is modified by the statement function reference processor. When encountered by the parser, CLM is called to provide the necessary mode coercion and to diagnose errors.

PAR.SUB: Entered from PAR.VAR (TREX) when the symbol is determined to be a subscripted array. SSO is called to set up the subscript operator. ARGMODE indication is set for subscript processing and exit is to PAR.SPS.

PAR.SBS: Entered from PAR.VAR (TREX) when the symbol is a substringed variable. Semantic tests are made (the variable must be type character) and exit is to PAR.SPS with ARGMODE indicator set for substring processing.

PAR.FUN: Entered from PAR.VAR (TREX) when the symbol is a function reference. The type of function involved has been determined. Based on this type, a second branch is made to correctly set the ARGMODE indicator, ARGMIS and ARGCOMA. Based upon individual function type, further semantic tests are performed, and exit is to PAR.SPS.

PAR.SPS: Set parenthesis stack. Entered whenever a left parenthesis is encountered. The routine is supplied new values for ARGMODE, ARGMIS and ARGCOMA. The current value of those cells is saved on OSTACK as well as the left parenthesis operator. The supplied values are then stored in the proper cells, thus altering the processing for comma and right parenthesis separation. Special syntax checking is done for null parameter lists and substring default. Exit is to PAR.NX.

PAR.CM: Entered from PAR.NX when O.COMMA is encountered. Also entered from PAR.COL when a legal colon token is determined. A comma (or colon) must pop all operators back to the parenthesis operator. This is done in a loop, calling POP. When the expression has been reduced, a call to POP, to pop the comma is made and processing will be as determined by ARGMODE.

PAR.COL: Entered from PAR.NX when O.COLON is encountered. If the colon token is legal, exit is to PAR.CM (colon parses as a comma). Otherwise, an error exit is taken to provide diagnostic and recovery.

PAR.LP: Entered from PAR.NX when O.LP is encountered. CFC is called to determine if a complex constant is present. If so, exit to TNK.DBL for processing. Otherwise, set up ARGMODE cells for parenthesized expression processing and exit to PAR.SPS.

PAR.DLP: Entered from PAR.NX when O.SLP is encountered. This dummy O.LP token is manufactured by the routine which called PAR and indicates that the ARGMODE cells contain valid information and should be retained. Exit is to PAR.SPS with the ARGMODE indicators set to those current values.

PAR.RP: Entered from PAR.NX when O.RP is encountered. Odd syntax constructs are tested (e.g. substringed array reference, logical if). If no irregular syntax is present, or if the syntax is legal, exit is to PAR.STD. Else, a diagnostic exit is taken.

PAR.EQL: Entered from PAR.NX when O.= is encountered. The legality of the assignment operator is tested, and if invalid, a diagnostic is issued. If the operator is assignment (not a do '='), processing is deferred pending the parse of the left side. Otherwise, exit to PAR.STD.

PAR.DIV: Entered from PAR.NX when O.DIV is encountered. This routine tests for possible integer divide, and if that is the case, sets a flag to force multiply operations first. Exits to PAR.STD.

PAR.PL: Entered from PAR.NX when O.PL is encountered. Determines if the operator is unary, and if so, adds a constant zero to ESTACK. Exit to PAR.STD.

PAR.MIN: Entered from PAR.NX when O.MIN is encountered. Determines if the operator is unary, and if so, converts the token to O.UMIN. Exit to PAR.STD.

PAR.MULT: Entered from PAR.NX when O.MULT is encountered. The syntax for passing alternate return labels as parameters is a statement label preceded by *. This unary * must be allowed and semantically tested here. If the * is not unary, exit to PAR.STD. Otherwise, make semantic tests and if an alternate return, translate the statement label via ISL. Else issue a diagnostic.

PAR.STD: Entered from PAR.NX or from special processing routines to process all operators. The operator just translated is compared with the operator on top of OSTACK (the priority of the operators are what is compared). The operator is either stacked (by PAR.ADOP) or will pop the top of OSTACK (via a call to POP). If the current operator caused a pop, upon return from POP, the operator is again tested against the new OSTACK top (a loop back to PAR.STD) unless the last pop was an O.RP popping an O.LP. Exit then is to PAR.NX.

PAR.ADOP: Entered when an operator is to be entered on OSTACK. The token buffer SETOP form (special case TP.) is modified to the OSTACK form (see section A-2-2) and the operator is added to OSTACK. Exit to PAR.NX.

PAR.ERR: PAREX Entered from various PAR. processors when apparent error conditions (of a CONO nature) occur. Tests are made to determine if special syntax is allowed in the current context. If so, exit to the relevant processor is made. If not, the diagnostic is set up and issued, the invalid token is skipped (perhaps more than one token) an error turple is issued and exit is to PAR.NX to continue processing.

PAR.EOS: Entered from PAR.NX when O.EOS is encountered. There are two possibilities when this routine is entered, the end-of-statement is real (at least as far as PAR is concerned) or the EOS is the assignment statement mechanism for processing replacement. For replacement statements, since multiple replacement is allowed, the '=' tokens are replaced with chain pointers, linking all the replacements together. A cell pointing to the innermost '=' is kept, and that token is replaced by O.EOS. When processing each level, the pointer is fetched into the cell and the O.EOS token is substituted. When the cell is null, all replacement has been processed (or no replacement was present). If the replacement case is being processed, the operands will be on ESTACK, so OSTACK is set for popping the replacement operator, and POP is called. This will continue until all replacement operations have been processed (interfacing with PAR.EQL). When the cell (ZLE) is null, a true end-of-statement occurred. If necessary, the final operator is popped and exit is made, via PAREXIT. Note that PAREXIT is a general exit point from most parser interface routines (for comma and right parenthesis delimited lists) and will often cause an exit from POP directly through PAR.

PAR.STOP: Entered when errors are of a magnitude that preclude recovery. An error turple is issued and exit is to the front end controller, via PSL (by passing the PAR exit).

f. POP: Popping (synthesis). POP is called from several of the PAR. routines to emit turples (operator, operand, operand). The POPNX vector entry corresponding to the operator being popped is fetched as are the two top ESTACK entries. If the operator is commutative, the operands may be reordered (for code optimization). The POP. routine defined in the POPNX entry is brached to for semantic analysis, front end optimization and turple emission. In many cases, exit will not be through the POP subroutine return, but through PAREXIT or some other PAR. routine. Exits are described with the relevant POP. routines.

g. POP.: The popper processing routines. Some of these routines are not at all straightforward. The parser interface routines for comma and right parenthesis popping call subroutines, are located in decks other than PAR (where they are described) and take non-standard exits. The relevant subroutines and interface routines which do reside in PAR will be described here.

POP.ERR: Entered when an error was detected in the POP phase of parsing. Sets up an error turple and processes via POP.STD.

POP.STD: Entered when no special semantic tests are required (from POP) and from the other POP. processors after their transformations are complete. Calls SDM to perform mode coercion (not all operators). Calls OMC to output the mode coercion and ADT to issue the turple.

POP.CM:
POP.COL: Entered when popping a comma (or colon) operator. The current ARGMODE value is used to extract the address of the processing routine. The C=xxx address is branched to. Note that many of the routines are not in PAR. In general, the interface routines reside on the decks which contain the relevant translators.

C=ERR: Entered when a comma isn't allowed in the current parenthesis structure. Exit to POP.ERR.

POP.RP: Entered either on a special syntax or when an unbalanced parenthesis condition occurred. (The unbalanced parenthesis is of the compiler generated ilk.) Exit to PAR.EOS if special syntax, issue a diagnostic and exit to PAR.STOP otherwise.

POP.PN: Entered from POP when a right parenthesis is popping a left parenthesis. The OSTACK entries for the old ARGMODE cells are unstacked. The A=xxx address of the processing routine is extracted for the branch. The current ARGMODE values are saved in scratch storage for reference and the former ARGMODE values (from OSTACK) are restored. Exit is to the relevant A=xxx routine. The notes of POP.CM also apply here. Of interest is the fact that many of the interface routines (but not all) are common code for C=xxx and A=xxx processing, differing only in the advance of the token buffer pointer.

A=IF: Interface routine for IF statement. Determines if the if expression is a reducible relational operand, via COR. Sets result cells for the if processor and exits via PAREXIT.

A=LIST:
A=DVL: Interface routine for I/O and DATA statement variable lists. Support for special syntax which has an O.SLP unstacked by O.EOS. Merely exits through normal POP exit.

POP=COM: This is a front optimization which delays output of a commutative and associative operator containing a constant. The idea is to keep pushing these down so that all constants in a string of such operators may be combined into a single constant at compile time. When the conditions are met, the operands are adjusted so that the constant is second, the operator is modified to indicate the deferred status and is replaced on the OSTACK and the operands are replaced on ESTACK. When both operands are constant, normal popping is performed (and the operation will be reduced). Otherwise, exit to PAR.ADOP to perform the mechanics of the deferral.

POP.DIV: Entered from POP when a divide operator is to be popped. Test is made for constant divide by zero. If so, a diagnostic is issued. Attempt is made to convert the divide operation to a multiply (or a series of multiplies, as operators are popped), when a divide pops a divide. The operator is changed to a special divide and replaced on OSTACK (via PAR.ADOP). This process interfaces with POP=COM. As a last resort, a divide turple is output.

POP.PL: Entered from POP when a plus operator is being popped. The following transforms are performed. If the first operand is the result of a unary minus operation, the operands are reversed and a subtract turple is output. Otherwise, exit to POP=COM to delay for possible constant reduction.

POP.MUL: Just a pseudo for POP=COM. Always delay on multiply.

POP.UM: Entered from POP when a unary minus is being popped. The last operator output is tested was minus, COR is called to determine if that operation is input to the current unary minus. If so, the order of the subtract is reversed and exit is immediate through POP. If a unary minus is input to a unary minus, the negation of operators is performed, the turple for the input unary minus is discarded and the operand is restored to ESTACK. Otherwise, the unary minus turple is output.

POP.NOT: Entered from POP when a NOT operator is being popped. Uses the code of POP.UM, except set up to check NOT as input to NOT (for possible cancellation).

POP.LOG: Entered from POP when popping a logical operator. If the operator is commutative, exit to POP=COM, else to POP.STD.

POP.LE:
POP.GT: Entered from POP when the operators LE and GT are being popped. Reverse the operands and convert the operator to GE and LT. Falls through to POP.REL.

POP.REL: Entered from POP when popping relational operators. Semantic test are performed on the operands. If parsing in constant expression mode, character relationals are processed via PCR. The relational turple is issued via ADT. If the relational is the only operator in the expression, an inverted skeleton is saved, to be used when a single relational occurs. (If ADT is called again before exit, the hold cell will be cleared. Exit is to POP.

POP.CAT: Entered from POP when a concatenation operator is being popped. Both operands must be mode character or a diagnostic is issued. If both operands are constant (test by LCH), compile time concatenation is performed by PCC and the new character constant is emitted by ECC. Otherwise, the operands are tested for fixed length via GOL and TH. bits are set, marking the turple as concatenation and fixed/variable length. Exit is to PAR.STD.

POP.EQL: Entered from POP when an assignment operator is to be popped. The mode of the right hand side is coerced to that of the left side via CMR. The legality of the left side is tested, and diagnostic is issued as necessary. If the left side is of type character, test is made to determine if any portion of the left side occurs on the right side. If so, a diagnostic is issued. DDR is called to diagnose do control index redefinition. If the LHS and RHS are the same, the turple is not issued and exit is to POP. Otherwise, exit to POP.STD.

C=SBS: Interface routine for substring processing, the first operand. If the colon is not immediately followed by O.RP, exit to POP. Otherwise, the variable to be substringed is tested for legality and the character length is determined. If length is variable, a VD. operand is formed. Exit is to POP.

A=SBS: Interface routine for substring second operand. Both operands have been translated. They are tested for constant, via LCH. If the substring bounds are constant, they are tested for valid range. In any event, a colon operator is output, calling SDM and ADT and then the substring operator turple is output, again using SDM and ADT. Exit is to POP.

CSM: Check substring mode. CSM is called by C=SBS and A=SBS to determine that the mode of a substring expression is integer (or Boolean). If not, a diagnostic is issued.

A=DO: Interface routine called when EOS or O.RP is found after the list of induction variables. Determines that sufficient values have been translated. If so, exit to PAREXIT. Else, issue a diagnostic, scratch the T.BLST entry and exit to the front end controller via PSL.

C=DO: Interface routine called to process the various do control and induction variables. Based upon the ARGCOMA count, the proper processing routine is selected. If too many variables appear in the list, a diagnostic is output and PAR is exited.

DOS: Process do start (initial value). The do '=' token is replaced by a comma token, to properly process the list of induction variables. The start value is converted, via CDP and stored on T.BLST. Exit to POP.

DOL: Process do limit. The limit value is converted via CDP and added to T.BLST. A default '1' is set for the do increment and fall through to DOI.

DOI: Process do increment. The increment value is converted via CDP and added to T.BLST. Exit is to POP. Since the increment is optional, the value is initially set to constant '1' and reset as required.

DOC: Process the control variable. Semantic tests of the legality of the control index are made and diagnostics are output as required. The T.SYM entry is marked as defined and DDR is called to diagnose active do redefinition. The control index mode is determined (and the legality of the mode tested) and the skeleton type (long or short) is selected. The control index is added to T.BLST and the symbol table entry of the do top label is modified with the symbol table index of the control variable. Exit is to C=DO.

CDP: Convert do parameter. The mode of the relevant induction value is tested for legality and diagnostic is issued as necessary. The mode of the induction value is coerced to that of the do control index, via CMR.

DTC: Determine trip count. Called from CDI (see LABEL, 3.15) when all induction values have been translated. Turples to provide the calculation TC=(M2-M1+M3)/M3 are output, via ACT. Semantic tests on the resultant value are made if processing a DATA statement implied do. The resultant value is coerced to mode integer via CMR and the resultant value is returned.

POP.EXP: Entered from POP when an exponentiation operator is being popped. This routine attempts to apply compile time transformation aimed at eliminating run time subroutine calls. The mode of the power and the base are determined. The table OM=EXP is used to determine the legality of the combination. Illegal combinations are diagnosed and exit is to POP. LCT is called to determine if the power is constant. If not, reduction is impossible. Semantic tests are made on the power and if possible, inline code is generated (via turple output). Real powers which are exact integers (e.g. 2.0) are treated as integer. The result will be a turple output, either a inline series of multiplies (or divides for negative powers) or an external library call.

TO.xI: Routines to select skeletons for inline multiplies (base mode x, power mode Integer). These small routines all exit to EXM.

EXD: Evaluate integer exponential. Called from EXP.I (see CONRED, 3.13) to subsume constant integer exponentiation. The actual reduction is performed by EXV.

EXM: Expand exponentiation into multiplies. EXM receives the base mode, the expansion limit and value of the power and actually selects the skeleton for turple output. The turple is output via ADT and exit is to POP.

EXV: Evaluate constant exponential. Called by EXD and EXM to reduce constant exponential expressions. If the expression can be reduced at compile time (implies integer power, but some bases will reduce) the constant is returned, otherwise, turples must be issued.

C=CALL: Interface routine for subroutine argument lists. Calls TPC to test for passed length concatenation and SSA to stack the argument. Exits to POP.

A=CALL: Interface routine for final subroutine argument. As in C=CALL, TPC and SSA are called. IAC is called to increment the argument list and VEL is called to validate the list. If necessary, CRL is called to process alternate return labels. EAL is called to emit the AP list turples and exit is to POP.

C=FUN: Interface routine for user function argument lists. Calls TPC and if alternate return labels were present, a diagnostic is issued. Call SSA to stack the argument and exits to POP.

A=FUN: Interface routine for user function final argument. Calls TCP, IAC and SSA to provide functions listed above. Calls VEL to validate the argument list and exits to GFR.

C=INF: Interface routine for intrinsic function argument lists. Calls TPC. Calls VAM to validate the argument mode and exits to POP.

A=INF: Interface routine for intrinsic function final argument. Calls in succession TPC, IAC and VAM. VIL is called to validate the argument list count. Falls through to ABEF If intrinsic is external, else exit to ABIF.

ABEF: Process external intrinsic function. ABEF determines if traceback is required and selects the proper call skeleton. TXI is called to process the function operand. The arguments are moved from ESTACK to T.ARG and exit is to GFR.

ABIF: Process incline intrinsic function. Calls ESF to evaluate the function. If the function has only one argument, exit is to POP.STD. Otherwise, the arguments are processed as a loop, with ADT called to output the inline turples. When the arguments are processed, if conversion is necessary, that turple is output by ADT. Exit is to POP.

C=STFA: Interface for statement function argument list. Merely a compatibility device. Exit to POP.

A.STFA: Interface routine for statement function final argument. The statement function argument list has been translated and any relevant turples have been issued. The arguments must now be processed using the statement function skeleton. The token buffer pointer is locked on the proper T.STF offset and exit is to PAR.SPS to begin processing of the statement function proper.

A.STFE: Interface routine for completion of statement function expansion. The token buffer pointer is reset, either to a new T.STF location (for nested statement functions) or to the actual token buffer, in which case the ALC lock is removed. Pointers and flags are reset to denote the current state of translation and CLM is called to provide mode (and character length) coercion. Exit is to PAR.NX.

GFR: Generate function reference. Entered from interface routine for argument list which requires an external call. EAL is called to emit the APlist turples. The proper call turple is provided by the relevant interface routine and processing is at POP.STD.

EAL: Emit actual parameter list (APlist). The arguments are located on T.ARG. The operator and count are supplied. EAL loops through the list, inserting mode information into the operator and issues the APlist turples, via EMT. The processed arguments are removed from T.ARG.

ESF: Evaluate special function. ESF is called by ABIF to process inline intrinsic functions. ESF consists of a series of routines to process those functions which require parser action. Those functions which completely reduce have their values on ESTACK. Others require ABIF to issue turples.

ES.CMPL: Process CMPL. Both arguments are coerced to real, via CMR. Exit to ABIF.

ES.LEN: Process LEN (character length). Calls GOL to get the length. If the length is constant, exit to ES.MASK to enter the constant. Otherwise, an external call must be issued. Exit to ABEF.

ES.LGE: ES.LGT ES.LLE ES.LLT	Process the character relational functions. The intrinsic function reference is converted to the form 'OP1.OPR.OP2' and the relevant turple is issued by ADT. Exit is to ABIF.
ES.MASK:	Process MASK. If the operand is a short integer constant, the mask is created and processed by NCS. (This is where the ES.LEN processor enters.) If not, a turple must be issued by ABIF. Either case exits to ABIF.
ES.SHIFT:	Process SHIFT. This routine attempts reduction of null shifts. If reducible (i.e. 0 mod60) ABIF is so informed, else a turple must be issued.
ES.LOCF:	Process LOCF. The addressability of the operand is determined via DOA. If not addressable, a diagnostic is issued. If the argument is mode character, an external call must be issued and exit is to ABEF. Otherwise, the LOCF is reduced to an ADDR operator and exit is to ABIF.
ES.RANF:	Process RANF. The random kernel is added to the symbol table, as necessary and a random number turple is issued via ADT. Exit to ABIF.
IAC:	Increment argument count. Called when processing the final argument of an APlist. The argument count is always one behind the number of arguments (to allow zero arguments). The ARGCOMA cell is updated with the true count.
MAD:	Mark argument defined. Called by SSA when an actual argument is added to T.ARG. If the argument is addressable (determined by DOA), its T.SYM entry is marked as defined.
SSA:	Stack subprogram argument. Called by the interface routines for subprograms which require an external call. The nature of the argument is determined. If not a label, the argument is added to T.ARG and MAD is called to mark it as defined. If the argument is a label, it is added to T.SLARG.

TXI: Tag external intrinsic. Called from PAR.VAR and ABEF when an external intrinsic must be called (the ABEF call is for intrinsics which may be either inline or external and are after analysis determined to be external). The information in the intrinsic function table and control card options are used to determine the external name the function is to take. A symbol table entry is made, via TSX, and the WB. and WC. information is filled in by TXI.

VAM: Validate argument mode. Called by A=INF and C=INF to determine the legality of an intrinsic function argument. If the function is generic, the first argument fixes the actual function and the mode of the other arguments. Otherwise, if the argument is not of the proper type, a diagnostic is issued.

VEL: Validate argument list for external. VEL is called by A=CALL and A=FUN to validate the argument count. If greater than the maximum allowable, a diagnostic is issued. If this is the first reference to the subprogram, it is marked defined and the argument count is entered in T.SYM. If previously referenced (not as an actual parameter) the current count is compared with the previous count and if the counts don't match, an informative diagnostic is issued.

VIL: Validate argument list for intrinsic. Called by A=INF to validate the argument count for an intrinsic function. If an argument count is specified and the current argument count doesn't match, a diagnostic is issued. If the intrinsic function is generic, the proper function is selected, based upon the mode of the argument(s). The argument list is scanned to determine that an invalid mixing of arguments didn't occur (if it did, a diagnostic is issued).

C=ARRAY: Interface routine for processing array subscripts. Calls are set up for possible diagnostics. SSR is called to standardize the subscript value. Exit is to PAR.NX.

A=ARRAY: Interface routine for processing the final array subscript. SSR is called to standardize the final subscript. GDI is called to get the dimension information relevant to the array. Determination is made (based upon dimension bounds check requests) whether to evaluate the subscripts inline or by external call. If the subscripts are to be evaluated inline, a loop on the subscripts is made, calling MSP, ASE and SLB to provide the necessary subscript calculations. The subscript value will be kept as constant as possible, with turples issued only as required. If the subscript is to be evaluated out-of-line, the relevant T.DIM entry is marked to materialize. Subscript APlist turples are output for each subscript expression. An APlist turple is output for the array name and ADU is called to emit the call turple for the bounds check routine (which will also evaluate the subscripts). If the subscript value was constant, the result will appear in the bias field of the array operand and exit is to POP. If the subscript was variable (or the array had adjustable dimension(s)) an array load turple is issued, via the exit to POP.STD. If the subscript contained a subscripted array in its calculation, an xmit operator is issued, to force an intermediate. This prevents a register deadlock condition in OPT=0 (and the operator will be squeezed out by OPT≥1). If the number of subscripts is not the same as the array dimensionality, a diagnostic is issued and exit is to PAREXIT.

ADU: ADT linkage subroutine. Called from the subscript processing routines to set up conditions for successful turple emission via ADT.

ASE: Add subscript expression. ASE is called by A=ARRAY to add the value of the current subscript expression to the cumulative offset being calculated by the subscript processor. If the expression is constant, it is merely added to accumulating bias and exit is made. If the value of the expression is a variable, the variable is added via a turple to the non-constant accumulation, represented as a variable or an intermediate. If no such accumulation

exists, the variable becomes the saved representation. If the subscript expression is an intermediate, analysis is made, attempting to reform the expression to remove constant portions (and move them into the constant bias). When those optimizations which can be performed are done, the resultant intermediate is treated as the variable case. If necessary, a turple for the add is issued, via ADU.

GDI: Get dimension information. GDI is called by the subscript processors to extract information about a given dimension bound for an array. GDI returns dimension count, the upper and lower bounds for the dimension and flags concerning these bounds.

MSP: Multiply by dimension span. MSP is called by A=ARRAY to multiply the accumulated offset by the span of the current dimension. If the span is constant, the accumulated bias is inquired. If constant, a compile time multiply can be performed. If the bias is not constant, then a multiply turple must be issued. Analysis is performed to optimize the turples making up the offset calculations and constants are reduced as applicable. If the span was variable, the multiply turple must be issued.

SLB: Subtract lower bound. SLB is called by A=ARRAY to subtract the lower bound value from the subscript expression. GDI is called to get the dimension information. If the lower bound is constant, it is merely subtracted from the accumulated constant bias. If variable, a subtract turple is issued, via ADU.

SSO: Set up subscript operations. SSO is called by PAR.SUB to set up the ARGMODE cells for subscript processing.

SSR: Standardize subscript result. SSR is called by C=ARRAY and A=ARRAY to standardize the subscript result. The mode is tested. If illegal, a diagnostic is issued. If not integer, the mode is coerced to integer. If the subscript is constant, semantic tests are made regarding magnitude and relation to upper and lower dimension bounds. Irregularities are diagnosed.

h. ACT: Add converted turple. ACT is called to output a turple which requires no mode field. The proper header is fetched from the PRIOR table and the priority fields are cleared. Calls to SDM, OMC and ADT result in the turple output.

i. ADT: Add turple to T.PAR. If the current skeleton is reducible, CCR is called to attempt constant reduction. If the turple was reduced, the constant value (TP. format) is placed on ESTACK and exit. SQZ is called to determine if an exact turple exists in the current sequence. If so, the intermediate for that turple is placed on ESTACK and exit. If the turple is to be issued, an intermediate is formed for it and the turple is emitted, via EMT.

j. CBB: Convert to base/bias. A filter routine for ADT to convert operands to base/bias (via BBC).

k. CDI: Check data interference. CDI performs the mechanics of the test for character replacement left side also being on the right. Called from POP.EQL.

l. CIL: Check illegal level 3 usage. Called by PAR.VAR when a variable name is encountered. If the variable is LEVEL 3, the current ARGMODE must permit its usage or a diagnostic is issued.

m. CLC: Coerce length of character result. CLC is called by CLM to coerce the length of a character expression. GOL is called to determine the expression length. If the length of the expression equals the desired length, exit. Otherwise, a character substring is issued, via ECS and a string operator turple is issued via ADT.

n. CLM: Coerce mode and character length. CLM is called from the statement function interface routines. The mode of the result is coerced to the desired result by CMR. If the mode is character, the length is coerced by CLC.

o. CMR: Coerce mode of result. CMR is called to coerce the mode of a result to a desired mode by such operations as replacement. The result mode and desired mode are tested via the MODC table. Illegal coercions are diagnosed. If no conversion is necessary, just exit. If the result is constant (determined by LCT), KCV is called to perform the conversion. NBC is called to register the new constant. If the result is not constant, a conversion operator turple is issued via ADT.

p. COR: Check if operand reducible. COR is called to determine if the input operand is the result intermediate of the last turple emitted. If so, the operand may be reducible.

q. DBE: Dimension bound error. When PARMODE is PM=DIM, certain parse conditions become illegal. DBE issues the diagnostic required, first setting up variable diagnostic FILL. cells.

r. DDC: Diagnose double and complex operands in an expression. Issues an ANSI diagnostic for mixed double and complex values in an expression.

s. DOA: Determine operand addressability. DOA is called to recognize the addressability of an operand (not a constant or expression). LCH is called to determine if the operand is a constant. If so, exit not addressable. If the EXPR bit is set, also exit not addressable. If the operand is INTR, determination is made if it is an array load or substring reference. If so, process the array or variable name. Otherwise, mark not addressable and exit. The variable or array name (supposed) symbol table entry is fetched and if not a label, the WB. is shifted to the VAR position and exit is made.

t. ECC: Emit character constant. An operand doesn't have enough fields to represent a character constant so it must be treated as a special case substring. ECC calls ECS to process the constant as a substring.

u. ECS: Emit character substring. ECS is called to output a character substring operator. Two turples are issued, a colon operator and a substring operator.

v. EMT: Emit turple to table. EMT (called using the EMIT macro) is called to output a turple to T.PAR or register some other table. The operator is in coded form in register B3 and the operands and output table (as necessary) are passed in registers. The coded operator can take two forms:

```
+-+-+-----------------+
!1!T!      ADDR       !
+-+-+-----------------+
 1 1         16

T   :   Output table (X1,A1) indicator
ADDR:   SETOP word address

+-+-+-+-+-------------+
!0!T!0!D!    SKEL     !
+-+-+-+-+-------------+
 1 1 2 2       12

T   :   As above
D   :   Buc. indication (for OPT=0)
SKEL:   Skeleton ordinal
```

EMT decodes the operand information, and based upon that, forms the TH. operator. T.PAR or the supplied table is allocated space for the turple and the operator and operands are moved to the table.

w. FAT: Flush argument turples. FAT is called by IOJ (see IO, 3.11) to move the IO APlist turples from T.IOARG to T.PAR at the end of an IO statement, or on a restart call.

x. FSA: Find symbol attributes. FSA is called by VEL to extract symbol attributes for argument expressions which are symbols only. The T.SYM WB. word is returned if the expression is a symbol, else an indication that it was not.

y. GOL: Get character operand length. If the operand is a character variable, the length, or indication that it is variable is returned. If the operand is a substring expression, and the substring operands are constant, the length is determined. If the operand is concatenation, only the fixed length or passed length attribute is known.

z. OMC: Output mode conversion. Called as a preliminary to outputting a turple where mode is a factor. The OMC call is preceded by a SDM call, which determines if conversion is necessary. If no conversion is needed, exit. Otherwise, determine the operand to be coerced and perform the coercion via CMR. Upon exit, the original operator is preserved and the operand not coerced is preserved. The coerced operand is replaced with the intermediate of the coercion turple.

aa. SDM: Set dominate mode. SDM determines if mode conversion may be necessary. Some turples have a required mode and some inhibit conversion. Many operators don't need conversion. The operands are tested and if the same mode, no conversion is necessary. If the modes are incompatible, a diagnostic is issued. Otherwise, the greater mode is selected dominate, and the operand of that mode is marked as the dominant operator.

bb. SPE. Skip parenthesized expression. Called when a symbol is invalidly followed by O.LP. The tokens in the token buffer up to and including the matching O.RP are skipped.

cc. SQZ: Squeeze operation if possible. SQZ scans T.PAR backwards to the CURST limit (last sequence break) trying to match the current turple to be output with an existing turple. Match must be 60 bit on the operator and the two operands. If a match is found, the turple will not be issued and an intermediate pointing to the duplicate turple is formed and added to ESTACK.

dd. STC: Set target characteristics . STC analyzes the left side of a character replacement statement, saving information to test (by CDI) for character data interference. Called by POP.EQL.

ee. TPC: Test for passed length character concatenation. TPC is called when such a construct is illegal. The TP.CAT and TP.LCF bits of the operand are tested, and if both on, a diagnostic is issued.

F14

3.13 CONRED: Compile Time Arithmetic

Abstract: CONRED consists of routines to convert DPC to binary constants, perform compile time constant arithmetic (including coercion and reduction) and routines to simulate implied do's in DATA statement variable lists.

Interfaces: CONRED is a front end deck and resides on overlays (0,0), (1,0) and (2,1). The routines of CONRED are called by many of the front end translators, most notably PAR. The DATA statement processor depends heavily on the simulation routines.

Data Structures:

CONRED defines the following structures:

a. Skeletons: The code skeleton macros and definitions are provided by comdecks SKPSET, SKPCONQ and COMSEIS (described in FSKEL, 3.15). The definitions are used in compile time arithmetic routines.

b. Cells: Flags, scratch storage and templates for use during compile time arithmetic.

c. KMOD: A matrix of constant conversion routines, giving routine offset, no conversion indicator or illegal conversion indicator. Used by KCV.

Routine Descriptions:

a. DEC: Convert decimal (DPC) constant to internal binary. DEC translates the tokens which can make up an arithmetic constant (not octal or hexadecimal) and converts them to binary. The integer and fractional parts are converted to binary (conversion depends on existence) and the exponent is translated, as applicable. The floating point conversion is performed by FSCALE, which is common to the CDC product set (thus all converted constants are binary compatible). DEC returns the constant mode and the upper and lower (for double precision and complex) halves of the binary constant. Any conversion irregularities are diagnosed.

b. DTA: DPC to ASCII conversion. DTA accepts a single DPC character and converts the binary value to the corresponding ASCII binary value. DTA uses the table DTACT, which is a D->A conversion matrix.

c. OCT: Convert octal/hex constant (DPC) to binary. OCT is called to convert O.OCT and O.HEX tokens to internal binary. OCT takes the input tokens and converts to binary and accumulates by shift and add operations. If any non-octal or hex digits are encountered, a diagnostic is issued. Octal constants are limited to 20 digits, hexadecimal to 15. Excess digits are truncated, with diagnostic. The binary of the constant is returned.

d. TNK: Translate numeric constant. TNK is called to convert any numeric constant. If the initial token is O.CONS or O.PERIOD, DEC is called. If the token is O.OCT or O.HEX, OCT is called. Otherwise, a diagnostic is issued.

e. FSCALE: Floating DP conversion. FSCALE is the CDC standard floating point conversion routine. It accepts 114 bits of significance, an exponent and sign and returns a double precision floating point (rounded) value. Provided by comdeck FSCALE.

f. CCR: Compute constant reduction. CCR is called to attempt reduction of two constant operands (or of a unary constant operand). LCT is called to test the operands. If either is not constant, exit with no reduction. Otherwise, call PCA to perform the compile time reduction.

g. CTA: Perform compile time arithmetic. CTA fetches the current operation skeleton and analyzes it via repeated calls to ISI (until the skeleton has been completely scanned). If reduction is performed, the new constant is registered via NBC. If any error or irregularity which would preclude reduction is noted, an exception exit (CTA.ER) is provided and is used by subordinate routines.

h. PCC: Perform character concatenation. Called by POP.CAT to concatenate two character constants. The combined length of the existing constants is determined and the required space is allocated on T.CON. The first operand is moved into place, using MVE=. The second constant is moved in behind the first, using MNS=. The result is blank padded, as required. NCM is called to determine if the new constant already exists on T.CON. If so, the constant just formed is scratched. The T.CON index and the constant length are returned.

i. PCR: Process character relationals. The T.CON indexes and lengths of the two character constants are provided to PCR, as well as the relational operator. CCS is called to compare the strings and a relational value is returned. That is compared with the relational operator for the expression and a logical constant (true or false) is determined. NBC is called to process this constant and the result is put on ESTACK.

j. ISI: Interpret skeleton instruction. ISI is called by PCA to interpret a skeleton instruction, performing at compile time the operations which would be done at run time (for results compatability). The current instruction in the skeleton being processed is burst into components (using PIK; see PUC, 2.3) and the instruction reconstructed in a form usable by this routine. LOP is called to load the operand and that information is factored in. COL is called to test the operand legality. The operation is then performed. (Intermediate results are saved.) If the operation was floating, range and indefinite conditions are tested and if true, exit to PCA.ER. Otherwise, normal exit is made.

k. LOP: Load operand. LOP is called by ISI to fetch the proper operand. Based upon the skeleton form, a branch table is entered and thence to an entry which provides the correct operand (or intermediate result) address. The branch table is produced by comdeck SKOP.

l. COL: Check operand legality. COL is called by ISI to test the validity of intermediate values produced by the ISI interpretation. If the instruction is UXi, CFO is called to test legality. If the instruction is PXi, CIO is called to test legality. If the instruction is not floating point, exit. Otherwise, call CFO to test both operands. If the instruction is DXi, if divide by zero, exit to PCA.ER. Else exit OK.

m. CFO: Check floating operand. Called by COL to determine if operand is out of range or indefinite. If so, exit to PCA.ER.

n. CIO: Check integer operand. Called by COL to determine if an operand is too large (>48 bits). If so, exit to PCA.ER.

o. <u>KCV</u>: Convert constant value. Called to convert a constant from one mode to another. The constant binary, old mode and desired mode are supplied. Table KMOD is used to extract the proper conversion processor and the conversion (if legal) is performed. The binary of the new constant and an indication of the conversion performed is returned.

p. <u>CCS</u>: Compare character strings. CCS is passed the address of two strings and their lengths. The strings are compared, one character at a time (via GNC) until the strings fail to match, or until both strings are exhausted. An indication of the relative values of the strings is returned.

q. <u>GNC</u>: Get next characters. GNC is called by CCS to fetch one character from each of the strings being compared. When either string is exhausted, the count is kept at zero, and blanks are supplied. The characters are to be compared by ASCII value (provided by DTA) and that value for each character is returned.

r. <u>LCH</u>: Load value of constant. LCH tests an operand to be a constant and returns values and indicators depending on whether the operand was a constant or not. If the mode is not character, the operand is tested to be TP.SHRT or to have ordinal S=CON. If so, a constant indicator and the binary of the constant are returned. Otherwise, nonconstant indicator is returned. If the operand is character, a constant is represented by a substring turple. If the operand is not intermediate or the intermediate is not to a substring turple, return with nonconstant indicator. The substring operand is then test for ordinal S=CON. If not, exit nonconstant. The second substring operand will now point to a colon turple. This will provide the length information, which is extracted via LCT calls. The return value will be a pointer to the constant in T.CON and the binary of the length.

s. <u>LCT</u>: Load binary of constant. LCT is called to provide the binary value of arithmetic and logical constants. If the operand is type character, intermediate or not TP.SHRT or S=CON, a non constant indicator is returned. Otherwise, the binary and nature of the constant is returned.

t. <u>LIR</u>: Load integer of real. LIR is passed a floating point value. Determination is made if the value is an exact integer (no fractional part). If so, the integer value is registered via NCS and the binary and TP. operand format are returned. Else, a failure indication.

u. NBC: Enter binary constant. NBC is passed the binary and mode of a constant. If single word, NCS is called to register the constant. Otherwise, the constant is entered into T.CON via NCM. The TP. operand of the constant and the mode are returned.

v. NCS: Enter single word constant. NCS is passed the binary and mode of a constant to be entered (into T.CON). The binary is tested, and if possible a short constant is formed (TP.SHRT, TP.BIAS contains binary). Otherwise, the constant is scanned into T.CON. If in table, the index is used for the operand bias. If not in table, ADW is called to add to T.CON and that index is used. Using S=CON as the ordinal, a TP. format operand is formed. Either form, short or long, will be returned as the operand.

w. SED: Simulate execution of DATA statement turples. SED is called by PVL (see DATA, 3.7) to process the turples generated for a data variable list. SED operates as a loop, processing each turple produced by CVL. The F.SCT table (see FSKEL, 3.16) is used to interpret the skeletons, using VS. format (see section A-2). Each turple is analyzed in turn, and if reducible, CDR is called to provide constant reduction. If not reducible, the data reduction option is taken and routines are branched to. NOTE: D=NOOP, D=ARY, D=SYBST, D=COLON, D=BSS and D=DOBD2 are merely returns to the loop top (SED.RTN) to fetch another turple. Error conditions result in an exit to SED.ABT where a diagnostic is issued and the process is ended. Normal completion of processing exits to SED.END to tie up loose ends before return to PVL.

x. CDR: Compute constant reduction. Called from SED on a reducible skeleton. Calls LCD to load the operands and PCA to perform the compile time arithmetic (if all was well with LCD).

y. LCD: Load constant (for SED). Called by CDR to load a data interpretation of a constant value, which may be a constant, the current value of an interpreted turple or a loop index variable. If the mode of the operand is not integer (or Boolean) exit to SED.ABT. LCT is called to determine if the operand is constant. If so, return the constant value. If the operand is variable, determine if it is a loop index which is stored into. If not, exit to SED.ABT, else return the value. If the operand is intermediate, return the value of the intermediate (in the turple header).

z. D=DVI: Entered from SED to process a data variable item. The item may be a scalar, an array, an array element or a substring reference. If a substring reference, EDS is called to evaluate the data scalar, LCD is called to evaluate the substring start and EDI files the address and count. For the other cases, EDS evaluates the scalar and EDI files the address and count. Exit is to SED.RET.

aa. EDS: Evaluate data scalar. Called by D=DVI to evaluate a data variable item. LCD is called to determine the number of elements represented. If the item is an array, LCD is called to get the index. ECB is called to evaluate the constant bias. GPS produces the product of spans and character length. (Note that non arrays have a T.DIM pointer of 0, which points to a dummy dimension information entry, describing an array with one dimension, one element, so counts will be okay for scalar variables.) If a dimension bound error occurred, exit to SED.ABT.

bb. GPS: Get product of spans (and character length). GPS extracts dimension bound information from T.SYM and T.DIM and character length information if the variable is type character. Returns product of spans in words and elements, character length and double/complex indicator.

cc. D=EXP.I: Entered from SED to process an integer exponentiation turple. LCD is called to get the exponent and again to get the base value. EXD is called to produce the value of the exponentiation and LCT tests the constant. Exit to SED.STO.

dd. D=STR.I: Entered from SED to simulate a store turple. LCD is called to get the right side and the store is simulated by SDV. Exit to SED.RTN.

ee. SDV: Store data value. SDV is called when a simulated store is to occur in a data list control variable of some sort. The variable and its store value (coded) are added to T.DVV, unless the variable is already present, in which case, the value field is updated.

ff. D=DOBS: D=DOBL D=DOBZS D=DOBZL Entered from SED when any do begin turple is encountered. The simulated do top address is stored in the branch field for DO.n. The number of trips is extracted by LCD and if negative, exit to SED.ABT. Otherwise, SDV is called to 'store' the trip count in DC.n. and exit is to SED.RTN.

gg. D=DOC.S: D=DOC.L Entered from SED when a do conclusion turple is encountered. LCD is called to obtain the control index and again to get the increment. The values are added and SDV is called to update the index. LCD is called to get the trip count value and that value is decremented by one. If the trip count value is zero, loop simulation is over, exit to SED.RTN. Otherwise, call SDV to update the trip count. The turple pointer is reset to the branch field of DO.n. and exit is to SED.RTN.

3.14 STMTF: Statement Function Processor

Abstract: STMTF consists of a routine to process statement function definitions.

Interfaces: STMTF is a front end routine and resides on overlays (0,0), (1,0) and (2,1).

Data Structures:

a. Cells: STMTF uses several scratch calls (in FEC) during processing of statement function definitions.

b. SP.: Dummy argument format for statement functions.

```
        +----------------------------------+-+-------------+
        !                                  !M!             !
        !                                  !O!             !
    SP. !              SYM                 !D!     CNT     !
        !                                  !E!             !
        +----------------------------------+-+-------------+
                       42                   3      15
```

SYM : Symbolic name
MODE: Argument type
CNT : Argument usage count

Routine Description:

SFD: Statement function definition processing. Enter from LEX when the statement type is determined to be 'statement function'. SSY is called to determine if the function name is in T.SYM. If in the table, semantic tests are made for conflicting declarations and any noted result in exit to a diagnostic processor. The argument list is scanned (in the token buffer) and the arguments produce two word entries on T.SCR, a starting SP. format entry and the symbol table entry. Invalid usage as dummy arguments is flagged as is duplication of any argument. Upon completion of the argument list scan, assuming no syntax errors were diagnosed, the function text is scanned. Each variable token encountered is tested against the list (on T.SCR) of dummy arguments. If a match is found, the argument usage count (SP.CNT) is incremented and the O.VAR token is replaced by an O.STFA token. Processing continues until the text has all been processed (O.EOS). The dummy arguments are tested and any which were not referenced in the text of the function are the subject of a warning diagnostic. The symbol table entry for the statement function is made (or updated) to include mode of result, number of arguments and the skeleton pointer. The text has been converted to the skeleton by replacement of the O.VAR tokens with the O.STFA tokens. The skeleton is copied from the token buffer to T.STF. T.SCR is cleared and exit is to the front end controller.

3.15 LABEL: Statement Label and Do Statement Processing

Abstract: LABEL contains routines to process definition and reference of statement labels and routines to translate and compile DO statements and to compile end code when a DO loop is terminated.

Interfaces: LABEL is a front end deck and resides on overlays (0,0), (1,0) and (2,1). LABEL interfaces heavily with control statements and the IO list processor.

Data Structures:

LABEL defines no data structures, but uses T.BLST heavily.

Routine Descriptions:

a. CUL: Check upcoming label. Called from the lexical scanner and ISL to translate the label field of the statement which is to be processed. The label field is examined and if nonblank characters are present, the label is normalized (leading zeros suppressed) and the label is packed in a format for processing by GSL. Non-numeric characters in the label field result in a diagnostic and a special flag set in the token buffer (in lieu of the normalized label).

b. GSL: Called from the front end main loop when a label is present. The symbol table is checked, via SSY, for the label being present. If so, and the label was previously defined, an error is flagged. If the label was just referenced, the usage is determined, and invalid definition is flagged (e.g., referenced as a format, defined on a non-format statement). The defined indicator is set in the symbol table entry. If the label is not in T.SYM, an entry is made, via ESY and the proper defined bit is set. If any errors were noted, a diagnostic is issued. If active structures (on T.BLST) are present, the label is analyzed by ALU.

c. ISL: Identify statement label. ISL is called whenever a statement label is referenced, by the pertinent translation routines. ISL is provided the statement label and the context in which it was referenced. SSY is called to determine if the label is in T.SYM. If the label is in the table, semantic tests are performed to determine if current usage is consistent with previous reference/definition. Of interest is whether the label is already defined and whether the reference/definition was as a format, executable, non-executable or do terminal label.

The tests cover the possible combination of usage and previous reference/definition 'fixing'. Invalid combinations are diagnosed. If a label is referenced as a do terminal (in the DO statement) special analysis is required to determine if the DO loop being defined is properly nested within any existing block structure (do or block if). If the label is not in T.SYM, it is added via ESY. Any new reference bits required are added to the symbol table entry. If the label may be relevant to a block structure, and a block structure is active, the label is analyzed via ALU. If cross reference is required, a T.REF entry is made and exit is to the caller.

d. PSL: Process statement label. PSL is the general exit entry for executable statements. Some of the managed tables are recovered when errors occurred. A call to CSB checks for sequence breaks. If the statement is a do terminal, PDT is called to provide that compilation. Exit is to the front end controller.

e. SDO: Setup do. SDO translates the DO statement. Entered from the front end main loop when a DO statement is encountered. Calls ASK to extract the keyword (DO has a slightly ambiguous syntax). The label is extracted by ASL. The control index is tested (for its presence only) and CDI is called to translate the DO statement. Upon return from CDI, the nopath condition is tested and if the loop is unreachable, a diagnostic is issued stating this and nopath is cleared (to avoid redundant diagnostics). Exit is to the front end controller, via PSL.

f. CDI: Compile do initial turples. CDI is called from SDO and from DOB (see IO, 3.11) to translate the do statement, provide semantic testing and to issue the do begin megaturple. The fatal error count at the beginning of CDI is kept so that if errors occur, the do structure can be marked as null. ISL is called to translate the do label. If errors occurred, exit is immediate. A new do begin label (DO.x) is created, via INN. Space is allocated on T.BLST for the current do loop. Information known at this time is added to T.BLST. PAR is called to translate the control index and the induction values. Upon return from PAR, these values are on T.BLST. DTC is called (see PAR, 3.12) to determine the trip count, which is saved on T.BLST upon return. ACT is called to issue a store turple of the initial value into the control index. If errors

were noted during translation of the DO statement (or implied do) T.BLST is marked as null for a DO statement and the T.BLST entry is removed for an implied do (RBE). Exit is to caller in the error case. If the increment value is constant, it is merely kept on T.BLST. If variable, a variable (DI.x) is invented via INN and that replaces the variable on T.BLST. A store turple is issued, via ACT, of the variable into the 'made' variable. The limit value and control index are converted to base/bias format (via BBC and MDD respectively). The trip count is tested for constant, via LCT. If constant and GT zero, the one trip skeletons will be selected regardless. If the trip count is zero and one trip loops were specified, a diagnostic is issued. If zero minimum trip loops are selected, a generated label is created (and stored in T.BLST) and a goto turple is issued to skip the loop code (when the trip count is constant zero). If the trip count is constant, its length is determined, and if too long for short loop (and short loops requested) a diagnostic is issued. Based on the trip count information, the proper do begin turple is selected. A new name (DC.x) is invented, via INN, for the trip count variable and its ordinal is saved in T.BLST. Finally, the 3 turple do megaturple is issued and exit is to caller.

g. PDT: Process do termination. PDT is called from PSL and DOC (see ID, 3.11) to provide end of loop processing for DO terminals and implied dos. If the do loop terminates with unterminated if blocks on T.BLST, the if block entries are removed by RBE and a diagnostic is issued. If the current T.BLST do entry doesn't match the terminating conditions, the error is diagnosed and all do's ending on the badly nested label are scratched from T.BLST. If the do definitions contained errors a diagnostic note is issued. FBS is called to finish the structure. (for DO statements). The do conclusion megaturple is output. The T.BLST entry for the do is removed and the loop attributes are propagated outward, if necessary. If more entries are present on T.BLST, test is made to determine if the next outer structure is a do loop which terminates on the same label as the just processed do. If so, loop back and continue processing. Otherwise, exit to caller.

h. ALU: Analyze label usage. An entry on T.BLST consists of fixed information (format) and variable information. The number of LA. words is variable. ALU analyzes the labels which are referenced or defined within a block structure and either adds LA. entries or modifies existing entries as required. If a label is being defined and the structure is block if, if the label was previously referenced, it must have been within the current arm or an error is output. A reference within a block if arm is merely marked as such. If the label is being defined and is within a do-loop, if it was referenced outside the loop it is marked as a loop entry. If the label is referenced within a DO loop, if it has been defined, its location is determined. If defined within the loop, no action. If outside the loop, mark as loop exit. If the label has not been defined, merely enter or update the LA. word. Adding a word (LA.) involves inserting in T.BLST and then incrementing the LC. word (LC.CNT) and replacing that entry.

i. DDR: Diagnose do index redefinition. Called when a do control index might be redefined (e.g. assignment left side, input list item, etc). First it is determined if the redefinition can be determined. If not exit. Also exit on any character operand. The operand is converted to base/bias format (as are all do control indices). If T.BLST is empty, exit. Otherwise, scan the table, checking the control index word of all do loops for a match with the current operand. A match means redefinition and a diagnostic is output.

j. FBS: Finish block structure. FBS is called when a do loop terminal is encountered (from PDT) and when a segment of an if block structure is complete (ELSEIF, ELSE, ENDIF, see KEY, 3.5). The nature of the block structure finished is determined. The current segment is copied to T.SCR and removed from T.BLST. If the current structure is a block if segment, the LA. words (if any) are scanned, and any labels defined within this if segment are marked unreachable (WB.INA). If there is an outer block structure, the LA. words are merged into the outer block segment one at a time, using ALM. Finally, the segment on T.SCR, less all the LA. words is copied back to T.BLST (for use by the calling processor). Exit to cleanup. If the current structure was a DO loop, the LA. words are again scanned, merging properties of the loop labels (entry, exit). When complete, the DO.n label is updated with information gleaned. If the loop has

an exit, it is marked. Entries are marked. Potential entries are noted. If the loop has an entry and no exit, a fatal diagnostic is issued. An entry with an exit is flagged nonANSI. If the loop has no exits and no legal entries, mark its DO.n as closed and mark all defined labels as unreachable. If potentially legal entries are possible, link them via LPE. If an enter block segment exists, merge in the current blocks LA. words, using ALU. Finally, restore the current block, less the LA. words to T.BLST. The cleanup exit clears T.SCR.

k. PDA: Propagate do loop attributes. PDA is called when some condition (e.g. subroutine call) need to be noted in the do loop header. PDT scans T.BLST in reverse, updating the DO.n symbol table entry with the required attributes.

l. RBE: Remove block entry. Called when something dire has happened to structured control statements. Normally an invalid nesting of do loops or an invalid nesting of do loop and if blocks. An indicator of the structure involved is passed to RBE. If an invalid if block, FBS is called to finish off the block and then the structure segment is scratched. If the structure is a do loop (DO statement), all loops terminating on the same label are discarded. Each segment is copied to T.SCR, unless it matches the terminal label of the discard do. When the entire T.BLST has been processed, the segments are copied back to T.BLST. If the structure was an implied do, T.BLST is shrunk back to its original size at the start of the statement containing the implied do (the entire implied do nest is discarded).

3.16 FSKEL: Front End Turple and Opcode Values

__Abstract__: FSKEL contain code skeletons and turple definitions used by the front end translators when OPT>O.

__Interfaces__: FSKEL is a special front end deck and resides on overlay (2,1).

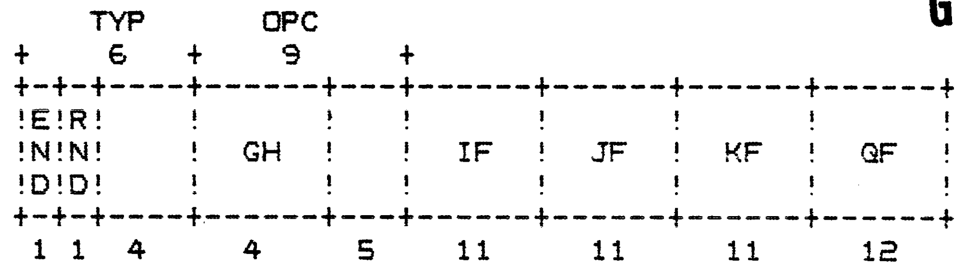
__Data_Structures__:

FSKEL consists of two basic parts, macros to define skeletons and the skeletons.

a. __Macros__: Provided by comdeck COMFSKL. The following macros are defined:

DEFINS	:	Define instruction
FORM	:	Form instruction skeleton element
SETCON	:	Set number/address field in skeleton (con)
SETOTH	:	Set number/address field in skeleton (non-con)
DEFPO	:	Define pseudo opcode
ENDS	:	End macro skeleton
ENDF	:	Flush last skeleton word
SETSPC	:	Set special skeleton
RESET	:	Reset current skeleton
BRANCH	:	Continue skeleton elsewhere
MICNAM	:	Generate micro of skeleton name
SKEL	:	Declare beginning of skeleton expansion
SUBSKEL	:	Declare beginning of sub expansion
SKEQU	:	Equate skeletons
SYBEQU	:	Declare equivalent pass 2 skeleton
CALL	:	Call external processor to process or partially process current turple
MICMIC	:	Get micro of a micro

b. __Skeletons__: The skeletons are provided also by comdeck COMFSKL, with help from imbedded comdecks COMSEIS, SKPSET, SKPCONQ, SKOP, DEFINS, SKEL and PARSKEL. The skeleton format is:



END: End of skeleton (last word) marker
RND: Rounded operation
TYP: Type of operator
GH : gh field
OPC: opcode
IF : i field
JF : j field
KF : k field
QF : q field

4.0 QUICK MODE CODE GENERATION ROUTINES (QCG)

G13

Because this section was produced independently of the bulk of the FTN5 IMS, the format differs from the other deck and routine descriptions. The decks included in the QCG group are:

QCGC
QSKEL
FUN
REG
GEN

Preface:

Since the particulars of the Quick Code Generator (QCG) are complicated while the general design is relatively simple, this discussion will start with an overview of the main algorithms and structures. It will attempt to explain first how QCG does what it does in a main case situation. Later on the discussion will turn to the obscure but necessary mechanisms of code generation. These will be discussed in a case by case, deck by deck, fashion. Hopefully, this will provide the new reader with the overall understanding needed to appreciate the causes of the algorithmic complexity which occurs at the code level.

What QCG Does:

The job of QCG is to turn incoming segments of intermediate language (I.L.) into FORTRAN'S own assembler language called prebinary and aplist. To do this QCG chooses minimal code sequences for incoming operator-operand triples in the intermediate language. It then fills out these minimal sequences with any necessary loads of the operands into registers. The necessity to load an operand is determined from the object program environment information which QCG continually updates and interogates. Thus on a triple by triple basis, QCG generates equivalent sequences of object code.

Elementary Definitions, Notation, and Example Turple (OPR, 1OP, 2OP):

A turple is an ordered operator-operand triple, (OPR,1OP,2OP). It is the basic unit of input to QCG. OPR is some arithmetic, character, logical or Boolean operation or some assembler directive to be applied to 1OP, the first operand, and 2OP, the second operand.

Example 1: The source expression I + J results in the turple (integer +,I,J).

Note: Each turple determines a unique minimal or "skeletal" code sequence. This is the main function of OPR. In special cases the determination requires information from the operands.

Intermediate Language (IL):

An intermediate language segment is a sequence of turples which is semantically equivalent to an integral number of FORTRAN source statements. The IL is also called the parsed file. (T.PAR)

Prebinary: Machine instructions and psuedo ops are passed to the FORTRAN assembler as prebinary. Prebinary contains all the opcodes, register numbers, constants and variable designators needed for the assembler to generate loadable binaries. Prebinary instructions will always be designated by their COMPASS mnemonics, i.e., IX7 X3-X2 instead of 37732. Macros and Psuedo ops will appear as they do in the object listing.

Aplist (Actual Parameter Lists):

Whenever there is a procedure call, actual parameter lists are generated to designate the actual parameters that will be substituted into the formal parameters of the procedure definition. These lists contain all the designators of length, relative offset, address and type needed by the assembler to set aside and format storage in the current suitable form for procedure calls.

Example 2: The turples generated by CALL P(A)

will be:	(APL,A,0)	aplist turple (2OP = ZERO)
	(SUBR,P,1)	subroutine turple 2OP = # aplist items

Resultant aplist is:

AP.n	APL	A	where this is the nth aplist to occur in this program unit

Example: A turple prescribes a skeletal code sequence which is unique up to register number. But this in no way implies that the same turple will always produce the same object code.

Let ALPHA be a source sequence containing the statement A = B. Suppose the IL (ALPHA) = [...,(=,A,B),...] where (=,A,B) corresponds to B = A. Possible object code follows.

Case i: (Minimal code case) assume that when QCG starts generating prebinary for (=,A,B) , the object time environment status indicates (A3) points to B and the value of A is in X6. Then we will get only a SA6 A3.

Case ii: (Maximum Code). If the object program status indicates that neither A nor B are in registers and furthermore X6 and X7 are jammed with "intermediate" results which must be saved for later use then QCG will generate:

SA1	A	load A
SA6	ST.n	save contents of (X6) in the nth temporary storage location

```
BX6     X1                        prepare for store to B
 .
 .
 .
possible intervening
code from next turple
 .
 .
 .
SA6     B                         delayed store to B
```

Example 3 demonstrates that, while QCG may be quite dumb, it does try to avoid pointless code bulk. This makes the map:

TURPLE   Object Code Sequence

a non-trivial matter to predict. The best one can do is identify the minimal code sequence, locate the pertinent operand loads, and thereby identify the code resulting from a given turple. This process is very tedious, but it is the only method available for deciding, "which turple caused the bad code," when bugs occur.

Example 4: Use the control card

FV,LO=O,SNAP=A

The program on file INPUT should be

```
PROGRAM TEST
I = J + K
END
```

FV should be the current, high cycle, FTN5, test mode compiler.
(1) Try to find the turples in the SNAP=A listing and the object code in the object listing, which pertain to I = J + K.

Minimal Code Sequences, Skeletons, and the Fundamental Algorithm of GEN.

QCG is a table driven program. The driving tables are in the code skeleton deck QSKEL which is generated by the Comdeck SKEL. The preceding section made a point of emphasizing the minimal code sequences associated with a given turple. It is in QSKEL, in the code skeletons where these sequences are laid out. The main flow that generates them is completely determined.

Lets define our terms.

Skeleton Word: A skeleton (or instruction) word is a word in QSKEL which is generated by one of the instruction macros, a BRANCH, or a CALL macro. Skeleton words select the order of code generation call. They cause branches to other skeleton words and calls to special processors. Most of the time, they control the construction of a target prebinary instruction which is one of the minimal code sequence instructions for an associated turple. This construction can often involve load code to guarantee the contents of the operand registers of the target instruction.

Example 5:

1. BRANCH DOC1,(T1)

This skeleton cause a branch to the skeleton word at DOC1 (do close 1). The temporary result in (T1) will be preserved.

2. CALL GAP

This skeleton word calls the special (general aplist) processor GAP.

3. FMR R1,L1,L2

```
This causes an  FXi  Xj*Xk
                RXi  Xj*Xk
```

to be generated depending on whether the round option is selected or deselected. Preceding this target instruction will be a load of 1OP into Xj and a load of 2OP into Xk if QCG deems it necessary.

Code Skeleton: A code skeleton is a table in QSKEL which begins with a SKEL and ends with ENDS macro call. It's body is a sequence of skeleton words. A code skeleton contains all the jump addresses, jump table indexes, operand references, register constants and miscellaneous constants needed to generate the code for an associated turple. In most cases, the minimal code sequence is stated symbolically within the body of the skeleton. Whenever this is true the minimal code sequence is expressed in the instruction macro calls.

Code skeletons are the link between the IL and QCG's assembler code. Each turple operator specifies a skeleton which in turn drives the generation flow through QCG.

Sub-skeleton: Sub-skeletons are like skeletons in all respects but two: (1) the header is a SUBSKEL macro; (2) they are never referenced directly by turple operators. Instead, sub-skeletons are referenced by special processors which are called by skeletons or another sub-skeleton. The way things usually work is:

1. A skeleton is called by the operator.

2. The skeleton calls a special processor.

3. The processor selects a sub-skeleton on the basis of operand properties.

See integer multiply.

EIS (Expand Instruction Skeleton):

Code generation begins in earnest in EIS, the main subroutine of the deck GEN and the controller of QCG. EIS is a triply nested loop. The outer level steps through the IL, turple by turple, and uses the operators to select code skeletons. The second level steps through skeletons, word by word, determining the processor type for each skeleton word. If a skeleton word is a normal instruction type, EIS drops to its lowest level. The lowest level shifts through the skeleton word satisfying the opcode, constant, address and i, j and k fields needed for a prebinary instruction.

General Flow:

Upon entry to EIS the current turple pointer is to the first turple of the IL.

I. Get a code skeleton pointer from the OPR of the current turple. If OPR = 0 then stop.

II. Select one of the paths A, B or C depending on the TYPE of the current skeleton word:

A. BRANCH type: determine a new skeleton word pointer in a new skeleton and go to II.

B. CALL type: Call a special processor. A special processor will:

1. Output aplist and/or prebinary.

2. Set up necessary generated operands.

3. And do one of the following:

a. Bump the turple pointer and go to I.
b. Select a subskeleton pointer and go to II.
c. Go to III.

C. Instruction type (several): Save the skeleton word opcode field; loop through the EIS register and constant processors satisfying the k or q, j and i fields of the instruction in that order.

III. If we are on last word of the skeleton, bump the turple pointer and go to I. Otherwise, stop the skeleton word pointer and go to II.

Example 6: Arriving at the turple (*,A,B) EIS will choose the following flow.

I. The * operator will be interrogated and the type real multiply skeleton will be read up. (Done at EIS.PNS)

```
MUL.R   SKEL
        FMR     R1,L1,L2
        ENDS
```

II. The TYPE of the first and only skeleton word is a normal instruction so we take path C. (Choice made in EIS.LNX)

PATH C: Before begining this discussion let's make some assumptions about the object program status so that we can predict some likely output code for this turple. Let's assume that rounding is turned off, B is in X1, A is not in a register and A3, X3 and X0 are free registers. Begin processing:

1. Strip off the opcode (floating multiply) and save the result in OPCODE. Position the k portion skeleton register field.

   Note: Each instruction type skeleton word has skeleton register fields for its i, j and k portions. These fields contain an ordinal into an EIS processor jump table (SKOP Table) and a micellaneous number of fields which pertain to operand number, result number, temporary result number, etc.

2. We are at the top of the (i, j, k) field expansion loop (EIS.NX in the code). This loop supplies register numbers for the i, j and k fields of our floating multipy instruction. As we proceed the register numbers are saved in the i, j and k fields of the compiler cell INS.REG. We are processing the k field. The k field of this instruction word (FMR R1,L1,L2) is given symbolically by L2. This indicates that the EIS processor designated would be a "check load" type (in fact EIS.L) and that the second operand is being processed. (Later, a precise method will be given to know the EIS Processor indicated by an instruction macro parameter). When control passes to EIS.L it will be discovered that the second operand, B, is already in X1. So, no load is needed. The k field of INS.REG is set to 1, the skeleton register field of the j part shifts into place and we go to EIS.NX.

3. We are now processing the j field. The instruction word parameter for this field is L1. This sends us back to EIS.L to check the status of A, the 1OP of this turple. We soon find that A is not in a register. So, A is loaded into the available load register pair (A3, X3). We emit the prebinary - SA3 A. The value 3 is passed to the j field of INS.REG, the skeleton register field of the i part shifts into place and we are headed back to EIS.NX.

4. The i field is represented by the parameter R1. This sends us to EIS.R to process the first and only result of this turple. EIS.R will select a result register, say X0. It then calls DIT (Define Intermediate Result) in REG. DIT makes the updates to the object program environment information which will indicate that: X0 has recieved the result of a floating multiply of the contents of A and B. The 0 from the X0 is passed to INS.REG's i field, and we return to EIS.NX.

5. At EIS.NX it is discovered that all the fields are filled. We jump to EIS.CMP for clean-up. Here the contents of OPCODE are combined with those of INS.REG to make the prebinary word: FX0 X3*X1. We output this instruction, see that this is the last skeleton word, step the turple pointer and start again.

Reading the Skeletons:

Figure 1: Call Formats for the Key Macros

Skeleton Macros:

```
AD      BRANCH      TO,IJK
AD      CALL        TO,ARG

AD      (inst)      I,J,K
AD      (inst)      I,JK
AD      (inst)      I,J,Q
AD      (inst)      I,J,S=... or =XS=...
```

Notation: (inst) can be any skeleton instruction type macro. See comdeck DEFINS in QSKEL for a list.

The SKOP macro:

```
IJK     SKOP        NOTLAST,LAST,parameters
```

How To Use the Instruction Macro Calls and the SKOP Comdeck Listing To Predict the Program Flow Through EIS:

We are almost in a position to state a precise method of using the code skeletons as a map through EIS. All that is lacking is a description of the comdeck SKOP.

SKOP (Skeleton Operand Link)

SKOP is a comdeck of SKOP macro calls. It is called in QSKEL and GEN. This comdeck automatically maintains the linkage between the two decks. In QSKEL, SKOP defines all the assembler constants related to the instruction macro parameters. These constants include the EIS processor jump table ordinals which appear in the skeleton register fields of the i, j, and k parts of instruction words. In GEN, SKOP sets up the EIS processor jump table.

Thus we define the SKOP macro twice for two contexts. In the first, SKOP acts on the instruction macro parameter types and in the second SKOP sets up processor references. This dual use results in the following important fact: Every SKOP macro call includes both a reference to a specific instruction macro parameter type and designators of corresponding EIS processors. This makes the SKOP comdeck automatically maintained documentation of the link between EIS and the code skeletons.

Example 7: A SKOP call

```
    L        SKOP     LNU,L,OP
```

The important parameters are the L in the location field and the LNU and L in the variable field. The location field L indicates an L type instruction macro parameter as either L1 or L2 in an (IS R1,L1,L2) call. The other L is to designate EIS.L. LNU is for EIS.LNU.

LAST and NOTLAST:

Just about now you are probably asking, "What EIS processor do I choose?" It is not exactly obvious in Example 7. The fact is that the choice of an EIS processor for a given parameter in an instruction macro call is a function of two variables. The choice depends on the parameter type and the position within the invoking code skeleton of the given parameter relative to other occurrences of the same parameter. In many cases, the EIS processor for the last occurrence of an instruction parameter will differ from the one chosen for all of the preceding occurrences.

NOTLAST and LAST are the SKOP macro parameters that designate the EIS processor. They are the first and second variable field parameters of SKOP resp. EIS.NOTLAST is the processor chosen for all but the last occurrence of a given instruction macro parameter. Obviously, as its name implies, the processor of the last occurrence is EIS.LAST.

Example 8 (Complex Add):

```
ADD.C   SKEL
        FAR         T1,LU1,LU2
        NR          RU1,0,T1
        FAR         T2,LL1,LL2
        NR          RL1,0,T2
        ENDS
```

NOTE: Open your QSKEL to the SKOP listing.

a. LAST and NOTLAST: Looking at the SKOP entry for the T1 parameter we have:

```
T   SKOP        AT,CT,TMP
```

NOTLAST=AT and LAST=CT. So, for the (FAR T1,LU1,LU2) word, control passes to EIS.AT during i field processing. For k field processing of the (NR R1,O,T1) word control goes to EIS.CT, the EIS.LAST processor.

b. Complete flow information for the word (FAR T2,LL1,LL2) within the context of the ADD.C skeleton:

(k field) - the k field is represented by the LL2 parameter and is the last occurrence of this parameter. SKOP entry is:

```
    LL      SKOP        LNL,LL,OP
```

We must use the LAST parameter, LL, to generate EIS.LL, the processor of the k part.

(j field) The j parameter LL1 yields another pass to EIS.LL for the reasons just stated. EIS.LL, incidentally, is the load checking processor for double precision lower parts and complex imaginary parts.

(i field) T2 is in the i position. It is a not last occurrence of a T type parameter. NOTLAST for T types is AT. So we go to EIS.AT. (EIS.AT sets up register values that are so temporary that they are discarded after their last use within a skeleton. EIS.CT, the last use processor, does the discarding.)

c. Constant parameters:

Up to now we have only considered parameters of the form:

```
<Parameter>  ::=  <Letter> <number>  !
                  <Letter> <Letter> <number>
```

These are normal parameters. The location field of their SKOP entry is given by the alpha part of the parameter. The following table gives the SKOP entry location field for all parameters:

Figure 2: SKOP Location Table

Parameter format	Location Field
Normal	Alpha part of parameter
6-Bit Constant	K
18-Bit Constant	Q
S=...	S

In the ADD.C skeleton in this example the (NR RU1,O,T1) word is generated using the 6-Bit constant O. The SKOP entry for this constant is:

```
K        SKOP        K,NONE,CON
```

This SKOP entry is typical for an abnormal parameter. The LAST parameter is NONE which indicates there is only one processor for this parameter. Whenever one of the SKOP processor designator parameters assumes the value NONE, then there is only one possible EIS processor and it is determined from the other SKOP parameter. Thus for (NR RU1,O,T1) the j field processor is EIS.K.

<u>Summary</u>:

To determine the EIS processor for a given instruction macro parameter, PARM:

I. Find the SKOP entry for PARM (Use SKOP Location Table)

II. IF (NOTLAST=NONE) then the processor is EIS.LAST. STOP.

III. IF (LAST=NONE) then the processor is EIS.NOTLAST. STOP.

IV. Determine relative position of PARM within the invoking code skeleton.

  A. If last occurrence, processor is EIS.LAST
  B. Else the processor is EIS.NOTLAST.

  STOP

<u>BRANCH and CALL</u>: Flow for the Non-instructions macros.

BRANCH and CALL macros also give useful information about program flow. While CALL functions are somewhat disjoint from the rest of the skeleton environment, BRANCH interacts directly with the instruction macro parameters.

When EIS.LNX encounters a CALL word, say (CALL TO,ARG), program flow passes to O=TO. Further program flow information must be determined from the special processor O=TO. The parameter ARG is formatted and passed to O=TO. If ARG is one of the normal instruction macro parameters like L1 or P3 then this appearance within a CALL call does not count as an occurrence in last occurrence computations.

BRANCH helps us cut and paste skeleton parts together. When a skeleton word generated by a (BRANCH TO,IJK) is encountered at EIS.LNX time, the skeleton word pointer is reset to TO and the EIS.LNX loop continues unmolested. The IJK parameter consists of one or more normal instruction macro parameters (for example, T1 or (T1,T2)). Parameter occurrences in IJK count as occurrences within the invoking skeleton and effect last occurrence computations. In fact, the IJK field of a BRANCH call has only one function: to insure last occurrence processing is de-selected for all of the IJK parameters preceeding the BRANCH call. IJK is just the set of instruction parameters that should be preserved across the change of skeleton.

Example 9: DOC.R - Do loop close code skeleton for do loops with a real index.

```
DOC.R      SKEL
           CALL        DOC,1
           FMR         T2,L1,L2
           NR          T1,0,T2
           BRANCH      DOCL1,(T1,L1)
           ENDS
```

Skeleton processing begins with a trip to O=DOC. Next we process the FMR and NR instructions as described in the preceding sections. We note that last use processors are not selected for T1 or L1 because they occur in the IJK field of a BRANCH call. When the BRANCH is encountered, control passes to the DOCL1 word in the skeleton DOC.L. One must always take care that the parameters in the IJK field actually do occur in the skeleton sequence referenced by the BRANCH, otherwise, last use processing for the IJK parameters will be missed. That always leads to bugs.

What To Do With a Bad-Code Bug Once the Eliciting Turple Has Been Found:

A naive approach to fixing bad object code bugs is to find the bad turple, set an IDP break at EIS.PNX which detects this turple, and then step until something goes wrong. This approach starts off well but it could take a long time indeed to step through all the code for a turple. A floating divide, for example, causes 51 passes through EIS.NX making a total of no less than 63 direct subroutine calls.

A better approach:

I. Determine the turple and find its skeleton.

II. Compare the minimal precribed code sequence of the skeleton with the actual code generated for this sequence. (It is often useful to underline the minimal code sequence as it appears in the object listing).

III. Now determine which skeleton word was active when the bad code occured. This involves more comparison of the object code with the skeleton and may be difficult. If the skeleton word is a CALL then read the O=... processor and skip IV and V.

IV. Determine if the problem occured in i, j or k field processing.

V. Now use the corresponding instruction macro parameters and SKOP to cut the field down to the exact EIS processor. Read the EIS processor.

VI. Use IDP.

The Coder:

We know now how EIS obtains a skeletal approximation of the object program. We know that turples give way to skeletons, a meaningful reduction in the level of abstraction, and we know that skeletons break the whole problem up into a sequence of well defined processes. Somehow real object code results, but to this point in our discussion we lack the link between abstract minimal code sequences and real executable assembler code. That link is the deck REG, the register manager.

REG controls the object program environment. It provides the operand load for an L1 parameter and the X-register for a T5. REG provides the numbers for and guarantees the contents of the registers involved in i, j and k field processing of skeleton words. REG knows who is where at all times, and, when the registers become jammed with useful values, REG determines the value that is most expendable. REG is the lowest level of code generation. It is QCG's coder.

The Object Program Environment:

Object program information resides in two tables: OUS (Operand Use Table) and REGFILE. The function of these tables is to answer two almost inverse questions. Given an operand, OUS knows whether that operand is in an object time register and if so which one. Conversly, REGFILE answers the question: Given a register, what value resides there? These tables are the domain of REG. They should never be manipulated outside of REG for the purpose of register management.

Structure:

REG and structure are almost a contradiction in terms. This deck is such a loose conglomerate of subtly similar yet different subroutines that the only reasonable order for sorting the listing is alphabetical. Some trends do emerge. There are status routines, for example. These routines update and query OUS and REGFILE. There are loaders and register allocators, and there are a few subroutines in a class alone. The interdependencies in this deck are fierce, and the algorithmic complexity rivals anything in the compiler. Viewing QCG from REG up can be totally mind boggling.

There is one fact that saves everything. EIS calls the shots. And this is the only perspective that works. Viewed in the context which the EIS processors suggest, groups of REG routines make sense. When working in REG, the reader should always keep this in mind. The structure of REG is really just the structure imposed by EIS.

Cautions:

1. Change as little as is necessary.

2. If you find an undocumented entry/exit condition that is heeded by at least one caller, don't change it; document it.

3. Small changes have big effects.

4. Document as you go. One comment per line is a nice start. (NOT A JOKE)

5. Avoid changing entry/exit conditions.

6. REG is the wrong place to start in QCG.

Basic Concepts:

Intermediate: Operands that are references to previous turple results are called intermediates. The turple referenced is the associate turple. The turple that contains the intermediate is the current turple.

Example 10:

Assume that the statement A = B + C is the first statement used to generate the IL segment IL. Then the first two turples of IL are (+, B,C) and (=, A,I1) respectively. The store turple, (=, A,I1), causes a store of the intermediate result, I1, of the addition turple, (+, B,C), into the variable A.

Operand Use: An operand is used by a turple whenever the code sequence for that turple requires the value of the operand at least once. Operands are used only once per turple even though many skeleton references may exist for the same operand.

Example 11:

(a) (direct use) The code resulting from (+, B,C) requires the values of both B and C so both are used by this turple.
(b) (Deferred turples and indirect use) The expression A + B(I) results in the turples

Turple 1: (array load, B,I)

Turple 2: (+,A,I1).

The intermediate I1 points back to turple 1, the array load for B(I).

When EIS hits turple 1 it calls the array load special processor O=SUBL which skips on to the next turple. The array load turple is a deferred turple. However, when the k field processor of the loating add of turple 2 discovers that it is processing I1, the intermediate reference to the array load of B(I), then it calls the array load subroutine SLD in REG. B(I) is loaded during the processing of turple 2. This load requires the value of the subscript I. I is required by code for turple 2. Therefore, I is used by turple 2 even though I is not an operand of turple 2.

REGFILE: This is the ongoing record of the object time registers. There are 24 words in REGFILE, one for each register. REGFILE entries are operand words or skeleton temporary register indicators. All used operands are recorded in REGILE when their values are placed in registers.

OUS: OUS, the operand use table, provides a direct lookup method for determining the register residency of a given used operand. It has one entry for each distinct operand in the IL which is used by some turple.

Status Word: A status word is an OUS entry. The most important content of the status words are status bits and reg-numbers field. Status bits indicate when an operand is in a register. Reg-numbers tell which one. Each status word has two status bits and reg-number fields. This is to accommodate the needs of two word data types.

OTR Format (Zero - Type - Register): The OTR of an object time register refers to its ordinal in the REGFILE. Written as three octal digits, the entries are: 000-007 for B-registers, 010-017 for A-registers and 020-027 for X-registers. The first digit gives the register number; the second digit gives the register type. The third digit is always zero and never used.

Use Count: Use count is the numeric field in REGFILE entries which is the register selection key. Entries with low use counts are selected first to be cleared. A REGFILE word with zero use count is considered to be empty. Interpretation of use count varies with entry type.

Entry Type:

Intermediate: Use count is related to operand use. During skeleton expansion, the use count of an intermediate is the number of turple uses remaining to the intermediate within the current IL. When the value of an intermediate is required for the last time by a given turple, a use total in the associate operator word is decremented by 1. This value is then placed in the use count field of the REGFILE entry for the intermediate. This will remain as the use count until the next turple using the intermediate is finished with it.

Non-Intermediate-Operands: Use counting for non-intermediates is less accurate than for intermediates. Use count is the number of times a non-intermediate occurs as an operand in the remainder of the IL. This does equal number of uses if there is no indirect use. Use count for non-intermediates is kept in the operand word itself. When a turple is finished with a non-intermediate it decrements the use count field in the operand by one and enters it in the REGFILE. Otherwise, non-intermediates are entered into REGFILE unchanged from the IL.

Temporary Result Indicators: These entries have the highest possible use counts, MAX.USEC and MAX.USEC-1. They never get selected until QCG is through with them. Their use count then changes to zero and they stop existing.

Locked Registers: A register is locked if the highest bit in the use count field for that REGFILE entry is set. This makes the use count so high that a locked register is never selected if QCG is functioning correctly.

Temp: (1) Temporary result registers that hold intra-skeleton results are called temps. They result from T and X type skeleton parameters. They have maximum use counts which means locked.

Temp: (2) When an intermediate looses its register residency before its last use (i.e., use count non-zero), it is stored to temporary storage. The intermediate is then in temp. Allocators try to avoid intermediates because storage to temp is expensive in code bulk and program size.

Register Convention:

REG respects and initiates many register conventions. Some can never be violated. Others, though not mandatory, should be respected. Entry/Exit conditions should be standard. The same quantities should always be passed in the same registers. To a great extent REG does just that. OTR, for example, is always returned in the same register, B2, from register allocators.

The three quantities that REG respects absolutely are: the turple pointer, the skeleton pointer and the skeleton word. Currently these values are kept in B4, A4 and X4 respectively. The turple pointer can only be changed legitimately for two reasons: to change the turple being referenced and to adjust for table crashes (only done in the deck ALLOC). The skeleton pointer and word should never change outside of EIS.

Though softer register conventions can not always be observed, some should always be observed. These are the register conventions used by subroutines of the same type. For example, register allocators are all obsessed with passing back an OTR that can be used by the caller. It makes sense that they all pass it back in the same register. Status routines pass back status words in the same registers. In general, soft conventions are simply those registers agreed upon by a specific type of subroutine to carry the focal data structures for that routine type. A soft convention must be an entry/exit condition.

Current Register Soft Conventions

(X5) = IL Operand or REGFILE entry

(A1) = pointer to status word

(X1) = status word

(B2) = OTR

(X6) = ORO (when (B2) = OTR)

Allocators:

When object time registers are needed for loads, stores or I-J-K field processing, one of the allocators routines is called. The task of an allocator is to find the most expendable object time register possible to satisfy the requirements of the caller. Some allocators actually clear the selected register before returning while others leave that decision to the caller. The names of allocators always end in R. Thus, they are easy to spot in the REG listing.

SFR is the heart of register allocation. It is a table driven scan routine which searches classes of REGFILE entries for the operand with lowest use count. When the lowest use count detected is non-zero, SFR also determines the non-intermediate entry with lowest use count.

The RG=table in REG drives SFR. Entries are referenced by tags of the form RG=xxx where xxx is one of the following register classes:

BADR or ADR -	(B1 - B5)	
TEMP or SET -	(X0 - X5)	non-store regs
LOAD or LDDX-	(X1 - X5)	load regs
INTR -	(X0, X6, X7)	non-load regs
STOR -	(X6, X7)	store regs
APL -	(A3, A2, A1)	not used

Each entry is 3 words: a control word, a scan word, and a reset word. The control word contains a mask length and element count for the given register class. It is in RS format. The scan word contains an even distribution of the OTR's of the registers in the class for the given entry. Each OTR field has a width equal to the mask length in the control word. This width is 60/n where n is the element count. During SFR execution, the OTR fields in the scan word determine the REGFILE entries to examine. OTR's are positioned via left shifts. The scan stops when a zero use count is detected or the class is exhausted. At this point the scan word is stored, as is, back into its position at RG=xxx+1. Thus registers in a given class are assigned in left circular order. The reset word is an unshifted scan word. It is used to initialize the scan word when EIS starts processing a new IL segment.

In general the following algorithm is used by SFR callers to select a register. If a zero use count entry is available, take it. Else, take the non-intermediate of lowest use count. If no non-intermediates are available, then take the entry with lowest use count.

Example_12: (AIR)

AIR, assign intermediate register is called by EIS.IR when an intermediate result register is needed. AIR does not clear a register, but instead passes back a recommended OTR. It first calls SFR to search the intermediate registers, RG=INTR. If an intermediate register is available with zero use count, AIR exits. Next, the special delayed store mechanism is checked and if a store register is cleared by this method we exit. Finally, AIR checks the load registers for an entry with zero use count or a non-intermediate. If this search is successful, AIR passes back the OTR and exits. Otherwise, AIR flags failure and exits.

Example_13: (ASR)

ASR, assign store register, is called whenever a store register is wanted. ASR clears the register selected.

Selection begins with a call to SFR to scan the class of store registers, RG=STOR. If an available non-intermediate is found or a zero use count occurs, ASR is done. Otherwise, ASR must store the intermediate found, to temp. This done, ASR zeros out the REGFILE entry for the selected register and exits.

The following code sequence occurs in EIS.IR:

```
RJ      AIR
PL      B2,EIS.IR20
RJ      ASR
```

The initial AIR call tries to find a register that needs no store to temp (i.e., a non-intermediate or an entry with zero use count). If no such register exists, indicated by (B2) less than zero, then we call ASR to clear a store register. We clear a store register because clearing a non-store register forces the clearing of a store register so that the contents of the non-store register can be stored to temp. Thus, we select a store register directly and avoid one extra store to temp.

Hard_Registers:

From time to time a specific register is required rather than just one out of a class. For example, the skeleton instruction (LD L1,,R.X3) will force the L1 register to be X3 rather than just any one of the load registers. Such register requirements are called hard register requirements.

To set a hard register, place the desired OTR in the cell RREG. The contents of RREG are negative otherwise. When SFR discovers RREG contains a valid OTR, the normal scan is wired off and the OTR in RREG is returned.

Hard register settings are in general dangerous because they will select the target register even if it is locked. Therefore, the QCG programmer should never use the hard register mechanism without totally understanding the context of the use. Hard registers are mainly used when all but the register allocation portion of a processor or subroutine is needed.

H16

Example 14: (SLD - Subscripted array load)

SLD writes the code for loads and stores for subscripted arrays. SLD calls GNR (Get Next Register) to get an array register. SLD directs GNR to get one from RG=LOAD, the class of load registers. But, when EIS.STO calls SLD to compile an array store, RREG is first set to the OTR of a store register which contains the array element to be stored. Now when GNR is called the store register is returned because GNR calls SFR which wires off the scan of the load registers. The rest of SLD functions normally only now the final set A instruction results in a store rather than a load.

Status Routines:

Correlating the information in OUS and REGFILE is the job of the status routines. The main routines are GST, STS and RUT.

GST, get status of tag, queries the register status of an IL operand or a REGFILE entry. GST gets the status word for a given operand along with a pointer to that status word. These values are returned to the caller. In addition, GST checks the appropriate status bit. If the bit is off, GST returns with OTR of zero to indicate the operand is not in a register. Status bit on causes GST to check the corresponding reg-number field in status word. If the REGFILE entry corresponding to the reg-number is equal up to use count of the input operand then GST returns the OTR in the reg-number field to indicate the operand is in that register. If the REGFILE entry and the operand input to GST do not match then the OTR is set to zero indicating the operand is not in a register.

GST performs one task that could cause confusion outside of its historical context. It sets up the cell GSTA in the following format:

42/Table Vector Word, 18/ordinal

Table vector word is the T. entry in PUC for the table containing the status word. Ordinal is the ordinal of the status word in that table. This mechanism seems pretty useless in FTN5 where OUS is the only table containing status words. But in old FTN4 status words could reside in any of 3 tables. Guarding against this eventuality in FTN5, the mechanism remains.

When we want to place an operand in the REFGILE we call STS, set tag status. STS is passed an OTR, an operand, an associated status word and a pointer to that word, a type indicator (upper or lower half), and a flag, UUC, which is either 1 or 0. The operand is placed in REGFILE+OTR using UUC to decrement the use count. The type field in the REGFILE entry is set to reflect the operand type. STS then updates the status bit and reg-number field and exits.

One of the most important tasks of STS and its slave, AUT, adjust use total, is to compute a use count. AUT returns a value to STS in the use count field of the operand. When STS subtracts the contents of UUC from this value returned, the resulting number is the operand use count. For intermediates AUT subtracts UUC from the use total in the associate operator. Thus the use total in the associate operator is always equal to the use count in the REGFILE entry whenever we exit STS.

RUT, reset use table, is used to clear register status. Use table is another name for REGFILE. Given an OTR, RUT will zero out REGFILE+OTR and clear the status bit of the associated status word. Moveover, if RUT is asked to clear an intermediate with non-zero use count, it sees to it that the intermediate is stored to temp. RUT is the only status routine that outputs object code.

Storing to temp can be tricky. Suppose RUT is asked to clear an intermediate in X3 and both X6 and X7 contain intermediates. RUT must first clear a store register, storing its contents to temp. Next the contents of X3 are transmitted to the store register just cleared. RUT is now ready and stores the original intermediate to temp. To accomplish this feat, RUT does the equivalent of calling itself. It uses variable B-register jumps to avoid looping. The algorithm is one of the most complicated. Read it carefully.

Other status routines are either simple slaves or extensions of STS or GST. DIT is an important STS extension. It is called when an intermediate is first entered into REGFILE. This happens when the associate turple result is first defined. No intermediate operand referencing this turple could have been encountered up to this point in the processing. This is because intermediates reference back to previously defined results. QCG must set register status at the time the turple result is defined, but it has no operand to place in REGFILE and no way of knowing where such an operand might occur. So, DIT exists to construct an intermediate operand to use on the fly and then to set status through STS.

<u>P2. Format</u>:

Unlike the rest of the compiler, QCG keeps its operands in P2. rather than TP. format. Because QCG requires a 9-bit use count field for each operand, this difference will probably never be resolved. Status words, which are derived from operands, are also in P2. format. The significant P2. fields are TAG, BIAS, IL, RG and ST. The latter three fields are just reformattings of the lower 18-bits. ST is for status words. RG and IL are for operands.

Though operands and status words share many field names, and even though they are both representations of the same quantity, the contents of these words differs radically. The TAG and BIAS fields of status words are the TAG and BIAS fields used for prebinary loads and stores. For operands, these fields are OUS and IL ordinals respectively. Operands contain meaningful CLAS and attribute bits which are wiped out by status fields in status words. In short, QCG takes advantage of the extra space afforded by the status word. The operand is really a two word quantity. There is an IL part, called the operand, and a status word part. Keeping track of which is which is a major problem for the coder in REG.

Analysis of P2. fields:

TAG:

Status Word: TAG is P2.'s form of TP.ORD. This is the prebinary TAG field and the aplist TAG.

Operands: TAG is the OUS ordinal of the status word for the operand.

BIAS:

Status Word: P2. form of TP.BIAS used for aplist and prebinary bias.

Operand: BIAS is meaningless for non-intermediate operands. For intermediates P2.BIAS is the IL ordinal of the associate operator.

IL and RG (operands only):

These fields contain class and attribute bits, the use count field, and the upper/lower half type indicator.

ST (status word only)

ST contains the status bits and the reg-number fields.

REGFILE entries are in operand format. Operands that are not use counted are like combination operand/status words. The TAG and BIAS are like status word fields. The rest of the word is like a normal use counted operand except the use count field is zero. Use counted operands never have a zero use count field.

The Load/Store Routines:

CLI (compile load instruction) and SLD (subscripted array load) are the main load/store routines. Note, though their names only indicate loading, both of these routines serve double duty as store processors. Any routine or processor which claims to load or store variables, will ultimately call SLD or CLI.

CLI loads or stores simple variables or arrays with constant indexes. CLI is passed an operand to load, its status word, an OTR and an ORO in standard registers. It will output correct load/store code for non-LCM formal parameters, LCM formal parameters, LCM non-formal parameters and non-LCM non-formal parameters. The latter case is the most common and uses the least code. This is the main difficulty with CLI: 90% of the code is executed 10% of the time. So, it is easy to fix a minor end case and break the compiler.

SLD is really 3 routines. Because array loads are deferred, the first job is to save the current turple pointer and replace it with the associate turple pointer for the array load intermediate being processed. Next, SLD is an allocator. It obtains two registers: one for the array load and one for the index. The index is usually referred to as the address function. The final phase is to compile the load/store code. This does not differ greatly from CLI.

Allocation in SLD is tricky. The problem is that SLD wants two distinct registers and sometimes it fouls up and selects the same one for both uses. At present there is a loop that deals with this eventuality. An address function register is selected and locked. Next an array register is selected. If the address function has not been cleared from its register by the process of getting an array register, then we're home free. However, if a conflict occurs, we loop back and reload the address function. We're still not free of trouble, because the address function might have been loaded into the array register. If that happened, we clear an intermediate or store register and transmit the address function to that register.

This code usually works but it is very pathological. In FTN4 about one out of ten code generator PSR's are flushed out by this very loop in SLD. The most disturbing aspect of this loop is that it admits that a locked register has been clobbered and still continues processing. Now, register locking exists to protect a quantity that often cannot be recovered. For example, once the k-register number for a skeleton instruction has been placed in INS.REG, EIS never checks to see if the value in that register remains valid. It locks that register during j-field processing so that the k-register will not be disturbed. But, if SLD is capable of clobbering a locked address function register, it is equally capable of clobbering a locked k-register. If this occurred, it would only be detected by bad code.

Aplists and Procedures:

The main growth area in the FTN4 to FTN5 transition was the area of aplist-procedure processing. Here we use procedures for any subprograms, i.e., user functions, subroutines and any FORTRAN Common Library subprograms. FUN is the deck associated with the output of aplists and the related object code.

Character substring and array loads involve library calls. Character expressions require library calls. The fact that character substrings, arrays or expressions could legally appear in any aplist meant that FUN had to acquire an aplist stacking capability. This added complexity as well as bulk.

It would be untrue to say that the more complicated aplist cases were just extensions of the simpler ones. The methods of building aplists differ greatly. Setting all of the fields for a given aplist entry involves a complicated sequence of decisions for almost every aplist and operand type. IO aplist entries require two words per entry while user aplists only require one. Character aplist entries require a support table to carry byte address information not required by other data types. But, despite varied requirements, all aplist processors can be fitted into one of three categories: Builders, Entry Generators and Outputters. Builders control the gross structure. They make header words, find and route aplist operands. Entry generators receive aplist operands from builders. They analyze the operand, set flags and fields and finally use all this to output an aplist entry. One operand may be passed to many entry processors before an entry results. Outputters process completely built aplists. They add the new aplists to the appropriate tables, add terminator words and output call object code.

Great effort was expended to maintain structural similarities in the aplist algorithm for the two code generators. Common subroutines were used whenever possible. In general these efforts were successful. Most of the major processors and subroutines in FUN have direct counterparts in BRIDGE.

Tables:

The main job of the aplist process is to build tables. One table is for scratch. The rest contain the address references, offsets and ordinals the FORTRAN assembler uses to build object time aplist tables.

T.APL is the table of completed aplists for user procedures and character library routines. APL drives the building of the object time table of user/character aplists at assembly time. Entries are one word. They result in one object time entry.

T.API is the aplist index table for user/character aplists. There is one entry for each user or character aplist created by FUN. The zero-th entry is zero. After that, the nth entry in T.API corresponds to the nth user/character aplist built by FUN. Entries contain the index of the first word of the corresponding aplist within the resulting object time table of aplists.

T.IOA is the table of completed IO aplists. It drives the building of object time IO aplists. Entries are two words. They contain a length or control code word and an item word. The order of the words depends on the entry type.

T.IOI is the IO aplist index table. It is exactly analogous to T.API.

T.CAC is the character address constant table. This table contains the character length and the beginning character position for character aplist items. Each character entry contains an ordinal to a T.CAC word in its bias field.

T.CLW, the character length word table, exists to support character array IO length words. IO length words specify a number of array elements to process. But, character array elements are not a fixed length in terms of storage units. So, character length per array element must also be passed to the IO processor. All this would overload one character length word and so the support table T.CLW was created. T.CLW entries are two words. The first contains character length information. The second is a standard IO length word. Back in the IO aplist, the length word for a character array just contains the T.CLW ordinal in its bias field.

T.SAP, the scratch aplist table, is the place where user and character aplists are built. T.SAP exists because of the problem of having to nest aplists. When an aplist on T.SAP is completed it is copied to the end of T.APL.

The Aplist Environment and Stacking

FUN maintains several cells to drive the construction of aplists. Together these cells are called the aplist environment. They include:

APLEN - The number of words in the current aplist at any given time in the construction process.

APIND - The index of the index table entry for the current aplist.

APTAB - The address of the PUC table address word for and the index within the table on which the current aplist is being built.

ATF - The IO indicator and other fields and bits used in entry generation.

There are three words reserved for each of the above mentioned structures. This is to accommodate the maximum aplist stacking depth of 3.

_EVEL - This cell keeps track of stack depth.

Whenever it becomes necessary to interrupt the building of one aplist so that a nested aplist can be built, we stack the current aplist environment, reinitialize it for the new aplist, and start building. To stack we merely move the contents of APLEN, APIND, APTAB and ATF to the locations APLEN+n, APIND+n, APTAB+n and ATF+n. n is just 1+C(LEVEL). We then increment the contents of LEVEL by one. When the nested list is built and outputted we pop back to the original aplist and start building again. To pop we just decrement LEVEL and restore the environment.

Building_Aplists

There are six aplist types: User, IO, Character Move, Character Relational, Character Array and Character Substring. They have different formats and different builders.

User_Aplists_Building

For any user procedure call in the source, QCG is passed a burst of aplist turples followed by a procedure turple. For function calls only, this sequence is preceded by a FAP, first aplist, turple. This marks the beginning of a function aplist turple sequence and gives FUN a chance to create an entry for character function results. There is one aplist turple for each parameter listed in the source call. Aplist turples have the format:

OPR:	GAP	General aplist operator
1OP:	ITEM	Operand for listed item
2OP:	0	Always zero

Procedure turples have the format:

OPR:	FUN or SURR	Function or subroutine
1OP:	Routine Name	Sym. Tab. ordianal of routine
2OP:	Count	Number of aplist turples

QCG builds a user aplist with this turple sequence. Each aplist turple causes a trip to O=GAP via the skeleton CALL mechanism. At O=GAP we initialize the aplist environment if needed. Next we call IAW to issue aplist word, route to the various entry generators and to build nested aplists. IAW is passed an operand and returns with one entry added to the current aplist. But, between entry and exit, many intervening aplists could have been built and output. Upon return from IAW, O=GAP bumps the turple and returns to EIS.PNX.

This process continues until the procedure turple is detected. At this time the building phase is done and control passes to the outputting processor indicated by the procedure turple.

Building IO Aplists:

Like user aplists, IO aplist building is driven off a sequence of consecutive IO aplist turples terminated by an IOF. (IO function) turple. IO aplist turples generate two word aplist entries - one for each operand. Aplist building commences when the first IO aplist turple is encountered and stops with the IOF turple. Output processing is controlled by O=IOF.

There are several IO turples. Their formats are:

```
OPR:    IOD          IO data operator
1OP:    ITEM         operand for item
2OP:    LENGTH       number of elements

OPR:    IOC          IO control operator
1OP:    CODE         code for control turple type
2OP:    ITEM         format number etc.

OPR:    IOU          IO unit operator
1OP:    UNIT ITEM    unit specifier operand
2OP:    LENGTH       non-trivial length
```

The IO aplist skeleton CALL processors are O=IOD, O=IOC and O=IOU. The first one encountered initializes the aplist environment. It is safest to make no assumptions concerning the order of the turple types within the aplist turple sequence. The processors work as follows:

O=IOC outputs its first entry for the control item via IAW. It then places a control code from 1OP in the mode field of the control item entry. Then O=IOC outputs a zero second entry.

O=IOU enters the item first and like O=IOD sets the unit control code in the item entry mode field. But, then O=IOU outputs a length word through IAW.

O=IOD outputs the data item first through IAW and then outputs the length entry. The length can require CLW support.

Example 15:

WRITE (5, 100) A

Resulting turples are:

```
1.  OPR:  IOU       IO unit turple
    1OP:  5         Constant for unit number
    2OP:  1         Short constant 1.

2.  OPR:  IOC       Control turple
    1OP:  IC.FMT    Short constant for format
    2OP:  .100      Sym. tab ordinal for statement label for the
                    format statement
```

```
3.  OPR:  IOD        Data turple
    1OP:  A          Sym. tab ordinal for the variable A
    2OP:  1          Length = short constant 1 for simple variable A
```

Resultant aplist is: (IOC flags IO control)

Entry 1

[TAG=Symtab OrdinalofCON.,BIAS=offsetof5inCON.,MODE=IC.UNT,IOC=1]

[TAG=0,BIAS=1,IOC=1]

Entry 2

[TAG=Symtab Ordinalof.100,BIAS=0,MODE=IC.FMT,IOC=1]

[0]

Entry 3

[TAG=Symtab OrdinalofA,BIAS=0,IOC=0]

[TAG=0,BIAS=1,IOC=0]

In this example it was assumed A was not character. Note that the IOC bit is set for IO unit entries as well as IO control entries. This is because IO unit turples are just special control turples. Thus IOC, the IO control bit, is set.

Building Character Aplists:

There are basically two structures for character aplists. There are special variable aplists and concatenation aplists.

Variable aplists for substrings and arrays are built by FVS, format variable substring, and FIA, format intermediate array, respectively. These routines are probably not trivial to read but they are pretty self explanatory. Since substring and array turples are both deferred, some subleties arise in the process of resetting the turple pointer. This is especially true for substrings of arrays which require two turple pointer resets.

The aplists that result have the structure:

For Arrays:

```
+------------------+
! Array Designator !
+------------------+
! Address Function !
+------------------+
```

For Substrings of Variables

```
+------------------------------+
! Variable designator          !
+------------------------------+
! Address Function (if exists) !
+------------------------------+
! 1st substring index          !
+------------------------------+
! 2nd substring index          !
+------------------------------+
```

Concatenation Aplists

Character relationals, character moves and character expressions in the midst of IO or user aplists result in concatenation type aplists. The basic structure is

```
+--------------+
! Header       !
+--------------+
! List of      !
! concatenated !
! items        !
+--------------+
```

The Header word varies for each use. For relationals the header carries the number of items on the left and right of the relational operator. For moves, also called stores, the header is just the target of the store. The only character expressions in aplists are concatenations. They are treated just like moves. The header word, the target of the move, is then a compiler generated temp. The rest of the list is just the concatenated variables in the order they occurred in the source.

Example 16:

a) relational: (A.EQ.B//C) results in the following aplist:

```
+-------+-------+
! 1     !     2 !          left count and right count
+-------+-------+
!       A       !          entry for A
+---------------+
!       B       !          entry for B
+---------------+
!       C       !          entry for C
+---------------+
```

b) move: A = B//C gets the aplist

```
+-----------+
!     A     !
+-----------+
!     B     !
+-----------+
!     C     !
+-----------+
```

c) nested concatenation: CALL A (B//C) has aplist form:

```
+--------------+
!     ST.n     !       nth temp storage loc
+--------------+
!     B        !
+--------------+
!     C        !
+--------------+
```

Concatenation turples are deferred. They are processed when one of the following turples is detected:

1) User or IO aplist turple for a concatenation

2) HSTO, character move, turple

3) HLEX and HREL, character compare, turples

These turples result in one of the following builders being visited:

1) DAC, determine aplist complexity, for nested aplists,

2) O=HSTO for character moves,

3) O=HLEX or O=HREL for character relationals.

Building concatenation aplists is non-trivial. This is because concatenation expressions define trees in the IL. To build, BGA, build generated aplist, must be called to drive the tree walk. The actual tree spanning is done by GNO, get next operand. One GNO call returns one concatenation tree leaf. The leaves are the actual aplist variable operands. The search is in end order. BGA calls GNO repeatedly for aplist operands. It passes these leaves on to entry processors or special variables aplist processors. When the leaves are exhausted, BGA returns. When BGA has been called the appropriate number of times, the caller formats a header word and outputs the concatenation aplist.

Example 17: (relational)

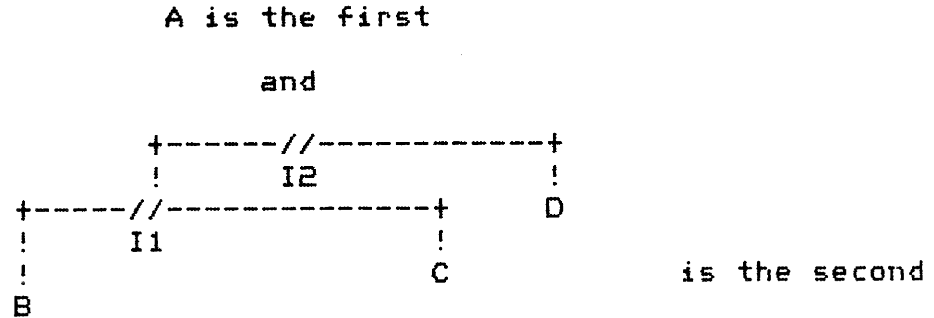
(A.EQ.B//C//D) results in the turples

1. (//,B,C)

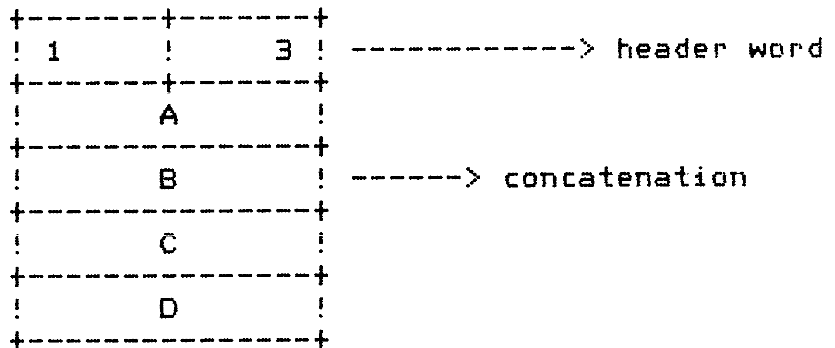
2. (//,I1,D)

3. (HREL,A,I2)

I1 points to turple 1. I2 points to 2. O=HREL will have to walk two trees:



O=HREL first reserves space for the header. Next it calls BGA for A, the trivial tree. BGA is called again for the I2 tree. Finally the header word is formatted from word count information returned by BGA and the aplist is outputed. The result is:



Entry Generation:

The process of setting up the ordinals, offsets, modes, bits, support tables and support object code for aplist entry words is called entry generation. This is done by SAP, standard aplist processor, for non-characters. PCA, pass character aplist, and ECA, enter character aplist, are the main character entry generators. All entry types are ultimately formatted and added to the current list through AAP, add word to aplist.

While most non-character entry processing is contained right in SAP, character processing is more dispersed. PCA and ECA are central but they call several support routines. Moreover, some character builder routines also engage in entry generation in simple cases. These include FVS and DAC.

Plug Code:

Address computation for certain types of variables is impossible at compile time. An array with a variable index is an example. To remedy this problem, FUN outputs address computation object code called plug code. The idea is to compute the address at object time and plug it in the aplist entry just before the procedure is called. Entry generators output plug code. Character array and substring object time library routines are really just plug code. To add a character array or substring to an aplist FUN proceeds as follows:

1. Stack the current aplist environment.

2. Build and output an array/substring aplist.

3. Output library routine call.

4. Pop the environment.

5. Add a dummy entry for the array or substring to the original aplist.

At object time, the library routine will plug the dummy entry with a meaningful entry that has correct address, character length and begining character position.

Status and CAC:

Most character entries require a CAC word to carry character length and beginning character position. Since it is not unlikely that two variables might have identical CAC words, CAC is squeezed for duplicates. This means every time a CAC entry is made, the table must be searched, slot by slot, for a duplicate. To avoid unnecessary searches, CAC status is set on a character variable once it has a CAC entry.

To do this FUN takes advantage of the fact that every character aplist variable has a register status word in OUS, but character variables never reside in registers. Setting status on a character means change the status word as follows:

1. Set the character status bit.

2. Move the old tag to a new field.

3. Replace the tag by the equivalance class base symbol table ordinal for the variable.

4. Place the ordinal of CAC word in the bias field.

When future references occur, CAC processing is skipped and the relevant information about the variable and its CAC entry is taken from the status word.

The status words for intermediates that are the roots of concatenation trees are set when a temp for the concatenation is defined. If future references to this concatenation occur, the tree walk is suppressed and the temp entry is outputted rather than an entire concatenation aplist.

GAS, get aplist status, checks operand CAC status. SSC, set status of character, sets it.

General Approach:

Almost every variable type requires some entry special processing. Non-character intermediates are stored to temp and the temp is passed as the aplist entry. Character formal parameters do not have CAC entries. Non-IO-length short constants must be expanded to CON. type constants before they can be entered in an aplist. IO-length short constants are not expanded. The only way to learn all the cases is to just sit down and read the code. But, just remember while you read, most of entry processing involves setting up an ATF, a status word and a store flag to pass to AAP. Read AAP first. That helps you know where the other entry processors are going.

AAP just copies fields and bits from the status word and ATF into the new aplist entry word. It also sets a plug bit from the store flag. Then it adds the new entry to the list and bumps APLEN.

Outputting Aplists

When FUN is finished building, the new list exists on either T.SAP or T.IOA. T.SAP aplists are copied T.APL. T.IOA lists remain on T.IOA.

The new aplist is then compared to all preceding aplists in the same table. If the new aplist is not a duplicate of a preceding list, its ordinal in the resulting object time aplist table is computed and placed in the appropriate index table entry. If the new list is a duplicate, it is squeezed off the table of completed aplists. Next, the ordinal in the object time table of aplists for the matching aplist is computed and stored into the index word for the squeezed aplist. Thus many index words can reference the same aplist.

Finally, call object code is outputted. This has the form:

```
SA1     AP.n
RJ      ROUTINE
```

or

```
SA1      IO.n
RJ       IO ROUTINE
```

n is just the index table ordinal of the associated aplist. AP refers to the non-IO object time aplist table. IO refers to the IO table. AP.n is the label for the nth non-IO aplist that was encountered in the program.

5.0 REAR END PROCESSOR

The FTN5 rear end processor consists of decks and routines to provide loader input (object code and loader directives) tables, map and cross reference listings and object listings. The rear end processor is common to both the QCG and CCG code generators.

5.1 REC: Rear End Controller

Abstract: REC provides routines to control flow of the rear end processor, and to provide functions required by that processor.

Interfaces: REC is a rear end routine, but resides on overlays (0,0), (1,0) and (2,3).

Data Structures

REC defines no structures.

Routine Descriptions:

a. REC: Rear end controller. REC controls flow through the rear end processor. CGE is called to provide a diagnostic function for CCG. REP is called to initialize the rear end and END performs storage allocation for variables not yet assigned storage. If necessary, MAP is called to produce allocation map, attribute and cross reference listings. BCT and PCA are called to convert constant tables (from CCG to rear end format. Based upon control options, the assembler is plugged to allow/disallow output of the binary and object listing and the loader 57 table. T.PB is properly initialized and if the binary tables are to be prepared, FAS is called to assemble the prebinary. Managed tables are cleaned up and exit is to the rear end loader.

b. REP: Rear end presets. REP presets working rear end cells, mostly upon basis of control card options.

c. END: Finish storage allocation. END assigns storage to quantities which have not been assigned storage by some previous process (e.g. common). All addresses assigned are relative. First, for functions, the VALUE. symbols are linked and given the value of the base member (the defined value). The amount of storage required for formats, constants, and APlist (both normal and IO) are determined and the block sizes are set by GBS. The size of the namelist block is determined next. The symbol table entries for format labels are updated to indicate the format block will be the relocation block. Local symbols are assigned block relative addresses via SMB and SSA. The length of the SUB and SUBO blocks are determined by processing T.FPI and copying FP.LEN and FP.SUBO entries to T.FPO. Storage is allocated on T.SUB and T.SUBO for the entries, T.SUB is, initialized by SBM=, T.SUBO by ISZ. The size of the

local block is calculated and the symbol table entries of local symbols are marked to be relocated relative to the local block. The tables T.GL, T.API and T.IOI are relocated, as are local equivalenced variables. The length of common and local storage required is calculated, and if too long for any SCM configuration, a diagnostic is output. Exit is to caller.

d. ADA: Assign dimension addresses. Called when a routine's dimension table entries are to be materialized. T.DIM is scanned and if DH.MAT is set on, the DH.RA is set to reflect the proper relative address (to a supplied block). The RA counter is updated by the size of the dimension information being processed, and processing continues until T.DIM is exhausted.

e. GBS: Generate block of storage. GBS is called by END to update the various local blocks on the local block table. The table entry is modified to reflect the amount of storage the particular block requires.

f. GCL: Get common lengths. GCL is called by END to determine the total size of all common blocks in a given class (LCM or SCM). The total size is returned.

g. ISZ: Initialize T.SUBO. ISZ is called by END to setup T.SUBO. The header and terminator words for each entry are initialized, the space for information being determined by T.FPO.

h. MER: Mark external relocation. MER is called by REC to mark the symbol table entry of any external symbols to be relocated external. T.SYM is searched, and all WB.EXT symbols are updated (WC.RLRB [ML.EXT]) to reflect this.

i. RAI: Relocate auxilliary tables. Called by END to perform the actual relocation of T.GL, T.API and T.IOI. Those tables in the WC. format.

j. SSA: Set symbol address. SSA is called from END to assign relative addresses to local symbols. The symbol table (T.SYM) is scanned, and if the symbol is a unique local symbol (e.g. not label, formal parameter, equivalence class member, etc), the current program relative address is assigned (in WC. RA). The size of the symbol is determined (in words, M.DBL and M.CPLX are doubled) and the program relative pointer is updated by this size. Processing continues unti T.SYM is exhausted.

5.2 RERRS: Rear End Diagnostic Texts

Abstract: RERRS contains the texts of all diagnostics output by the rear end processor.

Interfaces: The interfaces for RERRS are the same as for FERRS, except that the calling routines are located in the rear end processor. RERRS is a rear end deck and resides on overlay (2,3).

Data Structures:

The data structures defined by RERRS are the same as those described by FERRS (see 3.2) except:

Texts: The RERRS diagnostic texts consist entirely of those defined by comdeck COMFERR.

5.3 FAS: Fortran Internal Assembler

Abstract: FAS contains routines to process the prebinary file input (from either code generator) and other relevant table information into binary tables of loader information.

Interfaces: When object listing is required, FAS interfaces closely with LIST. FAS is a rear end deck, but resides on overlays (0,0) and (1,0) as well as (2,3).

Data Structures

FAS defines several data structures for use in building the binary (loader tables) output:

a. BT=: EQU symbols representing loader table codes.

b. BT.: Binary table formats. The format used depends on the type of table.

Normal Loader Table

```
     +---------+---------+-+-----+------+-+-------------+
     !         !         !P!     !      !T!             !
BT.  !   CN    !   WC    !M!/////!  RL  !Y!/////////////!
     !         !         !D!     !      !P!             !
     +---------+---------+-+-----+------+-+-------------+
         12        12     1   8     9    1       17
```

CN: Code
WC: Word count
PMD: PMD flag
RL: Relocation indication (block)
TYP: Block type

XTEXT Table

```
     +---------+---------+-+-+------+--------------------+
     !         !         !P!/!      !                    !
BT.  !   CN    !   WC    !M!/! RLX  !////////////////////!
     !         !         !D!/!      !                    !
     +---------+---------+-+-+------+--------------------+
         12        12     1 2   9             24
```

CN: Code
WC: Word count
PMD: PMD flag
RLX: Relocation index

Partial Word Text Table

```
    +--------------+----+-+-+-------+-------------------+
    !              !    !R!/!       !                   !
BT. !     LEN      !BCP !P!/!  RB   !        FWA        !
    !              !    ! !/!       !                   !
    +--------------+----+-+-+-------+-------------------+
          18         6   1 2   9             24
```

LEN: Length
BCP: Beginning character position
RP: Replication indicator
RB: Relocation base
FWA: First word address

Replication Table

```
    +-------------+---------+------+-------------------+
    !             !         !      !                   !
BT. !      C      !    B    !  RS  !        AS         !
    !             !         !      !                   !
    +-------------+---------+------+-------------------+
          15          12       9            24
```

C: Number of times block is copied
B: Block size
RS: Relocation base
AS: Relative address

Symbol Table Header

```
    +---------+---------+-+-------+-+-+-+-+------------+
    !         !         !/!       !L!D!T!/!            !
BT. !   CN    !   WC    !/!  LO   !T!S!Y!/!    SA1     !
    !         !         !/!       !B!T! !/!            !
    +---------+---------+-+-------+-+-+-+-+------------+
        12        12     2   10    1 1 2 2      18
```

CN: Code
WC: Word count
LO: Language ordinal
LTB: Last symbol table flag
DST: Dimension descriptor flag
TY: Program type
SA1: RA to store register A1.

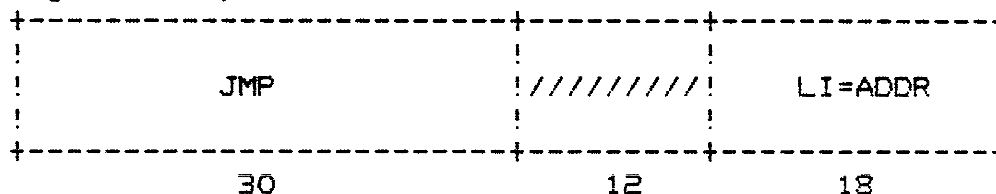
c. <u>Templates</u>: Area for templates and build area for the various loader tables FAS must construct. Templates include: Prefix (77) table, loadset preset, loadset map, loadset library, error table. Build areas include replication tables, partial word text table, text table.

d. MODEV: A small table used to convert the mode values (M.xxx) used by FTN5 to the forms used by CID/PMD, COBOL and FCL.

e. IOCAD: A branch table used by KIO to select the proper processing routine for I/O control items. Provided by comdeck COMSIOC and a rewrite of the ICDEF macro.

f. OCPSUD:
IPSUD:
FAPSUD: A branch table used by FAS and RAD to select the proper processing routine for pseudo instructions. Provided by comdeck COMSPSU and a rewrite of the PSUD and IPSUD macros.

FAPSUD is the entry used by LIST (B-5.5) to process object listing of the pseudo instructions. Format is:



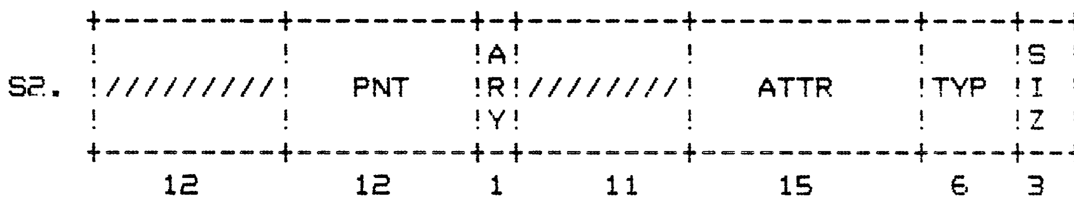
JMP: Jump instruction to FO= or FI= routine
LI=ADDR: Address of LI= routine in LIST (this field is a 'soft' external)

g. Cells: FAS defines various cells to be used as flag, scratch storage and counters.

h. S2.,S3.: The reformat of the symbol table for use by CID/PMD.

First Word Normal WA. format.

Second Word



PNT: Dimension offset (or null)
ARY: Array flag
ATTR: Attributes (see below)
TYP: Mode
SIZ: Length of symbol table entry

J6

```
S2.ATTR
SUB:     Subroutine
NLST:    Namelist group name
ENT:     Entry point
PARM:    Symbolic constant
1REF:    Stray name
MAT:     Materialized
LAB:     Statement label
DEF:     Defined
EQV:     Member of equivalence class
FUN:     Function
EXT:     External
CGS:     Compiler generated symbol
FP:      Formal parameter
LEV:     Level 0 and LCM
LCM:     LCM/ECS variable
```

Third Word

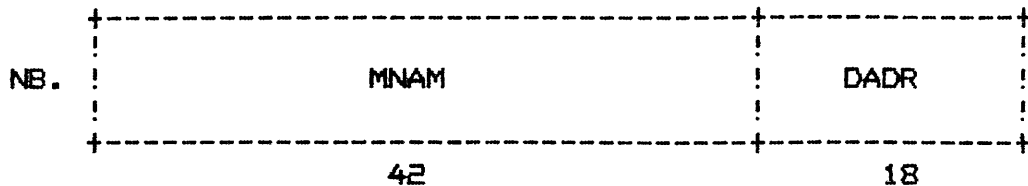
```
     +---+-----+-+-------------+-+---+-------------------+
     !/  !     !C!             !/!   !                   !
S3.  ! / ! RB  !L!    CLEN     !/!BCP!        RA         !
     !  /!     ! !             !/!   !                   !
     +---+-----+-+-------------+-+---+-------------------+
       3   9    1       17      1  4          24

     RB:      Loader block number
     CL:      Implied character length flag
     CLEN:    Character length
     BCP:     Beginning character position
     RA:      Relative address
```

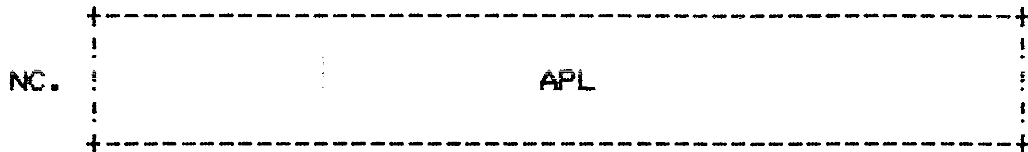
i. NA.: Namelist definition beinary output formats. KNG
NB. transforms T.NLST into the following format:
NC.

```
     +-----------------------------------+--------------+
     !                                   !              !
NA.  !               GNAM                !     NMEM     !
     !                                   !              !
     +-----------------------------------+--------------+
                      42                        18

     GNAM:    Namelist group name
     NMEM:    Number of members
```

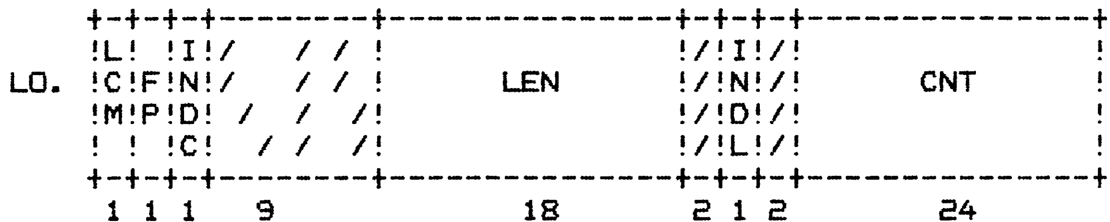


MNAM: Member name
DADR: Run time dimension address (array)



APL: Member aplist

j. <u>LO.</u>: Character length array binary format. OCL transforms T.CLW entries into the following format:



LCN: Count is LCM
FP: Count is formal parameter
INDC: Count is address
INDL: Length is address
CNT: Number of elements
LEN: Length of each element

<u>Routine Descriptions</u>:

a. <u>FAS</u>: Fortran internal assembler. Called from REC to assemble the prebinary file/table from either code generator. FAS performs some initializations on tables and files and determines if source errors were detected. If so, exit to END.ERR. Otherwise, the control cells and some templates are initialized and FAS goes into its control loop at FASRTN. RNI is called to fetch the next pseudo instruction. The value is used as an index into OCPSUD, and the proper FO= routine will be evoked. Sequences of machine instructions are begun by a pseudo instruction. A pseudo end instruction will stop the loop.

b. FO=: Processing routines for the FAS control loop. All routines (except FO=END) return to FASRTN upon completion of processing. The FO= routines are really an integral part of FAS.

FO=ADDR: Entered when the pseudo for FCL initialization routine is encountered. REL is called to relocate the APlist word. STX stores the relocated word in the text table build area and FBP provides the object listing interface.

FO=APL: Entered when an APLIST pseudo is encountered. PAT is called to preprocess the APlist table. If LCM pointers were present, T.LCA is appended to T.APL, reformatted for APlist. KAP is called to compile the APlist and POL to print object lists.

FO=BMI: Entered when the BMI pseudo is encountered. This pseudo indicates that a sequence of instructions (pseudo and/or machine) is to follow. RAD is called to process.

FO=BSS: Entered when a BSS pseudo is encountered. Calls RBS to relocate the BSS pseudo instruction.

FO=CON: Entered when the CON pseudo is encountered. Calls POL to provide object listing. Calls SMW to process the contents of T.CON.

FO=EQUN: Entered when an EQUN pseudo is encountered. Calls POL for object listing of the negative relocation macro.

FO=FMT: Entered when the FMT pseudo is encountered. Calls POL to provide object listing. Calls SMW to process the contents of T.FMT. If in QCG mode, and format labels were assigned, FLA is called to process.

FO=FVEC: Entered when a FVEC pseudo is encountered. Processes the associated file name. Creates a file pointer, outputs it via STX and calls FBP to provide object listing interface.

EQ=PLIM: Entered when the PLIM pseudo is encountered. Fetches the run time print limit and treats as a file pointer. Exit to FASRTN through FO=FVEC.

EQ=IDNT: Entered when the IDNT pseudo is encountered. DIT is called to output the identification tables and POL to provide object listing.

EQ=IOM: Entered when the IOM pseudo is encountered. Processes the I/O APlist table, when present. PAT is called to preprocess T.IOARG. KIO compiles the I/O APlist and OCL outputs character length arrays. POL is called to provide object listing support.

EQ=LCC: Entered when a LCC pseudo is encountered. The relevant loader directive is output and POL is called for object listing.

EQ=LOO: Entered when a LOO pseudo is encountered. The object list switch is reset as directed.

EQ=NLST: Entered when a NLST pseudo is encountered. KNG is called to compile the namelist group definitions. POL provides object listing support.

EQ=USE: Entered when a USE pseudo is encountered. PUSE is called to modify origin and parcel counts. DTX dumps the building text table. POL provides object listing.

EQ=TRAC: Entered when the TRAC pseudo is encountered. The program unit name is fetched (DPC). The TRACE macro is issued via STX and FBP provides object listing support. STX is called to output the TEMPAO word.

<u>FO=END</u>: Entered when the END pseudo signals the end of assembly. POL is called to list the object end card. OSB outputs the SUB and SUBO blocks. DTX dumps the final text table and DLF finishes the link and fill tables. DFD outputs the 5600 and 5700 tables. Exit is to REC through FAS.

<u>END.ERR</u>: A FAS routine which is entered when fatal errors were diagnosed. PIT is called to output the identification table and an error binary is output. Exit is to REC, via FAS.

c. <u>RAD</u>: Relocate and dump prebinary instructions. RAD is called by FO=BMI to process a sequence of instructions (pseudo, machine or CCG format). Upon entry, RAD performs some initialization and then enters its control loop at RADRTN. RNI is called to read the next instruction. Determination is made if the instruction type is QCG or CCG. If CCG, the instruction format is converted to QCG form, via CII. The instruction (in QCG format now) is decoded (partially). If a packed instruction, the current parcel is removed and decoding continues. The instruction now can be determined to be machine or pseudo. If pseudo, processing is at RAD=PSI. If the machine instruction is 15-bit, it is properly positioned in the current build word. If object listing is required, VFD is called to format the instruction, and RAD90 is the finisher. For 30-bit instructions, address decrement necessity is determined and calculated if necessary. REL is called to relocate the instruction and the build word and header word (relocation bits) are updated. VFD is called to format the instruction (if object listing is required) and the 15-bit/30-bit processing merges at RAD90, where POL is called to provide listing interface. Return is to the control loop at RADRTN.

d. <u>FI=</u>: The entry here is RAD=PSI and comes from RAD when the current instruction is pseudo. The instruction provides an index into IPSUD, which in turn provides the address of the proper routine. All FI= routines (except FI=EMI) return to RADRET.

EI=BCI: Entered when a BCI pseudo is encountered. Sets the switch which indicates CCG format instructions follow.

EI=DATA: Entered when a DATA pseudo is encountered. BNW is called to begin a new word. T.DATS is allocated enough room to process the data entries. RMI reads multiple prebinary words. POL is the object listing interface. DDS dumps the data table to binary. DTX finishes the text table.

EI=ECI: Entered when an ECI pseudo is encountered. Resets the switch which indicates CCG format instructions. Note that the BCI/ECI pairs must be bracketed by BMI/EMI pseudos.

EI=BSS: Entered to process a BSS which occurs in an instruction sequence. Calls RBS to relocate the label and ESL to enter in the 5700 table.

EI=BOS: Entered on encountering a BOS pseudo. In test mode, this provides an IDP interface. Calls CLE to construct a line number table entry. POL is the object listing interface.

EI=CPL: Entered when a CPL pseudo is encountered. BNW is called to force upper. The associated character length information is extracted from the T.SYM entry and the control word is output via STX. FBP provides the object listing interface.

EI=EMI: Entered when the EMI pseudo signals the end of the current sequence of instructions. Exits to FASRTN, via RAD and FO=BMI.

EI=JPI: Entered when a JPI pseudo is encountered. Reformats the pseudo into more machine-like format and finishes processing in FI=UJP.

EI=LOO: Entered to process an object listing directive which is embedded in an instruction sequence. Properly updates the listing control cell.

EI=OTR: Entered to process the object time reprieve directive. Reformats the OTR pseudo to SBO BO+LINENO and goes to RAD to process as a normal 30-bit instruction.

EI=RJ3: Entered when an RJ3 pseudo is encountered. Reformats the pseudo into machine-like form and finishes processing in FI=UJP.

EI=LDO: Entered when a LOO pseudo is encountered. Processes a level 0 LCM load. Only on machine configurations where possible.

EI=STO: As above, except STO pseudo.

EI=SBOI: As above, except SBOI pseudo.

EI=SUBI: Entered when a SUBI pseudo is encountered. Formats the substitution word and outputs via STX. FBP is the object listing interface.

EI=UJP: Entered when a UJP speudo is encountered. Reformats as at FI=JPI and FI=RJ3. Those routines then merge. BNW forces upper and REL provides relocation. The binary is stored in the text table by STX and RAD is entered to process object listing.

EI=RJ6: Entered when an RJ6 pseudo is encountered. The return jump with trace back is formatted and BNW and REL process. STX stores in the text table and FBS provide object listing.

EI=USE: Entered to process a USE directive imbedded in an instruction sequence. BNW forces upper and DTX restart the text table. PUSE is called to process the USE, DTX again clears the text and POL provides object listing support.

FI=ZERO: Entered when a ZERO pseudo is encountered. Puts a word of zero in the text table via STX. FBP is the object listing interface.

e. BNW: Begin new word. BNW is called when it is necessary to force upper during the building of text words. The current build word is padded with NOOP instructions, as ncecessary and the word is added to the text table via STX. ROL is called to set the object listing origin.

f. BST: Build substitution table. BST is called from REL when it is necessary for address substitution of an instruction (or APlist item). The relative address and parcel position within that word is determined and that information, together with the formal parameter number is combined to make an entry which is added to T.SUB.

g. BSZ: Build level zero substitution table. On systems where level zero parameters are permitted, BSZ will be called to make a T.SUBO entry. Analogous to BST.

h. CAB: Copy adjusted bits. CAB is called by FST to copy and transform selected WB. bits into S2. bit positions.

i. CLE: Create line table entry. CLE is called from FI=BOS to output a line number (source) table (6700) entry to T.LNT. The line number to be output is passed with the BOS psuedo.

j DDS: Dump data statements. DDS is called from FI=DATA to process the entries on T.DATS. Each data group on T.DATS is processed, in a loop. The DA. format header is processed and the information is digested. If the object of data initialization is character, the information will be output to T.PTXT and T.PTXTR for later output as a partial word text table. Non character data is output to a standard text table, via STX. Replication (DA.RP on) is handled for both by setting up loader replication operators. Upon completion (T.DATS is fully processed), DTX is called to flush any remaining entries in the text table and T.PTXT and T.PTXTR are flushed by OTB.

k. DFD: Dump 5700/5600 tables. DFD is called by FO=END to output the line number (5700) table and the run time dimension and symbol tables (5600), as required. If needed, T.LNT is dumped to binary via OTB. The dimension information (T.DIM) is collected on T.SCR and output via OTB. FST is called to format and output the run time symbol table.

l. DIT: Dump identification tables. DIT is called from FO=IDNT to output the loader 'first group' tables. PIT is called to output the IDNT (7700) table. If the program unit is not 'block data', DLC is called to output the loadset directive. The entries on T.BLKS are formatted and output as the PIDL (3400) table. If the program is not 'block data', the entries on T.ENT are formatted and output as the ENTR (3600) table.

m. DLC: Dump loadset control. DLC is called by DIT to output loader loadset directives. LDSET directives for LIB=, COMMON= (when common blocks were saved) and PRESET and MAP (when PMD is to be called). The directives are formatted and built on T.SCR and then written directly to the binary file via WLF.

n. DLF: Dump link, fill and xfill tables. DLF is called by FO=END to output the LINK (4400), FILL (4200) and XFILL (4100) tables. First the link table (T.LINK) is processed, if present. The table is sorted and entries are formatted in place. The loader table header is made, and the link table is directly output to binary, via WLF. Processing is the same for the fill table, if present. The xfill table isn't sorted, but its processing is similar. DLF may be called from ALC if severe memory shortage occurs (and FAS processing is going on). The building of T.LINK, T.FILL and T.XFIL is mostly done by REL.

o. DTX: Dump text table. DTX is called whenever it is necessary to restart a text table. The usual case is when the table is full (each text table can contain up to 15 words). Other cases would be change of USE block, data initialization, etc. If the text table build area is empty, no action takes place. Otherwise the text table header is finished by merging in count and relocation information and is written to the binary by WLF. The text table build area is reinitialized to continue building the next text table.

p. ESL: Enter statement label (in 5700 table). ESL is called by FI=BSS when a 5700 table is to be produced. The current table (if user defined) is converted from DPC to binary and that value is put into the current T.LNT entry. Note: This code is wired off by REC if a 5700 table is not to be output.

q. FBP: Format binary and print. FBP is an object listing interface routine. It is passed a binary value, which it converts to DPC, via WOD. This is in turn passed to POL to provide the object listing. Note: This code is wired off if object lists are not to be output.

r. FLA: Format labels assigned. FLA is called by FO=FMT to process T.LA (format labels appearing in an ASSIGN statement). Each label that appears in an ASSIGN statement will have a word (with the table address) output via STX. POL provides object listing interface.

s. FST: Format symbol table. FST is called by DFD to format and output the run time symbol table. the entire symbol table is scanned, with transformed entries being output to T.SCR. ADA is called to assign run time dimension table addresses. Statement labels are not output. CAB is called to transform the symbol table attribute bits from WB. to S2. format. The FTN5 internal mode is transformed to CID mode. The WC. format is converted to S3. format and the entry is added to T.SCR. Upon completion, the 5700 table is output to binary.

t. KAP: Compile APlists. KAP is called by FO=APL to reformat T.APL and to output the transformation to binary. The current build word is padded with NOOP, as necessary, and DTX is called to finish the text table. T.APL is processed in a loop, one iteration per entry. The nature of the APlist item is determined. If a symbol, the relevant symbol table entry is fetched, and analyzed. Character items result in T.CAC entries being formed. Labels are processed specially. The address field is relocated by REL and a text word is added via STX.

u. KIO: Compile I/O APlists. KIO is called by FO=IOM to reformat T.IOAPL and to output the transformation to binary. The current build word is padded with NOOP if required and the text table is flushed by DTX. Each T.IOAPL item is analyzed in a loop. If the item is a control variable, the proper processing is determined by the IOCAD table. The transformation includes converting FTN5 internal mode into FCL mode and setting the code and indicator bits for FCL. The AP list entry is relocated by REL and the resultant text word is stored by STX.

v. KNG: Compile namelist group. KNG is called by FO=NLST to process namelist group definitions. DTX is called to finish the text table. ORD outputs the run time dimension table. The loop is initialized for processing. SNR is called to set the namelist registers (actually, delimit the group on T.NLST). The group header is fetched and the number of members determined. An inner loop processes group members. The member name is fetched from T.SYM (DPC) and together with the run time dimension table, make an entry which is output by STX. An I/O APlist entry is formed by PAW and it is put in the text by STX.

w. OCL: Output character length arrays. OCL is called by FO=IOM to transform T.CLW entries into binary output (LO. format). The CLW macro word is generated (relocated by REL) and output to the text table by STX. The LO. format is saved on T.CLWB for later output. Processing continues until all T.CLW entries are processed.

x. ORD: Output run time dimension table. ORD is called by DLF and KNG to produce the run time dimension table. ORD can either output all T.DIM entries or only those which were materialized (DLF the former, KNG the later). If the dimension information entry is to be processed, the header is output, via WWB. The D1. and D2. words for each dimension are analyzed, and relocation bits set as required. The run time dimension table contains only span and lower bound, so one word suffices, and it is output via WWB.

y. OSB: Output sub blocks (for SUB, SUBO). Called by FO=END to produce the address substitution blocks required. PUSE is called to switch USE BLOCKS. DTX finishes the text table. The entries of the table (T.SUB or T.SUBO) are output to the text table via STX and PUSE is called to switch the USE block back to its original state.

z. OTB: Output table to binary. OTB is called to write the contents of a managed table to the binary file. OTB is passed the loader table header and the managed table to output. The header word is appended to the table, the table length is calculated, and the table is written to the binary file via WLF.

aa. PAW: Prepare APlist word (for namelist group member). PAW is called by KNG to manufacture an I/O APlist entry for a namelist member name. Its functions are similar to KIO, except only variables, arrays or array elements are allowed and only one APlist entry is produced per call. REL is called to provide the relocation. The entry is left in the build word.

bb. PIT: Product identification table. PIT is called by END.ERR and DIT to provide the 7700 table. The date, time, program unit name, etc are stored into the BT.IDNT template and is written directly to the binary file via WLF.

cc. POL: Print object listing. POL is the main object listing interface. If the current instruction to be printed (instruction includes both pseudo and machine) begins a word, PIA is called to convert the origin to DPC. PIK is called to produce the listing (see LIST, 5.5). Note: This code is wired off if object listing is not to be produced.

dd. PUSE: Process USE Pseudo. PUSE is called when USE blocks need to be switched. The origin and parcel counters of the previous block is saved and the current values of the new block are substituted. Comdeck COMFUSE.

ee. RBS: Relocate BSS pseudo. RBS is called by FO=BSS and FI=BSS to relocate a label. BNW is called to force a new word. If the label caused storage to be reserved, DTX is called to output the text table. POL provides the object listing interface.

ff. REL: Relocate 30-bit instruction. REL is called to relocate all 30-bit instructions (and some 60-bit fields [e.g APlist entry]). REL receives the instruction to relocate (PB. format), the parcel and origin counters and tests various flags and switches in relocating the address field. REL tests the symbol table entry of the variable (or other name) to determine the relocation necessary. The block (if the relocation is not dummy argument or external) and bias field plus the WB.RA field provide the relocation in most cases. As necessary, T.FILL, T.LINK and/or T.XFIL entries are made. Upon exit, the relocated instruction is in the building word and the proper relocation bits have been set.

gg. RNI: Read next instruction. RNI is called by FASRTN and RADRTN as the first step of the prebinary processing loop. An instruction word is read from the prebinary file and saved for analysis by the two loops.

hh. RMI: Read multiple instructions. RMI is called by FI=DATA to fetch the data initialization words which follow the DATA pseudo. A count of words to read and FWA of storage area is passed to RMI.

ii. ROL: Store origin counter in line buffer. ROL is an object listing interface (at the binary level). If the current instruction (pseudo or machine) begins a parcel, the origin (binary) is stored for conversion.

jj. SMW: Store multiple words. SMW is called by FO=CON and FO=FMT to dump the contents of T.CON and T.FMT to the binary file (with no relocation necessary). STX is called for each word on the relevant table to output to a text table.

kk. SNR: Set namelist registers. SNR is called by KNG to partion T.NLST into groups. SNR returns a pointer to the header of the current group and an indication of the number of members.

ll. STX: Store text table entry. STX is called to enter a word in the text table build area. The word and the associated relocation byte is passed. The word is added to the text table and the relocation bits are merged in. If the entry fills the text table, DTX is called to flush it to the binary file.

mm. WLF: Write LGO file. WLF is the interface routine which is called (using WLGO macro) to write information to the binary file. FWA and length of output are supplied. Note: This code is wired off if binary is suppressed.

nn. WWB: Write word to text or scratch table. WWB is called by ORD to output a word to either T.SCR or the text table build area. The word, relocation information and type of output are supplied.

5.4 MAP: Storage Map, Attributes and Cross Reference

K4

Abstract: MAP produces a variety of listings, depending upon the selection of control statement LO= options. The LO= options relevant are:

- A: Attributes
- M: Storage map
- R: Cross reference

Any combination of these attributes may be requested (or none) and the resulting listing will vary accordingly.

Interface: MAP is a rear end deck and resides on overlays (1,0) and (2,3). Note: If object listing and some form of MAP listings are not required, the space occupied by MAP becomes available for managed tables.

Data Structures

a. Segment Table:

The Map Segment Table (MST) has one entry for each unique reference map or "segment" to be output. Each entry is one word in length, and contains the following information:

1. Bits 59-30 contain the FWA of the formatter table corresponding to this segment.

2. Bits 29-0 contain the FWA of an initializing routine which will set up all conditions necessary for the controller (MOC) to output this segment.

A macro (DMSTE) was created to define map segment table entries. It creates a symbol which specifies the position of an entry within the table, and uses the VFD instruction to set up the entry itself.

b. Formatter table:

There is one formatter table for each segment, and it consists of one entry for each combination of the LO options (excluding LO=0). Each entry is one word long and contains the following:

1. Bits 59-30 are zero.

2. Bits 29-0 contains the FWA of the formatter corresponding to one combination of the LO options.

c. Formatters

K5

Each map segment could have any one of seven different formats, but as it is now implemented, a maximum of three formats are available; thus each segment has at most three different formatters. These formatters completely specify the information that will appear, and the number of columns that information will occupy. Each formatter consists of a variable number of one word entries terminated by a zero word, that specify a datum, and the number of columns that datum may occupy:

1. Bits 59-30 contain the number of columns.

2. Bits 29-0 contain the FWA of a routine which will output a datum.

A macro (DTE) was created to define a table entry, and by arranging the macro calls, virtually any segment format can be defined. For example, a format for the variable map corresponding to LO=R is currently defined as:

```
VARA    DTE    3,XB
        DTE    7,NAM
        DTE    3,XA
        DTE    11,REF
        DTE    0,0
```

Which specifies that the first 3 columns are to be occupied by a datum output by routine XB, the next 7 columns by a datum output by routine NAM, etc.

Routine Descriptions

a. MOC: MAP output controller

Initialization:

This task consists of indexing into MST to get the address of the initializing routine for the next map segment to be output and transfering control to it. This routine will set up all conditions necessary for MOC to continue, including the extraction of the formatter address, which is returned in register B5.

Heading and Title Output:

The key to understanding this task is the fact that the routine FWA's in each formatter entry actually point to display code data that is to become the heading for the corresponding datum. Thus the controller need only loop through the formatter, extract an address, and a length, and then do an A register set to point the buffer filling routine (PCB) to the right place. When a zero word is discovered in the formatter, one replication of the entire heading for the current segment has been packed into the output buffer. When the proper number of replications (determined by an initializing routine) are in the buffer, OTH (output title and heading) is called to print the title, and the newly pieced together heading.

Map Output:

The logic for this task is very similar to that of header output. MOC loops through the formatter, once for each symbol returned by RNI, extracting a routine address, and transfering control to that address plus the length of the heading (in words). The routine (field processor) gathers its data, formats it and sends it off to PCB along with the number of columns it is to occupy. Again, actual output of the line image is delayed until the proper number of trips through the formatter have been completed.

When RNI discovers that there are no more symbols to process, the buffer is flushed, and control returns to the top of MOC's loop to process the next map.

b. The Initializers:

IRA: This routine has three tasks to perform for MOC:

1. Select the appropriate formatter.

2. Gather all symbols of the proper kind into a table, and sort them.

3. Determine the format of the map segment.

One of the entry conditions is that the proper MST entry be in a X register, so completing task one requires extracting the formatter table address, and indexing into that table to retrieve the address of the proper formatter.

Task two is accomplished by selecting a pair of bit masks corresponding to the map segment to be output, and calling a routine (STS) to filter the desired kind of symbols from the symbol table. The table is then sorted by calling SST.

The third task is handled by calling DMF (determine map format) with the formatter address, the number of symbols to be output, and the MST offset available in registers.

IRB: Although this routine is called as an initializing routine by MOC, it actually controls the output of the entire common + equivalence map. This was necessary because of the fact that this map can't be output in the column oriented fashion of the other map segments.

c. Determining_a_Map_Format:

This is probably the most important and the most complicated aspect of MAP. The routines discussed in this sub-section are responsible for selecting page layouts that are attractive, easy to read, and paper conserving.

DMF: Determine map format. The parameters that determine a page layout are:

1. The number of symbols to be output in the current segment.

2. The number of columns one replication of the map segment will occupy (MOCTC).

3. The width of the page (in columns).

4. The number of lines remaining on the page.

DMF needs to calculate parameter 2 only, as the others are available on entry. A loop similar to MOC's heading output loop accomplishes this.

The basic strategy is to layout a map segment as far across the page (as opposed to down) as is possible, given the page width. For instance, if one copy of a map segment needs only 36 columns, and the page width is 72 columns, the symbols are divided in half, and two "separate" maps will be output, i.e. instead of outputting:

```
P
--Symbolic Constants--
-- NAME -- TYPE -- VALUE --
   A      real       3.14
   B      integer    10
   C      real       5.1
   D      real       7.0
```

K8

The following might be the output:

```
--Symbolic Constants--
-- NAME -- TYPE -- VALUE  --NAME -- TYPE -- VALUE
   A       real     3.14     C       real     5.1
   B     integer    10       D       real     7.0
```

There are three situations in which this division would be suppressed:

1. LO=R selected. Since a large number of references to a given symbol is not an uncommon occurrence, it was decided to dedicate any extra page width to the otuput of these references. Thus, anytime LO=R is selected, there will be only one section of a map segment printed across the page.

2. There are too few symbols to make it worthwhile. An arbitrary minimum was set on the number of rows that can appear in a map segment (MAPMRL) for a given number of sections, N. The idea is to find the largest N such that:

   $$\frac{\#_symbols}{N} \geq MAPMRL.$$

   Thus for the symbolic constant map example,

   $$\frac{4}{2} \geq MAPMRL,$$

   and since MAPMRL is currently set equal to 3, the division shown would not occur.

3. The page width is too small. Obviously, if the page width and the segment width were 72 and 40 respectively, then only one section will fit. But what if the segment width in this case was 80? The only solution is to chop off at least 8 columns; but lopping off columns in an arbitrary manner would not be acceptable. Thus, if DMF detects this situation, it calls a routine named DELF to delete fields from the segment in an intelligent manner. This routine is explained in detail in section d.

Once N has been determined, and stored into MOCTC a routine named SRNI is called to set up pointers that define exactly which symbols are to appear in each division, and control is returned to the DMF's caller, IRA.

SRNI: Set RNI parameters. This routine is designed to be called once at initialization time, and then once more each time a page is filled during the output phase.

The primary tasks of this routine are to calculate the exact number of symbols that can fit on the page, given N as calculated by DMF, and to determine which symbols are to appear on the page. The number of symbols that fit on the page is determined by the following formula:

NS = (NL-HL)*N

Where NL is the total number of lines remaining on the page, and HL is the number of lines needed to print a heading. This information is used to set pointers delimiting the symbols (in T.SCR) to be output on the current page in the following manner:

1. The index of the first to be output on this page (SRNI.FED) is set equal to the index of the last element output on the previous page (SRNI.LED) plus one.

   i.e. SRNI.FED = SRNI.LED+1

2. The index of the last element to appear on this page (SRNI.LED) is set equal to SRNI.FED calculated above, plus (NS-1).

   i.e. SRNI.LED = SRNI.FED+NS-1

The symbols thus delimited are then divided into N sections, and the indexes defining each section are stored in cells accessed by the routine RNI to determine the next symbol to process.

RNI: Return next index. This routine has the responsibility of selecting the next symbol to be processed by MOC. The basic idea is perhaps best illustrated by an example. Suppose that a total of 14 symbols are to be output for the current map segment, and further suppose that DMF had determined that 8 should go on the current page, and 6 on the next. The following diagram shows how the initial call to SRNI would have partitioned T.SCR:

```
SRNI.FED    +---------------+
            !       A       !
            +---------------+
            !       B       !
            +---------------+
            !       C       !
            +---------------+
            !       D       !
            +---------------+
            !       E       !
            +---------------+
            !       F       !
            +---------------+
            !       G       !
            +---------------+
SRNI.LED    !       H       !
            +---------------+
            !       I       !
            +---------------+
            !       J       !
            +---------------+
            !       K       !
            +---------------+
            !       L       !
            +---------------+
            !       M       !
            +---------------+
            !       N       !
            +---------------+
```

Also, suppose that the map was to appear in two sections across the page. The group of symbols delimited above would have been further partitioned as:

```
              +---------------+
              !       A       !
              +---------------+
              !       B       !
Section 1     +---------------+
              !       C       !
              +---------------+
              !       D       !
              +---------------+
              !       E       !
              +---------------+
              !       F       !
Section 2     +---------------+
              !       G       !
              +---------------+
              !       H       !
              +---------------+
```

In order to have section 2 appear to the right of section 1 on the output, it is necessary to bounce back and forth between the two sections of the table, selecting symbols in the following order: A, E, B, F, C, G, D, H.

To accomplish this, the following algorithm is used:

1. Add to the previous index returned (RNI.PI), the length of the corresponding section to get NI.

2. If NI is greater than the index of the last element in the division (SRNI.LED), subtract the difference of SRNI.LED and SRNI.FED from that number; otherwise NI is correct.

   i.e.

   RNI.PI = NI for NI≤SRNI.LED
   RNI.PI=NI-(SRNI.LED-SRNI.FED) for NI>SRNI.LED

If all the symbols for the current page have been output, SRNI is called to determine the symbols to appear on the next page (if there are any left), and to divide them into the proper number of sections. The above algorithm is then repeated if there are more symbols, or the routine is exited if there aren't.

DELE: Delete fields. This routine is called when one replication of a map segment will not fit on a page. Fields are deleted on a priority basis until the length of the segment becomes less than or equal to the page width.

One table for each map segment that could exceed the minimum page width in length was created. These tables specify fields to be deleted, and the order in which deletion is to occur. Deletion is accomplished by replacing addresses of field processors in a formatter with the address of a processor which does nothing. For example, if the page width is set too small for one of the variable map formats to be output, the following table is accessed:

```
DELFPT1    VFD    60/XC
           VFD    60/ADR
           VFD    60/BLK
           VFD    60/XA
           VFD    60/SZE
           VFD    60/XA
           VFD    60/0
```

This table specifies that an XC processor should be the first one deleted, an ADR processor second, etc.

4. Field Processors

Routines that do the retrieval, formatting and output of the data items are called field processors. Each one consists of a 4 word display code heading, followed by code to carry out its function.

Data Retrieval

Each processor is passed an entry from T.SCR (specifying a symbol to process), and the number of columns it is allowed to use to output it's data. In general, data retrieval consists of a symbol table access, but some data require access to other tables, and/or special processing.

Data Formatting

Each processor is coded to format it's data in a specific way. The only formatting common to every processor is the justification of it's output within the columns allotted to it. This function is performed either by explicit hard coded logic, or by calling a routine (JIF) which does it for them. In general, an output that comes in several pieces is done with hard logic, while one piece items are handled by VIF.

e. Common + Equivalence Map

Overview

This sub-chapter describes the output and some of the reasons for choosing the output format, along with a brief discussion of the main loop controlling the output.

The Output

The output of this map describes, in an almost pictorial way, how the programmer made use of common blocks, how he assigned storage within the blocks, and how he defined equivalence relations among variables. For each common block, a line describing the blocks attributes is followed by a pictorial description of the memory layout within it. After all common blocks have been so described, a pictorial description of all local equivalence relations is output.

For example, a program might contain the following:

```
COMMON  A  X,Y,Z(5)
EQUIVALENCE (W,Z(3))
```

The common + equivalence map would then contain:

```
A  LEVEL=1, SIZE=7 words SCM
   X<1> Y<2> (Z<3-7>W<5>)
```

In general, the line describing block attributes contains:

1. The block name, bounded by slashes (// for blank common).

2. The block's level, as defined by a LEVEL statement.

3. The block's length and type of storage units (CHAR or WORDs).

4. The memory type (SCM, CM, ECS, LCM, etc.).

5. The word SAVE if the block name appeared in a SAVE statement.

The description of the layout of memory consists, in general, of a list of the variables contained in the block, with equivalence classes bounded by parens. Each variable name is accompanied by indices of the first and last storage units it occupies within the block, separated by a dash and bounded by brackets.

A program containing local equivalence classes, such as:

```
DIMENSION X(10), Y(5), Z(5)
EQUIVALENCE (X(1), Y(1)), (X(6), Z(1))
```

would have the following common + equivalence map:

```
--LOCAL EQUIVALENCE--
    (X<1-10> Y<1-5> Z<6-10>)
```

The indices in this case are relative to the beginning of the class.

The Main Loop (IRB)

This routine is called by *MOC* as an "initializing" routine, but it will actually control the output of the entire common equivalence map, and then force *MOC* to bypass it's own output control logic by telling it "nothing to output".

IRB is a straight-forward implementation of the highest-level of logic for this map.

Initialization and Preparation

This sub-chapter will describe the routines that initialize and massage the input to the routines that do the actual output of the various parts of the common + equivalence map.

GNB: Get next block. This routine has three functions:

1. Determine the next block to process.

   All that needs to be done here is to increment by one an index (OBI.BI) into the block table (T.BIAS), and determine when the last block has been processed by comparing the index to the length of the table.

2. Detect and skip the SAV$E block.

   When a SAVE statement is encountered in a source program, the compiler generates this user-transparent block. When this block is encountered by GNB, in order to preserve it's transparency, it must not appear in the output, and therefore must be ignored.

3. Get block members.

   As in other map segments, block members are filtered out of the symbol table via *STS*, and then sorted. the main difference is that *STS* is made aware that GNB is calling via the flag STS.BI (containing the block index) and the *MST* offset. Instead of putting the name of a block member in each table entry, it's relative address is calculated and entered. Thus after calling the sort routine, the table is ordered by increasing relative address, rather than increasing alphabetic order.

   Another difference is that each equivalence class must somehow be delimited within the table. Taking advantage of the fact that after sorting, members of a given equivalence class must be in consecutive table entry's, *MEC* is called to mark equivalence classes by placing a count of the number of class members in the table entry of the first member of a class encountered.

MEC: Mark equivalence classes. As mentioned in the previous paragraph, this routine places a count of the number of members in an equivalence class into a field in the table entry of the first class member encountered. As an optimization, groups of consecutive non-equivalenced variables are also marked, placing the complement of the count into the table entry of the first member of such a group.

The algorithm that implements this is the following:

1. If the next table element is equivalenced, go to 4.

2. Count the number of consecutive non-equivalenced table members.

3. Put the count into the table entry fetched in 1., go to 1.

4. Locate this element in T.ECT (which contains all equivalence classes).

5. Count the number of variables in this class.

6. Put the count into the table entry fetched in 1., go to 1.

This algorithm is halted when the last element in T.SCR has been accounted for.

In addition, as a class is located in T.ECT, the base member of that class is marked "processed" by setting bit 59 of it's T.ECT table entry. This is to enable *GLE* to easily find each local equivalence class (bit 59 of the base member will not be set).

GLE: Get local equivalence. This routine scans T.ECT for a base member that doesn't have bit 59 set. Such a base member marks the start of a local equivalence class. The next step if to create a T.SCR entry for each member by transfering the TE.SYMI field to the MT.WAI field, and the TE.BIAS field to the MT.RA field. Bit 59 of the base member is set to prevent duplicate output, and after each member of a class has been processed, the member count is placed into the T.SCR entry of the first member of that class encountered.

f. Output and Control

This sub-chapter will discuss the routines that output a memory layout.

OML: Output memory layout. The main responsibility of this routine is to detect the start of an equivalence class, and to delimit it by outputting surrounding parens. The algorithm involved is as follows:

1. Extract the MT.NMEC field of the next T.SCR entry.

2. If the quantity is negative, go to 4.

3. An equivalence class has been encountered. Output a left paren, call *OCEI* to output the class, then output a right paren. Go to 1.

4. A group of non-equivalenced variables has been encountered. Call *OCEI* to output them, go to 1.

The algorithm is halted when the last entry of T.SCR has been processed.

OCEI: Output common + equivalence items. This routine is passed the address of the first item to process, and the number of items to process. For each item, the routines *NAME*, *FIRST*, and *LAST* are called to determine (and save) the display code and length in characters of the name of the item, the index of the first storage unit it occupies, and the index of the last storage unti it occupies, respectively. This information is then output one piece at a time by successive calls to the buffer packing routine *PCB*.

5.5 LIST: Object Code Listing Routines

Abstract: LIST provides a listing of the object program in the form of a pseudo COMPASS assembly listing.

Interfaces: LIST is a rear end routine and resides on overlays (1,0) and (2,3). LIST interfaces with and is driven by routines of FAS (Fortran Assembler, 5.3). Note: If object listing is not required the space occupied by LIST becomes available for manager tables.

Data Structures:

List defines the following data structures:

a. **PS.:** Shifted program tag. Used by WAP for APlist listing.

```
     +-------------+-------------+-----------------+
 PB. !      RA     !     ORD     !/////////////////!
     +-------------+-------------+-----------------+
           18            18               24
```

RA: Relative address
ORD: Ordinal (symbol table)

b. **Cells:** LIST defines scratch storage and templates for production of object listing of machine instructions. Individual routines describe templates for production of the various types of object listing required for the pseudo instructions.

c. **LBT:** Local block table. Defines the ordinals of the program unit's local blocks. Provided by comdeck COMSLBT.

d. **E.PIK:** Machine instruction formats. This table is in PUC (section B-2.3) but is used by LIST to format the machine instructions for object listing.

Routine Descriptions:

a. **PIK:** Print instruction conversion. PIK is the LIST controller. It is called from POL (FAS, B-5.3) to process the instruction just assembled. Inputs are the instruction (PB. format) and an indication if the instruction is machine 30-bit, machine 15-bit or pseudo. PIK determines the nature of the instruction, and if pseudo, uses FAPSUD (FAS, B-5.3) to determine the address of the proper LI=processor and jumps to that processor. For machine

instructions the i,j and k fields are extracted, prefixed with the proper register prefix and saved. Short instructions are basically done, except for formatting. Long instructions must have KTX called to convert the k field to readable form and KTY to convert the offset. The processes merge to fetch the instruction format (from F.PIK) and format the register fields properly. The address field, when present, is moved into position and LINEBUF (the object line building area) is filled with the formatted instruction. WOF performs the actual output. If necessary, WSM is called to output an address substitution macro.

b. LI=: Print pseudo instructions. The LI= group is really a part of PIK. The individual routines perform their function and exit to POL through PIK. A brief description of the routines follows:

LI=BMI:
LI=BCI
LI=OTR
LI=EMI
LI=ECI
LI=LOO
These entries do nothing. They are present so that FAS/POL can mechanically call PIK without testing the pseudo for listing necessity. They merely return to POL immediately.

LI=ADDR: Entered from PIK to list the ADDR pseudo. KTX and KTY are called to convert the tag and bias fields and PVF prints the line.

LI=APL: Entered from PIK to list the APlists of the program unit (not I/O APlists). Sets a switch to process standard APlists. Calls WAP to list the APlists and WLP to list LCM pointer cells.

LI=BOS: Entered from PIK to list the source line number. The line number is converted to DPC via CDD and the line number is output in the form of a comment:

'* LINE xxx"

LI=BSS: Entered from PIK to list a labeled BSS. Calls PBS to perform this function.

LI=CON: Entered from PIK to list all constants of the program unit (on T.CON). Entries are set up for WCC to produce T.CON listing and WCC does the work.

LI=CPL: Entered from PIK to provide object listing of the CPL pseudo. The character variable tag is converted via KTX and the length is converted via KTY. The object listing line will have the form of a macro and is output by PVF.

LI=DATA: Entered from PIK to process the DATA pseudo. Normally, this results in no action (as LI=BMI) but in text mode, T.DATS is dumped by DMT=.

LI=END: Entered from PIK to list the END pseudo. A single COMPASS END line is produced, and listed by PVF.

LI=EQUN: Entered from PIK to list the negative relocation macro. SFN is called to blank pad the LENP. name and WOF produces the object listing line.

LI=FMT: Entered from PIK to list all formats of the program unit (T.FMT). WCF is called to perform the object listing of the formats. For OPT=0, if format labels were assigned, PBS is called to provide that listing.

LI=FLA: Entered from PIK to process the actual labels of assigned formats. The header had been prepared by LI=FMT. The labels on T.LA are converted by KTX and output by PLL.

LI=FVEC: Entered from PIK to print a file vector macro. The macro field is formatted and the symbol table WA. field is modified to allow printing of the file name. KTY converts the buffer length and SFN blank pads the file name. PVF outputs the line.

LI=PLIM: Entered from PIK to output the print limit pseudo. The line is formatted and then LI=FVEC logic completes the object line output.

LI=IDNT: Entered from PIK to produce object lines corresponding to the IDNT pseudo. WCS lists the block statistics. An IDENT line is then output, via PVF.

LI=IOM: Entered from PIK to produce object listing for the I/O APlists. Initialization and switches are set for WAP to produce I/O APlist (as opposed to normal APlist) and WAP is called to produce the lines. WCL is called to provide listing of character length arrays.

LI=JPI: Entered from PIK to produce object listing of a JP instruction. Since the JP instructions in general are treated by the code generators as pseudos and are reformatted into instructions by FAS (B-5.3), LI= entries are provided. The instructions is formatted partially and final processing is at LI=RJ6.

LI=LCC: Loader directive production. The decision was made not to attempt object listing of loader directives. This routine merely exits to POL, via PIK.

LI=NLST: Entered from PIK on the NLST pseudo. Operates like LI=DATA, calling DMT= to dump T.NLST (in test mode only).

LI=RJ3: Another jump formatter, this for RJ without traceback. Exit to LI=RJ6.

LI=RJ6: Another jump formatter, this for RJ with traceback. Contains the formatting for the jump instructions. Exits to PIK to treat as a 30-bit instruction.

LI=SBOI: Entered from PIK to process the SUBOI pseudo. sets the macro name and exits to LI=SUBI.

LI=SUBI: Entered from PIK to process the SUBI pseudo. Sets the macro name. SBOI merges here. KTX is called to convert the tag and the macro line is output via PVF.

LI=UJP: Another jump formatter, this for EQ. exit to LI=RJ6.

LI=USE: Entered from PIK to list the USE pseudo. The pseudo is formatted and output by PVF.

LI=TRAC: Entered from PIK to list the TRAC pseudo. The TRACE. macro is formatted and the object line is output by PLL.

LI=ZERO: Entered from PIK to process the ZERO pseudo. ZWI is called to output a zero line.

c. KTX: Convert tag to external format. KTX is called to convert a PB. format tag into external (i.e., readable) format. The nature of the tag is determined and if not a symbol table entry, a prefix (based on the tag pointer) is fetched and the number is converted to DPC (octal). The conversion returned will be of the form PP.xxx, where pp is the prefix and xxx is the converted number. If the tag is a symbol table tag, the proper T.SYM entry is fetched. If not label or external, the WA. field is returned. Labels are preceded by period and externals by '=X'.

d. KTY: Convert constant field. KTY is called to convert a numeric field into octal DPC.

e. KUB: Convert upper bits. KUB is called by WCA and WIO to convert the high order 12 bits of a (APlist) word to DPC.

f. PBS: Print BSS. Called when a BSS instruction is to be output. The amount of storage is converted by KTY. KTX is called to get the label field converted. The name is blank padded by SFN and PLL outputs the line.

g. PVF: Pack variable field (for listing). PVF is called to output an object line which requires formatting of the variable field. The variable field position is tested and as required, blanks are added. The line is printed via WOF.

h. PLL: Publish listing line. PLL is called to output an object line which may need formatting. The original fields are converted to octal DPC, if necessary, via WOD. If the code field required relocation, a '+' is inserted. PVS is called to pack the location, instruction and operand fields. The origin is converted to DPC as necessary. The object line is output, via WOF.

i. PVS: Pack variable strings. PVS operates like PVF, except that a range of words to be blank filled is passed. They are blank padded, as required.

j. VFD: Variable field definition. VFD is called to format and position the instruction field of an object listing code line. The binary of the instruction is converted to octal DPC and stored in the proper parcel of the instruction field.

k. WAP: Write APlists compiled. WAP is called by LI=APL and LI=IOM to output object listing for APlists. The APlist to list is on T.PTXTR. If no entries are present, exit is immediate. Otherwise, the address of the actual listing routine (WCA or WIO) is plugged to the RJ code to provide common interface, with two formatters. The APlists are copied to T.SCR and sorted in ascending order by SST. Each APlist is considered a group. The header BSS is formed and output via PBS. Then each individual AP entry is processed by the proper listing routine, as plugged by the entry conditions. Processing continues for all APlists.

l. WCA: Write coded APlist. WCA is called by WAP to format and list a single standard (non I/O) APlist entry. The AP entry is analyzed. First, KTX is called to convert the tag to output form. If the AP entry is type character, the BCP and CLEN fields are converted by KTY. The bias field is converted by KTY. The results are stored in a template and KUB is called to convert the upper bits. PVF finishes formatting and outputs the line. WSM writes the SUB macro, as necessary.

m. WCC: Write constants. WCC is called from LI=CON to output the object listing for the constants on T.CON. For each constant on T.CON, WCC formats and converts the address field, converts the binary constant to octal and alpha DPC and formats both forms for output one line for each constant in output, via WOF.

n. WCF: Write formats. WCF is called by LI=FMT to list the formats on T.FMT. The binary on T.FMT is converted to alpha DPC and that is formatted and then output by WOF.

o. WCL: Write character length array. WCL is called by LI=IOM to output any necessary character length arrays. PBS is called to print a 'CL. BSS' line. If character lengths are present, the CLW macros are output using the binary on T.CLWB). The binary for each is converted to octal DPC and WIO is called to output the item.

p. WCS: Write block statistics. WCS is called by LI=IDNT to list the block statistics for local blocks. The listing subtitle is set to object listing and WOF is called to output a new title line. The block origin table, F.LBT and TLBN are used to form the output lines, which are printed by WOF.

q. WIO: Write I/O APlist. WIO is called by WAP to format and output I/O APlists. The results of all field conversions are stored in a template for listing. KTX is called to convert the tag. If the entry is a control item, the control code is saved for format. The various fields are formatted, using KTY and comma separators. PVF provides the listing of the object line.

r. WLP: Write LCM pointers. WLP is called by LI=APL to process any T.LA entries which were copied to T.APL. The tags are converted, via KTX and the offset by KTY. The line is output by PVF and WSM is called to output a SUB macro.

s. WSM: Write SUB macro. If no address substitution is present, exit is immediate. Otherwise, the proper T.FPI entry is converted, via VFD and the resultant line is output by PVF.

t. ZWI: Zero word item. Called when necessary for an object line which represents a binary zero. ZWI formats the DPC of zero and prefixes with '-' as necessary. Output is by WOF.

6.0 CCG_ROUTINES

L9

The routines and decks described in this section comprise the Common Code Generator and the FTN5 interfaces to that product.

6.1 CCGC: CCG Controller and Support

L10

Abstract: CCGC is the controller for the CCG process. It consists of routines to format information from the front end and to reformat for rear end processing.

Interfaces: CCGC is a CCG interface deck and resides on overlay (2,2).

Data Structures:

a. Cells: CCGG provides interface cells with standard names for use by CCG. Flags and messages are passed to and from CCG here.

b. Tables: Managed table cells (pointers and sizes) for tables used by CCG and tables which must be preserved by CCG. Provided by comdecks COMSTAB, COMSTAD and COMSTAS.

c. OCPSUD: IPSUD Tables of WO= and WI= pseudo instruction processing routines. Provided by comdeck COMSPSU and rewrite of PSUD and IPSUD macros. Inbedded in comdeck COMFWIN.

d. AI.: APlist index table

```
+-+-+-+-------------+-------------+-------------+
!E!B!/!             !             !             !
!Q!A!/!    LINK     !     LEN     !    INDX     !
!V!S!/!             !             !             !
+-+-+-+-------------+-------------+-------------+
 1 1 4        18            18            18
```

EQV : Equivalenced APlist
BAS : Base member of class of equiv APlists
LINK: Link pointer
LEN : Length
INDX: Index

Routine Descriptions:

a. CCGC: CCG controller. Entered from FTN22 (INIT22, B-2.9.5), CCGC controls the flow of CCG compilation. BRIDGE is called to transform the front end information (IL, tables, etc) to CCG compatible format and to interface CCG. If OPT>2, CG$GPO is called to perform global optimization. OTC is called to output terminal code. Exit is to the code generator loader.

b. DPT: Define program tag. Called by DLT to define area for generated labels.

c. FA=LOL: List one line. FA=LOL provides CCG an interface for output of messages. Calls WOF to provide the write function.

d. HE$ABT: These are standard named error exit routines.
HE$CTX Since CCG [and overlay (2,2)] have no diagnostic
HE$EPX capability, a flag indicating a diagnostic is saved and the actual message will be output by REC (B-5.1).

e. HR$LDC: List dead code. A standard named CCG interface routine to list dead code line numbers and routines. The line numbers are converted to DPC by CDD and the messages output by WOF.

f. WIN: Write instruction (to prebinary file). Called using the WCODE macro. WIN takes an instruction and formats it in PB. format. If the instruction is pseudo, PSI is transferred to for processing. If the instruction is 15-bit, attempt is made to add it to a packed prebinary word. Some short instructions require transformation of the j or k fields. As required, this is done. If the instruction is 30-bit, the current format is PB., and the instruction is merely output to F.PB (first outputting any partial 15-bit package which had been collected. Exit is to caller by a B-reg jump. Comdeck COMFWIN.

g. PSI: Process pseudo instruction. The pseudo instruction is used as an index into PSTAB and the proper WO= or WI= processor is called. Brief descriptions follow:

WI=BOS: Pseudo in proper form, output as instruction.

WI=INDI: Initialize origin, current block index and parcel counter. Output pseudo as instruction.

WI=LDO: Increments parcel count. Outputs as long
WI=STO instruction, but counts as short.

WO=ADDR: Reserves space for the pseudo word and call
WI=CPL DLT to advance the origin counter. Outputs
WO=PLIM as long instruction.
WI=SUBI
WI=SBOI
WI=ZERO
WO=FVEL

WO=TRAC: Fetches the T.SYM ordinal for TRACE. and calls DLT to define a label tag. The same for TEMPAO. Treats as long instruction.

WO=CON: Outputs the pseudo as long instruction.

WI=USE: Calls PUSE to process the pseudo. Outputs
WO=USE as long instruction.

WI=OTR: If necessary, a parcel is cleared. The pseudo is output as long instruction, and the origin and parcel counters are updated.

WI=RJG: These machine instructions are treated as
WI=UJP pseudos. The origin counter is set to
WI=JPI force upper after the instruction. Treated
WI=RJ3 as long instruction.

WI=BSS: Calls BNN to begin new word. Formats the
WO=BSS tag and calls DLT to define the label.
WO=BSSZ Treats as long instruction.

WO=END: Calls PUSE to restore USE block. Treats as long instruction.

WI=LOO: These pseudos are merely output as long
WO=LOO instruction, with no special processing.
WI=EMI
WI=BCI
WI=ECI
WO=NLST
WO=APL
WO=IOM
WO=FMT
WO=EQUN
WO=LCC

WI=DATA: Calls BNN to force a new word and then processes as long instruction.

Note: All the routines of PSI return to WIN for final processing. PSI is on comdeck COMFWIN.

h. DLT: Define label tag. Called to force upper (increment the origin counter) and to make a PB. format tag (if a tag is to be produced). The force upper is always done. If a tag is necessary, its type is determined. If symbol table, the WC. field is updated with current RA, RL and RB information. A generated label is processed by DPT. All other tag types are invalid here. Comdeck COMFWIN.

i. IIS: Issue temporary storage. Called by OTC to output the TEM. USE block to prebinary. The counts of various temporary storage classes have been updated and a USE pseudo is output, then BSS pseudos for each class is output via WIN. A USE BUF. pseudo is then output. Comdeck COMFITS.

j. PUSE: Process USE pseudo. Comdeck COMFUSE. Described in FAS (B-5.3).

k. OTC: Output terminal code. Called by CCGG upon return from CCG to output hanging code. Code is output to prebinary, via WIN, using the ISSUE macro, which provides formatting and the WCODE macro. CG$CUB is called to compile any variable dimension code. An ECI pseudo is issued. If address substitution is required, USE and BSS pseudos are issued and OSI is called to output the index table. A USE for the START. is issued and OSC is called to output SUB code for the header. MZP is called to mark possible level 0 vardims. A USE pseudo for TEM is issued as is the SUBO BSS pseudo. OSI is called to issue the SUBO index table. A USE pseudo for START. is issued and OZC is called to issue SUBO code for the header. OVC output vardim code and OLC outputs FP local copies. Alternate entry code is issued as required (a mini version of what has been done for the main entry). An EMI pseudo is issued and CAW is called to convert first L$APT and then L$IOI to WC. format. ITS is called to issue the temporary storage pseudos and MEP completes the processing. An END pseudo is issued.

l. OLC: Output local copies of formal parameters. OLC is called by OTC output any required local copies of formal parameter for each entry of the program unit. The entries are in FP. format. RLIST type entries are formed by SRI and BCI and ECI pseudos are issued as needed.

m. SRI: Store RLIST instruction. SRI is called by OLC to issue RLIST instruction to L$TXT.

n. OVC: Output vardim code. OVC is called by OTC to issue any vardim code. If code is present, it is moved to L$TXT via MVT. MAV marks vardims which are applicable to the current entry and MMV marks materialized vardims. A BCI pseudo is issued. CG$CPC is called to compile the code and an ECI pseudo is issued.

o. MVT: Move vardim to L$TXT. MVT is called by OVC. Space is allocated on L$TXT and the entries on L$VDT are moved there for processing.

p. MMV: Mark materialized vardims. MMV is called by OVC to mark those necessary. The copy on L$TXT is looped through and stores are processed. Needed stores are given the real CA from L$VDT, otherwise, the store is NO OPed.

q. ISA: Issue save AO. (or RJ CPL.). ISA is called to issue the save AO or RJ CPL. pseudos. If interactive debug was selected, code to save A1 is issued. If the program unit is type character function, code to save AO is issued. If unique entry parameter lists are present, code is issued to save registers and a CPL. RJ is issued. Comdeck COMFISA.

r. MAV: Mark vardim appropriate (to this entry). The L$VDT copy on L$TXT is scanned, using L$VDI. If no FP's for the current entry, no action is taken. Otherwise, the parameters are tested against VDI, and if a match on upper, lower bound or span, the VDI entry is marked by MVD. If the program unit is passed length character, that VDI entry is marked by MVD.

s. MVD: Mark vardim. MVD is called by MAV to set the proper bits in VDI.

t. OSI: Output sub index table. OSI is called by OTC to output address substitution index tables (SUB and SUBO) as needed. The SUB index words are formatted and output to TST.

u. SLE: Squeeze last entry. SLE is called by OSI after issuing a sub index table entry to attempt elimination of duplicate entries.

v. OSC: Output sub code. OSC is called by OTC to issue any required address substitution code at entry points. Code is formatted and issued to set up entry condition for and a RJ to SP5. Comdeck COMFOSC.

w. OZC: Output SUBO code. OZC acts as OSC, except level 0 substitution code is issue, as necessary. On comdeck COMFOSC.

x. CAW: Convert APlist index table to WC. format. CAW is called by OTC to convert APT or IOI to WC.RA format. The entries are scanned (on the pertinent table) and the AI. form entries are reformatted to WA.

y. MEP: Miscellaneous end processing. Called by OTC to finish up CCG interface processing. The length of the run time constant table is determined. CG$IEP is called and MDV reformats DIM VD pointers.

z. MDV: Mark dimtab vardims as needed. The dimension table, DIM is scanned and if variable dimensions that materialize are present, CG$AVO is called to obtain a converted CA field.

6.2 CSKEL: Form Code Skeleton Tables.

Abstract: CSKEL contains code skeletons and opdefs for use by the CCG routines.

Interfaces: CSKEL is a CCG interface deck and resides on overlay (2,2).

Data Structures:

CSKEL consists of macros to define the skeletons and opdefs and the skeletons and opdefs.

a. Macros:

RMM=:	Force micro evaluation for remotes
SKEQU:	Equate skeletons
SUBSKEL:	Declare beginning of sub expansion
SUBEQU:	Declare equivalent pass 2 skeleton
ENDS:	End macro skeleton
FORM:	Form instruction skeleton element
SETCON:	Set number/address field in skeleton (con)
SETOTH:	Set number/address field in skeleton (non-con)
BRANCH:	Continue skeleton elsewhere
CALL:	Call external processor to process or partially process current turples
SKOP:	Skeleton of operator
SKPSET:	Set SKOP numeric selection parameters
M.I.:	Macros to define machine instruction skeletons.

b. Opdefs: Provided by comdeck OPRDEFS. Described in RLINK (B-2.4.4).

c. Skeletons: The skeletons are provided by comdeck SKEL. The format is:

```
     +---+---+---+----+---+----+---+----+----------+
 SK. !TYP!OPC!IAD!INUM!JAD!JNUM!KAD!KNUM!    QF    !
     +---+---+---+----+---+----+---+----+----------+
       6   9   5   6    5   6    5   6       12
```

TYP:	Types of operation
OPC:	Opcode
IAD:	i address
INUM:	i number
JAD:	j address
JNUM:	j number
KAD:	k address
KNUM:	k number
QF:	q (constant) field

6.3 CCG Routines

M1

The CCG-proper routines are maintained by the 170-CCG project and are described in that IMS. For reference, the following deck comprise CCG.

- CGTM: Code generation table manager.
- MIO: Mass storage I/O routines.
- FBV: Form bit vectors.
- GPO: Global program optimization.
- GRA: Global register assignment.
- SQZ: Redundant operation elimination.
- MCG: Machine code generation.
- BDT: Form dependency tree (graph).
- CFA: Control flow analysis.
- UDT: Usage/definition table processing.
- PROSEQ: Process accumulated sequences.
- Output: CCG (test mode) output routines.

6.4 BRIDGE: Front End IL - CCG Interface

Abstract: Bridge accepts IL from the front end processors and converts it to a form understood by CCG (RLIST). The converted turples are passed to CCG, which generates code and returns to BRIDGE.

Interfaces: BRIDGE is an interface deck for CCG and resides on overlay (2,2).

Data Structures:

a. TS.: Turple Status table

```
    +--------+-------------+------------+-------------+
TS. !   AT   !    CLEN     !    RNL     !    RNU      !
    +--------+-------------+------------+-------------+
        6          18           18           18
```

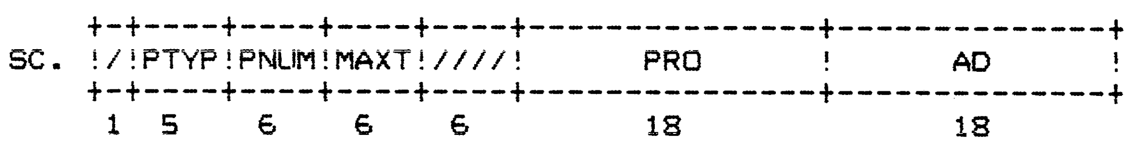
```
AT   :    Attributes
          FR :    Function result
          FRL:    Lower function result
          DEF:    Deferred processing
          RTV:    Right branch visited
          SUB:    Substring
CLEN:     Character length
RNL :     R# - lower result
RNU :     R# - upper result
```

b. AI.: APlist type information

```
    +-----+-----+-----+------------------+-----------+-+
    !     !     !     !                  !           !I!
AT. ! ATR ! MOD ! OPT !//////////////////!    RES    !O!
    !-----+-----+-----+------------------+-----------+-+
       5     5     6          26              17     1
```

```
ATR  :    Attributes
          LEN :   Processing I/O length
          IOC :   Processing I/O control
          CHAR:   Character type
          NUL :   Nonunity array length
          LEV :   Level 0
MOD  :    Mode
OPT2:     High 6 bits of AP.
RES  :    Reserved
IO   :    IO indicator
```

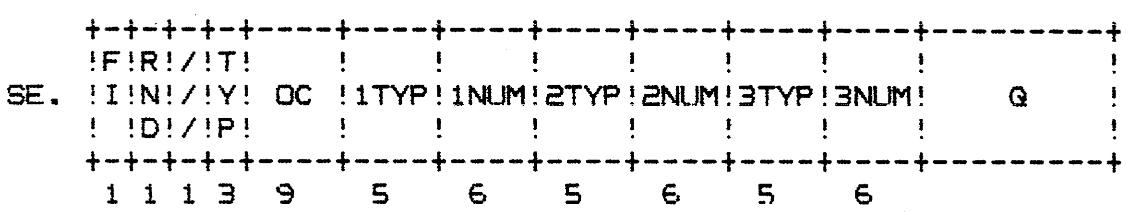
c. SC.: Skeleton descriptor word



```
     +-+----+----+----+----+-------------+-------------+
 SC. !/!PTYP!PNUM!MAXT!////!     PRO     !     AD      !
     +-+----+----+----+----+-------------+-------------+
      1  5    6    6    6        18            18
```

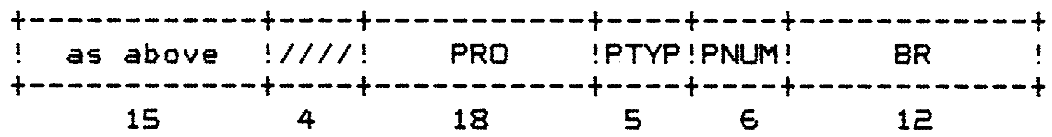
```
PTYP:    Call parameter type
PNUM:    Call parameter number
MAXT:    Maximum temporarys
PRO :    Call processor address
AD  :    Set address
```

d. SE.: Skeleton expansion word.



```
     +-+-+-+-+----+----+----+----+----+----+----+---------+
     !F!R!/!T!    !    !    !    !    !    !    !         !
 SE. !I!N!/!Y! OC !1TYP!1NUM!2TYP!2NUM!3TYP!3NUM!    Q    !
     ! !D!/!P!    !    !    !    !    !    !    !         !
     +-+-+-+-+----+----+----+----+----+----+----+---------+
      1 1 1 3  9    5    6    5    6    5    6
```

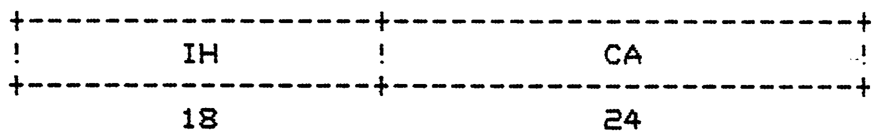
```
FI  :    First instruction in skeleton
RND :    Roundable operation
TYP :    NZ if branch or call
OC  :    Instruction opcode
nTYP:    Register type
nNUM:    Register number
Q   :    Address field
```



```
+------------+----+-----------+----+----+------------+
!  as above  !////!    PRO    !PTYP!PNUM!     BR     !
+------------+----+-----------+----+----+------------+
      15       4       18       5    6        12
```

```
PRO :    Call process address
PTYP:    Call process type
PNUM:    Call process number
BR  :    Branch index.
```

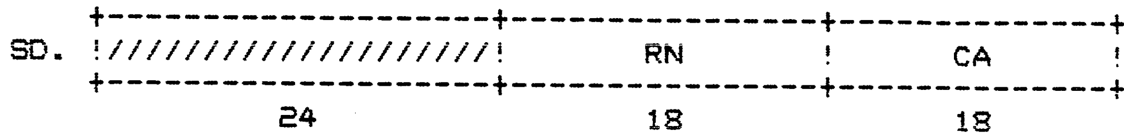
e. SY.: Symbol information



```
+------------------+-----------------------+
!        IH        !          CA           !
+------------------+-----------------------+
         18                   24
```

```
IH:      Symbol ordinal
CA:      Constant or offset
```

f. SD.: Sequence descriptor



RN: Register number
CA: Constant or offset

Routine Descriptions:

To be supplied.

a. BRIDGE: Initialization and main loop.

b. INS: Initialize new statement.

c. P=SEG: Reset turple counters.

d. TSP: Terminate statement processing.

e. PCS: Process current sequence.

f. IST: Issue statement temporary stores

g. CIR: Convert instruction to RLIST.

h. GOV: Get operand value.

i. LOP: Load operand.

j. ACA: Adjust CA.

k. IRI: Issue RLIST instruction.

l. LIR: Load intermediate result.

m. SDT: Store double word temporary.

n. SFT: Scan for temporary's processor.

o. STR: Store temporary result.

p. GAR: Get array result.

q. ROT: Read one turple.

r. ROW: Read one word (from buffer).

s. RFB: Refill buffer.

t. P=HDR: Header turple processor.

u. P=GFD: Generate file declarations. Comdeck COMFGFD.

v. P=PLIM: Process PLIM turple. Comdeck COMFPLI.

w. P=CDLOO: Specify object listing on/off.

x. P=CODOT: Specify zero/one trip DO loops.

y. P=CDCS: Specify user/fixed collation.

z. P=DATA: Process DATA turple.

aa. P=SEX: Process start of executables.

bb. ICP: Issue CP. and GPL. tables. Comdeck COMFICP.

cc. P=GAP: Process general APlist.

dd. P=FAP: Process function APlist.

ee. P=IOD: Process I/O data APlist.

ff. P=IOC: Process I/O control APlist.

gg. P=IOU: Process I/O unit APlist.

hh. IAW: Issue APlist word.

ii. PCL: Process character array item length.

jj. PAE: Process APlist turple operand.

kk. AIT: Add APlist to APL/IOA.

ll. FMU: Mark multiple CON. entries.

mm. PNA: Process namelist APlist.

nn. PAC: Process character APlist.

oo. PCO: Process character operand.

pp. IAC: Issue APlist for concatenation.

qq. ETT: Enter target temporary into APlist.

rr. PCI: Process character item.

ss. CEL: Check evaluated concatenation.

tt. ICE: Issue character expression.

uu. SAE: Stack APlist environment.

vv. PAE: Pop APlist environment.

ww. FAR: Form array reference.

xx. FVS: Form variable subscript.

yy. GCL: Get character length.

zz. SCB: Subsume constant character bias. Comdeck COMFSCB.

aaa. PCA: Process character APlist.

bbb. ECA: Enter character APlist.

ccc. PCAE: Enter FP character APlist.

ddd. IAP: Intrinsic function APlist processor.

eee. SUB: User function/subroutine processor.

fff. P=IOE: I/O function processor.

ggg. P=INE: Process intrinsic turple.

hhh. DER: Define function result.

iii. P=LRJ: Insert line number into RJ6 instruction.

jjj. CPL: Chain parameter list.

kkk. PPL: Process parameter list.

lll. SCA: Store character APlist.

mmm. CGE: Issue compiler generated function.

nnn. P=HSTO: Process character assignment.

ooo. P=HRLE: Process character relationals.

ppp. P=DOO: Process one trip DO loop.

qqq. P=DOZ: Process zero trip DO loop.

rrr. P=DOB: Process do begin.

sss. P=DOC: Process do conclusion.

ttt. P=PDE: Process do end.

uuu. P=IOLC: Process collapsed IO implied do.

vvv. P=PLA: Process label turple.

www. ALT: Add label to text.

xxx. GLT: Get label tag.

yyy. LAB: Label definition.

zzz. P=ENT: Process alternate entry point.

aaaa. P=PEX: Process exit macro.

bbbb. P=PASG: Process ASSIGN.

cccc. P=PGT: Process unconditional jump.

dddd. P=PAG: Process assigned goto.

eeee. P=PCG: Process computed goto.

ffff. P=JGOC: Process computed goto turples.

gggg. P=RGT: Process alternate return.

hhhh. P=IF: Collect IF flow.

iiii. EDT: Deferred turple processing.

jjjj. P=PAR: Process array turple.

kkkk. P=HCAT: Process concatenation turple.

llll. P=HSBS: Process substring turple.

mmmm. P=HCOL: Process colon turple.

nnnn. P=IM: Select integer multiply subskeleton. Comdeck COMFSIM.

oooo. P=ID: Select integer divide subskeleton. Comdeck COMFSID.

pppp. P=MASK: Select mask subskeleton. Comdeck COMFSMK.

qqqq. P=MOD: Select mod subskeleton. Comdeck COMFSMD.

rrrr. P=SHIFT: Select shift subskeleton. Comdeck COMFSSH.

ssss. P=BVD: Process start of vardim code.

tttt. P=EVD: Process end of vardim code.

uuuu. P=FIN: Terminate IL processing.

vvvv. MDV: Mark dimension vardims needed.

wwww. P=LCC: Process loader directives.

INDEX

N16

SECTION A: OVERVIEW

		MF
1.0	COMPILER STRUCTURE	B5
2.0	GLOBAL DATA STRUCTURES	B13
3.0	COMDECKS	F8

SECTION B: DECK AND ROUTINE DESCRIPTIONS

1.0	TEXTS	F15
1.1	FTN5TXT	F16
1.2	COMPLTXT	G5
1.3	CCGTEXT	G6
2.0	CRADLE ROUTINES	G8
2.1	FTN	G9
2.2	UTILITY	G12
2.3	PUC	G15
2.4	LINKAGE DECKS	H2
2.5	PEM	H9
2.6	ALLOC	H71
2.7	SNAP INTERFACE ROUTINES	H14
2.8	IDP	I1
2.9	INITIALIZATION ROUTINES	I12
3.0	FRONT END ROUTINES	J5
3.1	FEC	J6
3.2	FERRS	J13
3.3	LEX	J16
3.4	HEADER	B11
3.5	KEY	B14
3.6	CDDIR	C4
3.7	DATA	C6
3.8	DECL	C12
3.9	TYPE	D3
3.10	FMT	D5
3.11	IO	D10
3.12	PAR	E5
3.13	CONRED	F14
3.14	STMTF	G5
3.15	LABEL	G6
3.16	FSKEL	G11
4.0	QCG	G13
5.0	REAR END ROUTINES	I15
5.1	REC	I16
5.2	RERRS	J2
5.3	FAS	J3
5.4	MAP	K4
5.5	LIST	L2
6.0	CCG	L9
6.1	CCGC	L10
6.2	GSKEL	L16
6.3	CCG ROUTINES	M1
6.4	BRIDGE	M8

INVERTED CROSS-REFERENCE LISTING OF COMDECKS AND THEIR CALLERS.

```
-COMDECK-/-------------------- CALLING DECK/COMDECK -------------------/

CCOMRPV   UTILITY
COMACPU   FTN5TXT
COMADEF   FTN5TXT
COMAERR   FERRS     RERRS
COMAMGM   FTN5TXT
COMAQCG   QCGC      FUN       REG       GEN
COMCMCS   IDP
COMCPAC   INIT00
COMCSBM   UTILITY   IDP
COMDDMT   CSNAP     FSNAP     RSNAP
COMFCIP   FTN       INIT00    INIT20    OVL10     OVL20
COMFDST   DECL
COMFECB   PUC
COMFFEI   INIT00    INIT10    INIT21
COMFERR   FERRS     RERRS
COMFGFD   GEN       BRIDGE
COMFGDI   INIT00    INIT10
COMFICP   GEN       BRIDGE
COMFISA   GEN       CCGC
COMFITS   QCGC      CCGC
COMFMAV   GEN       CCGC
COMFOSC   GEN       CCGC
COMFPLI   GEN       BRIDGE
COMFROR   INIT00    INIT10    INIT21    INIT22
COMFSCB   FUN       BRIDGE
COMFSCS   FEC       RLINK
COMFSID   BRIDGE
COMFSIM   BRIDGE
COMFSKL   QSKEL     FSKEL
COMFSMD   BRIDGE
COMFSMK   BRIDGE
COMFSSH   BRIDGE
COMFTTL   FTN
COMFUSE   QCGC      FAS       CCGC
COMSPSU   COMFWIN   QCGC      CCGC      FTN5TXT   FAS
COMFWIN   QCGC      CCGC
COMSEIS   COMFSKL   QSKEL     FSKEL     INIT00    INIT10    INIT21    CONRED
          GEN
COMSERR   PEM       FERRS     RERRS
COMSIOC   FTN5TXT   IO        FAS
COMSLBT   PUC       REC       LIST
COMSQCG   FSNAP     QCGC      FUN       REG       GEN
COMSQRF   COMFSKL   QSKEL     FSKEL     QCGC      FUN       REG       GEN
COMSPBD   FTN5TXT
COMSSYM   SYMDEFS   FTN5TXT
COMSSYC   FERRS
COMSTAB   PUC       CCGLINK   CCGC      INIT22
COMSTAD   CGHCDTD   COMSTAB   PUC       CCGLINK   CCGC      INIT22
COMSTAS   CGHCSTD   COMSTAB   PUC       CCGLINK   CCGC      INIT22
DEFINS    COMFSKL   QSKEL     FSKEL     FUN       REG       GEN
FSCALE    CONRED
OPTIONS   FTN5TXT
PARSKEL   COMFSKL   QSKEL     FSKEL
SKEL      COMFSKL   QSKEL     FSKEL     CSKEL
SKOP      COMFSKL   QSKEL     FSKEL     CONRED    GEN       BRIDGE    CSKEL
SKPCONQ   COMFSKL   QSKEL     FSKEL     CONRED    QCGC
SKPSET    COMFSKL   QSKEL     FSKEL     CONRED    QCGC
```

COMAIDP	FTN5TXT			
COMATOK	IDP	LEX		
COMCBUB	IDP	LEX		
COMCBUN	IDP	LEX		
COMCCDD	UTILITY	IDP		
COMCCFD	FTN	PUC		
COMCCIO	UTILITY	IDP		
COMCCOD	FTN	IDP		
COMCDXB	UTILITY	IDP		
COMCIDP	IDP			
COMCLFM	IDP			
COMCMNS	UTILITY			
COMCMVE	UTILITY			
COMCRDC	UTILITY	IDP		
COMCRDW	UTILITY	IDP		
COMCRSR	IDP			
COMCSFN	UTILITY	IDP		
COMCSST	UTILITY			
COMCSTF	INITCO			
COMCSVR	IDP			
COMCSYS	IDP			
COMCTOK	IDP	LEX		
COMCWOD	UTILITY	IDP		
COMCWTC	IDP			
COMCWTH	UTILITY			
COMCWTO	UTILITY			
COMCWTW	UTILITY	IDP		
COMCXJR	IDP			
COMCZTB	UTILITY	IDP		
COMDTOK	FSNAP			
COMPCOM	FTN			
COMQSVR	PEM	IDP		
COMSIDP	CSNAP	IDP	FSNAP	RSNAP
COMSTOK	IDP	FSNAP	LEX	
FA=CLO	UTILITY			
FA=DEFS	FTN5TXT			
FA=EOF	UTILITY			
FA=EOR	UTILITY			
FA=FLSH	UTILITY			
FA=OPE	UTILITY			
FA=RDC	UTILITY			
FA=RDW	UTILITY			
FA=RWX	UTILITY			
FA=SET	UTILITY			
FA=WTC	UTILITY			
FA=WTW	UTILITY			
OPRDEFS	RLINK	CSKEL		

*********** MRRC04H //// END OF LIST ////

*********** MRRC04H //// END OF LIST ////

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

<NOMOS>

1 CROSS-REFERENCE LISTING OF DECKS/COMDECKS, AND COMDECKS THAT THEY CALL.

```
-CALLER- /--------------------- CALLED COMDECKS ----------------------/

HISTORY
*COMDKS*
CCOMRPV
CGHCDTD   COMSTAD
CGHCSTD   COMSTAS
COMACPU
COMADEF
COMAERR
COMAMGM
COMAQCG
COMCMCS
COMCPAC
COMCSBM
COMDDMT
COMFCIP
COMFDST
COMFECB
COMFFEI
COMFERR
COMFFFN
COMFGFD
COMFGOI
COMFICP
COMFISA
COMFITS
COMFMAV
COMFMFP
COMFOSC
COMFPLI
COMFRNC
COMFROR
COMFSCB
COMFSCS
COMFSID
COMFSIM
COMFSKL   COMSEIS   SKPSET    SKPCONO   SKOP      COMSORF   DEFINS    SKEL
          PARSKEL
COMFSMD
COMFSMK
COMFSSH
COMFTTL
COMFUSE
COMSPSU
COMFWIN   COMSPSU
COMSEIS
COMSERR
COMSIOC
COMSLBT
COMSOCG
COMSORF
COMSPBD
COMSSYM
COMSSYC
COMSTAB   COMSTAD   COMSTAS
COMSTAD
COMSTAS
DEFINS
ESCALE
```

```
FWACOM
OPTIONS
PARSKEL
SKEL
SKOP
SKPCONQ
SKPSET
SMACROS
SYMDEFS   COMSSYM
ZZZCOM
*TEXTS*
FTN5TXT   OPTIONS  >FA=DEFS  COMADEF  >COMAIDP  COMACPU  COMAMGM  COMSIO
          COMSPSU  COMSPBD   COMSSYM
CWEOR1
*FTN*
FTN       COMFCIP  >COMPCOM  COMFTTL  >COMCCFD  >COMCCOD
UTILITY  >COMCCDD  >COMCDXB  >FA=SET  >COMCMVE  CCOMRPV  >COMCMNS  COMCSB
         >COMCSFN  >COMCSST  >COMCWOD >COMCZTB  >COMCCIO >COMCRDC  >COMCRD
         >COMCWTH  >COMCWTO  >COMCWTW >FA=CLO   >FA=EOF  >FA=EOR   >FA=FLS
         >FA=OPE   >FA=RDC   >FA=RDW  >FA=RWX   >FA=WTC  >FA=WTW
LISTLNK
PUC       COMSTAB  COMSTAD   COMSTAS  COMSLBT  >COMCCFD  COMFECB
QCGLINK
CCGLINK   COMSTAB  COMSTAD   COMSTAS
CSNAP    >COMSIDP  COMDDMT
PEM       COMSERR  >COMQSVR
ALLOC
IDP      >COMATOK  >COMSIDP  >COMSTOK >COMCLFM  >COMCIDP >COMCBUB  >COMCBU
         >COMCCDD  >COMCCIO  >COMCCOD >COMCDXB  COMCMCS  >COMCRDC  >COMCRD
         >COMCRSR  COMCSBM   >COMCSFN >COMCSVR  >COMCSYS >COMCTOK  >COMCWO
         >COMCWTC  >COMCWTW  >COMCXJR >COMCZTB  >COMQSVR
INIT00    COMFCIP  COMCPAC   >COMCSTF COMFGOI   COMFFEI  COMSEIS   COMFRO
INIT10    COMFGOI  COMFFEI   COMSEIS  COMFROR
INIT20    COMFCIP
INIT21    COMFFEI  COMSEIS   COMFROR
INIT23
*FRONT*
FEC       COMFSCS
FERRS     COMSSYC  COMAERR   COMSERR  COMFERR
FLINK
FSNAP    >COMSIDP  COMSQCG   >COMSTOK COMDDMT  >COMDTOK
LEX      >COMATOK  >COMSTOK  >COMCTOK >COMCBUB  >COMCBUN
HEADER
KEY
CDDIR
DATA
DECL      COMFDST
TYPE
FMT
IO        COMSIOC
PAR
CONRED    SKPSET   SKPCONQ   COMSEIS  FSCALE    SKOP
STMTF
LABEL
*QCG*
QCGC      COMAQCG  COMSQCG   COMSQRF  SKPSET    SKPCONQ  COMFWIN   COMSPS
          COMFITS  COMFUSE
QSKEL     COMFSKL  COMSEIS   SKPSET   SKPCONQ   SKOP     COMSQRF   DEFINS
          SKEL     PARSKEL
FUN       COMAQCG  COMSQCG   COMSQRF  DEFINS    COMFSCB
REG       COMAQCG  COMSQCG   COMSQRF  DEFINS
GEN       COMAQCG  COMSEIS   COMSQCG  COMSQRF   DEFINS   SKOP      COMFIS
          COMFOSC  COMFMAV   COMFICP  COMFGFD   COMFPLI
CWEOR2
*REAR*
```

```
RERRS      COMAERR    COMSERR    COMFERR
RLINK     >OPRDEFS    COMFSCS
RSNAP     >COMSIDP    COMDDMT
FAS        COMSIOC    COMSPSU    COMFUSE
ZEROLNK
MAP
LIST       COMSLBT
CWEOR3
*BRIDGE*
CCGC       COMSTAB    COMSTAD    COMSTAS    COMFWIN    COMSPSU    COMFITS    COMFUS
           COMFISA    COMFMAV    COMFOSC
BRIDGE     SKOP       COMFGFD    COMFPLI    COMFICP    COMFSCB    COMFSIM    COMFSI
           COMFSMK    COMFSMD    COMFSSH
CSKEL      SKOP      >OPRDEFS    SKEL
FSKEL      COMFSKL    COMSEIS    SKPSET     SKPCONQ    SKOP       COMSQRF    DEFINS
           SKEL       PARSKEL
INIT22     COMSTAB    COMSTAD    COMSTAS    COMFROR
CWEOR4
*FRAME*
OVL00
OVL10      COMFCIP
OVL20      COMFCIP
OVL21
OVL22
OVL23
CWEOR5
COPYWS

***********          MRR0042  //// END OF LIST ////
```


```
***********          MRR0042  //// END OF LIST ////
```