MICRODATA

Express COBOL

INTERNAL DESIGN SPECIFICATION

PREPARED BY

CALIFORNIA SOFTWARE PRODUCTS, INC.

525 NORTH CABRILLO PARK DRIVE, SUITE 300

SANTA ANA, CALIFORNIA 92701

OCTOBER, 1979

~~MARCH, 1978~~

CALIFORNIA SOFTWARE PROD

Table of Contents

change

change ↓

1. INTRODUCTION

The purpose of this specification is to provide a working document for the MICRODATA COBOL implementation. It is intended to be expanded at later date and used as a maintenance manual.

Chapter 1 covers the design of the product and some of the rationale behind the selection of design approach.

Chapters 2 and 3 deal in detail with the internal intricacies of the compiler: Compiler Data Structure, MOM Instruction Repertoire, Stack Management Technique, etc.

Chapter 4 describes the seven phases of the compiler. Each phase is described chronologically and work files are discussed as the compiler creates or first references them.

Chapter 5 describes the generated object code and its interface with various tables at runtime.

Chapter 6 is reserved for the runtime library descriptions.

2. PRODUCT OVERVIEW

2.1 Compiler

The compiler is multiphase, i.e., modularly coded by compiler function and COBOL language division. It is also multipass in that it reads the user's program in its original and subsequent encoded forms more than once. It utilizes random access devices for the storage of encoded text streams. The compiler will translate 1974 ANS COBOL language statements into machine language instructions which will be output in the form of a relocatable object module which can then be loaded by the system loader along with selected runtime I/O and library routines for subsequent execution of the user task. The compiler outputs listings of the user program along with various debugging aids to help the user in getting the program operational.

2.2 Generated Code

Due to the nature of the ~~MICRODATA~~ Express computer, it is not considered practical to attempt to generate in-line code for the majority of the functions of the COBOL language. The primary reason is that the ~~MICRODATA~~ Express computer is not a business oriented computer; that is, decimal arithmetic and string manipulation must be done with software. Because of this, the design of the COBOL object system includes library routines for performing most functions and compiler generated code consists of a series of calls to these routines.

2.3 Implementation Language

A special interpretive software development language, MOM (Macro Operation Module), is used in the development of the compiler.

MOM source language statements are processed by a special MONITRAN compiler that produces an object file which is linked with MOM interpreter routines to produce

The MOM statements are similar to any Macro type language with one basic exception: instead of being expanded into in-line code, they are executed by the interpreter (segment of the COBOL Compiler) that decodes them and calls the appropriate subroutine to perform the MOM function. This approach permits a substantial saving in main memory to be realized over standard assembly language programming approaches.

Another advantage to MOM instructions is that they are designed to operate on dynamic stacks. Thus, their use allows most of the needed tabular information during compilation to be kept in main memory. This reduces the use of secondary storage for passing of information from phase to phase and the attendant encoding and decoding of that information. The use of stacks not only speeds up the overall compilation, but permits the spilling of tables to disk when main memory becomes full. This means the user can compile a sizable program with a limited amount of memory.

One final advantage to MOM coding is that it is clearer than assembly language coding. Therefore, it is easier to code and debug, and usually easier to understand the code of other programmers.

While there is some overhead for the decoding of each MOM the number of instructions performed during the entire compilation is actually fewer than with conventional assembly language coding. This occurs because:

- Stack management eliminates unnecessary reading, writing, encoding and decoding of text streams.
- The powerful nature of MOMs, (average one MOM instruction equals 20 assembly language instructions) greatly reduces the number of overlays required in the compiler. This translates to fewer loads, less swapping, less interface communication, fewer interface problems and ultimately fewer complaints from users when the product is released.

3. COMPILER DESIGN

3.1 Compiler Interpreter Module

3.1.1 Overview

The MOM (Macro Operation Module) Interpreter is a group of routines and an interpretive control loop that simulates a hypothetical "compiling machine." The core of the interpreter is the control loop which interprets MOM instructions, executes them, and maintains the MOM pseudo-instruction counter. The "MOM machine," is a stored program machine with single-address instructions. The primary difference between the "MOM machine" and a conventional one is the organization of memory into a number of single memory locations plus a number of named stacks whose lengths vary dynamically during compilation.

A stack is a last-in-first-out memory in which the value most recently stored in the stack (the "Top") may be removed, exposing the value next most recently stored, and so on. Stacks are dynamically allocated and, if necessary, spilled to auxiliary storage and their contents may be searched, added to, or entirely deleted.

The Work Stack, which is the principal operating element of the MOM machine, is special in that many MOMs address the top element implicitly, in addition to an explicit operand. The Exit Stack contains subroutine return points as well as the calling program's Answer Box setting.

The Answer Box is a two-position switch which is set by some MOMs and tested by others.

Control pseudo-instructions deserve special attention here. The jump MOMs (J, JS, QJS, etc.) alter the MOM location counter. J (jump) merely resets the location counter. JS (jump to subroutine) and QJS (query jump to subroutine) perform the same function but in addition they create a link entry on a push-down stack called the Exit Stack. The execution of PX (pop and exit) causes the most recent entry on the Exit Stack to pop up and it is used to reset the MOM location counter.

3.1.2 Basic Interpretive Data Structure

Inherent to the design and understanding of the MOM interpreter are basic data structures either referenced as MOM instruction operands directly, or maintained by the various MOM subroutines.

1. Work Stack

   The top location of the work stack is referred to as W0, the next to the top location is referred to as W1 and so on. Many MOMs refer implicitly to W0. Two operations basic to many MOMs are "fetch" and "pop". Fetching is the operation of extending the work stack by one location, so that what was W0 becomes W1, etc., and copying the value fetched into the work stack as the new value of W0. Popping is the operation of reducing

the length of stack (by one or by some stated amount). Popping one location from the work stack makes what was W1 into W0, etc. This operation is what is referred to when the term "pop" is used without qualification.

2. MOM Subroutines and the Exit Stack

Although only a minority of MOM subroutines are actually recursive, MOM coding is usually done in a way to facilitate recursion, since it sometimes happens that a routine written to be non-recursive becomes recursive as a result of modification of program specifications or implementation strategy. Thus, all MOM subroutine calls are potentially recursive. The effect of the jump to subroutine MOM (JS) is to lengthen the exit stack by one ~~position~~ location and store there the location of the MOM following the JS. The "exit" operation retrieves this return point from the exit stack and reduced the length of the exit stack by one ~~position~~ location.

3. Answer Box

The Answer Box is a two-position switch set by MOMs whose mnemonic names begin with Q. It affects the execution of MOMs whose mnemonic names end with :T and :F suffixes. A MOM with a :T suffix, such as J:T will be executed if the Answer Box is set True and performs no operation if the Answer Box is set False. The suffixes :T and :F may only be appended

to Conditional MOMs. The Answer Box is saved on the exit stack so that each subroutine level has its own answer box. This means that ordinarily invoking a subroutine does not affect the Answer Box in the calling routine. The exception is a subroutine which is called by the QJS MOM and some XEC routines which are expected to set the Answer Box.

4. String

Each COBOL symbol is formed in the string area by QQ or QQA MOM or an XEC routine and is in packed bias 38 internal code. The symbol is hashed from the internal code and the hash is used as the argument for reserved word recognition and also for COBOL source symbol collection.

5. Common

The Common area is used primarily in conjunction with constructing and then registering of a group into a stack. The Common area is referenced implicitly by the REGF MOM and explicitly by others.

6. Field

The Field area is used primarily to implode from or explode into the information. It is referenced implicitly by IMPL and EXPL MOMs and explicitly by others.

7. Internal Representation of COBOL Words

Valid COBOL word characters are assigned values in the range 0 through 37 as follows:

0	Blank
1	Hyphen
2 - 11	Numeric Characters (0 to 9)
12 - 37	Alphabetic Characters (A through Z)

A three-character symbol is packed into a half-word according to the algorithm:

((C1*38)+C2)38+C3 where Cn is the assigned character value

8. W0, W1, W2---etc., as MOM Operands

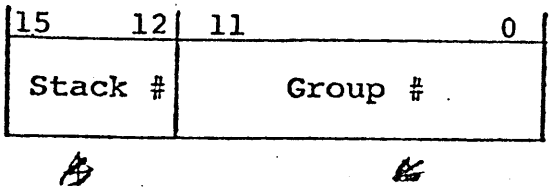
A line containing a MOM operation code which references memory may have as its operand a symbol of the form W0, W1, W2 standing for the current top word of the work stack, etc. When such a line is translated, the operand will reference a fixed location in the compiler data area. The execution of a fetch or pop type of MOM will lengthen or shorten the work stack. Only five halfwords of the work stack (W0 - W4) are kept in the fixed area. The rest of the stack is dynamic. All stack references are made through a stack pointer in W0.

9. Stack Pointer

A position in a stack can be indicated by giving the stack number and the distance in groups from the bottom of the stack. These two values, stack number and group number, are combined together in a

Group size depends on the particular stack and can vary as the compiler phase changes. Group sizes are contained in the compiler data area "group base" in each phase. 3-5

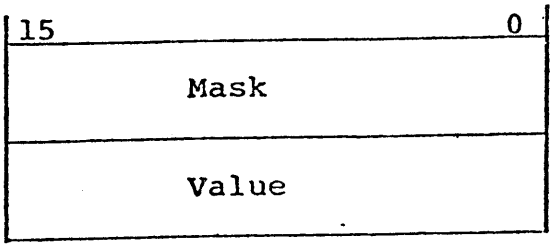
single element called a "Stack pointer". The position indicated by a stack pointer remains effective even after the stack management routines have rearranged stack memory.



The size of field 1 (stack #) implies that the contents of stack 16 or 17 cannot be accessed directly, (~~Exit~~ Work and Exit stacks, respectively) The size of field 2 (group #) implies a maximum of 4096 for each of following COBOL names: data, file and procedure.

10. Earmark

An Earmark is a data structure that is used to interrogate, ~~define or delete~~, a specific characteristic of an ~~earmark~~ value in the top halfword of the work stack. Specific MOMs reference Earmarks as their operand value. Earmarks consist of two halfwords: a mask and a value.



Mask	A bit pattern used to isolate the trait value in the stack trait word
Value	A bit pattern used to compare trait values after isolation

11. Character Scanning

Character Scanning is performed during the syntactical analysis of COBOL source input. The primary routine in control of scanning is named NXTCHR. Calls to NXTCHR can be made either by the character handling MOMs (QC, QQ, etc.) or directly in XEC MOM. NXTCHR returns the next valid COBOL character and is

responsible for source sequence, comment, debugging and continuation card processing. Several variables are maintained during the COBOL source input scanning to record the information being scanned and the position of scan. At all times, the variable CRNTCH holds the source statement character which is currently being inspected. The position of the current character within the input record is recorded in a variable CHPOS. The position of the current character within the input record is recorded in a variable BUFPT.

12. Stack Statistics

Associated with each COBOL stack is a group of information items called stack statistics.

Stack statistics consists of the following entries.

1. ~~Search~~ Group size
2. Memory Top
3. Memory Bottom
4. Disk word count
5. Group size
6. ~~Core~~ Memory Top
7. ~~Core~~ Memory Bottom
8.

Group size is a count of the number of words in each stack group.

~~Core~~ Memory top and ~~core~~ memory bottom are addresses that define the extent of the main memory ~~core~~ resident portion of a stack. ~~Core~~ Memory top is the address of the first word on the next group to be added to the stack. When ~~core~~ memory top equals ~~core~~ memory bottom, the stack is empty.

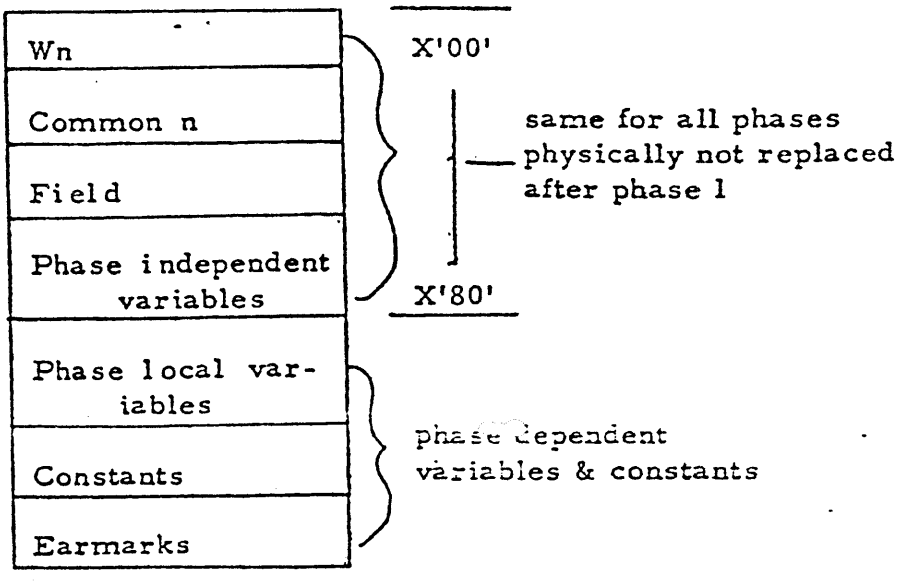
Memory
~~Core~~ top of stack n can never be larger than memory ~~core~~ bottom of stack n+1. When an operation on stack n would cause memory ~~core~~ top of stack n to exceed memory ~~core~~ bottom of stack n+1, the stack management routines reallocate the stacks to allow stack n to grow.

Disk word count is the number of words from the stack that have been written to disk. When disk word count is zero, the disk resident portion of a stack is empty. Most stacks are initialized in the empty state; that is, disk spill count equals zero and memory ~~core~~ top equals memory ~~core~~ bottom.

Search size is the number of words to be compared during a Query search (QSRCH) operation.

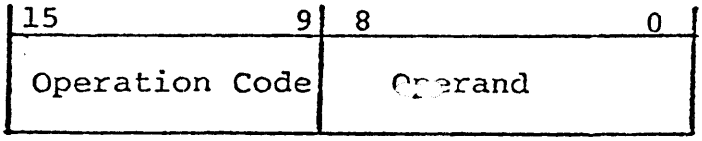
important

13. Data Base



3.1.3 MOM Instruction Format

- MOM instructions are made up of an Operation Code and an Operand Field



a. Operation Code

The operation code is a seven-bit field that identifies the MOM. It is numbered so that the leftmost ~~two~~ two bits are used to ~~test the~~ determine the ~~types~~ types of MOM. The leftmost bit indicate whether the MOM is conditioned (either :T or :F) or not conditioned. If the MOM is conditioned, the next leftmost bit indicate either True conditioned (:T) or False conditioned (:F). ~~If~~ The operation code assignment is as follows:

Op codes = 0 – 31 :T, execute the MOM if the Answer Box is True

= 32 – 63 :F, execute the MOM if the Answer Box is False

= 64 – 95 ~~Cond~~ execute always for conditional MOMs with no suffix.

= 96 – 127 execute always for No suffix unconditional MOMs. ~~are~~ are all one for unconditional MOMs.

b. Operand Field

The Operand Field is a nine-bit field that addresses static storage, data structure, stack number, character constant or immediate value.

3.1.4 MOM Instruction Repertoire Description

3.1.4.1 Operand Type

The following list defines the operand functional codes:

1. A Static memory cell address

   Static memory cells are grouped under ~~the~~ the static base. The address A is word relative to that base. The maximum size of A is 511. W0 through W4 are static memory cells.

2. S Stack number

   Stack numbers are immediate data

3. R Rung

   Rung number is immediate displacement value into the stack group.

4. C Character is immediate eight-bit display format character representation.

5. E Earmark address

   Earmarks are under the static base. Earmark address is a word relative displacement into the base.

6. F Flag number ~~static~~

Flags are two-position (TRUE/FALSE) switches. There are 64 flags. Flag number is immediate data.

7. P Pattern address

Patterns are under the static base.

Pattern address is a word relative displacement into that base.

8. SJ Self relative jump

A self-relative word address is ~~either~~ greater than -256 and less than 256. ~~plus or minus 256~~.

9. J Jump Address

A jump address is either local or global.

A local jump address is forward self-relative value of less than 512.

A global jump is an index into a jump table of 16-bit addresses.

10. N Number

is immediate numeric data.

- In the descriptive paragraphs, an arrow ⟶ denotes replacement of the contents of the location on the right by a copy of the value on the left. "SJ ⟶ MOM Location Counter" means that the MOM location counter is incremented by SJ.

Parentheses around an expression denoting a memory location denote the contents of that location. The phrase "new W0" indicates that the work stack is lengthened so that the previous (W0) becomes (W1). The phrase "pop one" means

that the top item is removed from the work stack and (W1) becomes (W0). The phrases "set True" and "set False" always refer to setting the Answer Box state either True or False.

The use of the vector operator, $\rightarrow$, indicates that the value must be a stack pointer. The expression $\overrightarrow{(A)}$ refers to the location pointed to by the pointer in location A.

3.1.4.2 MOM Instruction Descriptions

1. Add

   ADD A

   (W0) + (A) $\longrightarrow$ W0

2. Add to Memory

   ADDM A

   (W0) + (A) $\longrightarrow$ A

3. Add to Memory and Pop

   ADDMP A

   (W0) + (A) $\longrightarrow$ A and then pop one

4. Logical AND

   AND A

   (W0) $\wedge$ (A) $\longrightarrow$ W0

5. Diminish and Jump

   DIMJ SJ

   (W0) - 1 $\longrightarrow$ W0. If (W0 > 0), then jump, otherwise pop one and execute the next MOM in sequence

6. Divide

DIV A

(W0)/(A) ⟶ W0

7. Exclusive Or

EOR A

(W0) ⊕ (A) ⟶ W0

8. Error Diagnostic

ERR N
ERRC N

Place Error Code, N, and statement number in EX-file

ERR gives a diagnostic on the token most recently accepted; ERR gives a diagnostic on the last token examined.

9. Empty Stack

ES S

Stack S is made empty

10. Explode Fields

EXPL P

(W0) is exploded into fields, FIELD n, according to the field pattern, P, then pop one

11. Fetch

F A

(A) ⟶ New W0

12. Fetch Immediate Left

FIL N

X'NN00' ⟶ New W0

zero not alpha O

13. Fetch Immediate Right

FIR N

X'00NN' ⟶ New W0

14. Fetch Pointer to Top Group

FPTG S

Zero rung pointer of the current top group ⟶ New W0

15. Fetch Ralative

FR R

$(\overrightarrow{(W0)} + R) \longrightarrow$ New W0

16. Implode Fields

IMPL P

IMPL is the inverse of EXPL; the destination word is new W0

17. Jump

J SJ

SJ ⟶ MOM location counter. The next MOM to be executed is at location SJ.

18. Jump to Subroutine

JS J

The current MOM location counter is pushed onto the top of the exit stack. The ANSWER BOX is also saved on the exit stack and will be restored when control is returned from the subroutine. Finally, J ⟶ MOM location counter. The next MOM to be executed is at location J.

19. Multiply

MPY A

(W0) * (A) ⟶ W0

20. Or

OR A

(W0) ∨ (A) ⟶ W0

21. Or Memory and Pop

ORMP A

(A) ∨ (W0) ⟶ A, Pop W0

22. Pop and Jump

PJ SJ

Same as Jump except W0 is popped

23. Push to Stack

PSH S

The stack S is extended one new rung with (W0)

24. Push to Stack and Pop

PSHP S

The stack S is extended one new rung with (W0), then pop one

25. Pop Work Stack

PW N

Pop N words from the Work Stack

26. Pop and Exit

PX N

N entries are popped from the Work Stack. The top entry of the exit stack is popped to define the new value of the MOM location counter and, if the entry was placed there by the

JS MOM, a new value of the ANSWER BOX.

27. Publish Character

PUBC C

Output character C to the print buffer

28. Publish

PUBL A

Output the character string to the print buffer. The address A is the address of the string to be published. The first word of the string is a character count; remaining words contain the characters two per word.

29. Query Adjust Pointer

QAP N

$\overrightarrow{(W0)} + N \longrightarrow \overrightarrow{W0}$. If (W0) is a valid pointer, set True. Otherwise, pop one and set False.

30. Query Bit

QB N

If Bit N of (W0) is on, set True; otherwise set False.

$0 \leq N \leq 15$

31. Query Character

QC C

If C equals CRNTCH, set True. Otherwise, set False.

32. Query Character and Advance.

QCA C

If C equals CRNTCH, set True and advance to next character positior Otherwise, set False.

33. Query Character, Space and Advance

QCSA C

If C equals CRNTCH and CRNTCH is followed by space(s), set True and advance to next character position. Otherwise, set False.

34. Query Earmark

QE E

If (W0)Λ(E)= (E+1), set True. Otherwise, set False.

35. Query Earmark and Pop

QEP E

If (W0)Λ(E)= (E+1), set True. Otherwise set False. In either case, pop one.

36. Query Equal

QEQ A

If (W0) = (A), set True, otherwise set False.

37. Query Fetch

QF A

If (A) is non-zero, (A)$\longrightarrow$New W0, and set Answer Box to True. If zero, set False

38. Query Fetch Relative

QFR R

If $((\overrightarrow{W0}) + R)$ is non-zero, $((\overrightarrow{W0}) + R) \rightarrow$ New W0. Set Answer Box to True. If $((\overrightarrow{W0}) + R)$ is zero, set False.

39. Query Flag

QFL F

If Fth Flag is one, set True. Otherwise, set false. $0 \leq F \leq 63$

40. Query Jump to Subroutine

QJS J

The current MOM location counter is pushed on the top of the exit stack. The Answer Box is not saved. When control returns from the subroutine, the Answer Box will contain the value last established in the subroutine. J→MOM location counter. The next MOM to be executed is at location J.

41. Query Less or Equal

QLE A

If $(W0) \leq (A)$, set True; otherwise, set False.

42. Query Quote

QQ Q

Q is the number of a reserved word in the Reserved Word Table. If the current source string matches the reserved word pointed to, the following steps are performed in order:

1. If the Comma Flag is still on, issue a diagnostic and reset the Comma Flag.
2. Move T (trait) field to Trait Word.
3. Set Answer Box True
4. Set QQ performed flag

Otherwise set the Answer Box False

43. Query Quote and Advance

QQA Q

Q is the number of a reserved word in the Reserved Word Table. If the current source string matches the reserved word pointed to, perform the following:

1. If QQ performed flag is on, clear the QQ performed flag. Otherwise, perform 1 through 3 as described in QQ.

2. Check FIPS level and issue diagnostic if required.
3. Set COMMA FLAG if ',' or ';' precedes the word.
4. If P (followed by a period bit) is off, go to b). Otherwise, check a period and a space follows. If not, issue a diagnostic.
5. If A (area A bit) is on, check to see that the word is in Area A. Issue a diagnostic if the word is in Area B.
6. If M (Both Area A and B allowed flag) is on, set the Area A Flag if the word resides in Area A.
7. If neither A nor M is on, and the word resides in Area A, issue a diagnostic.
8. The current scanning pointer is advanced to the next space character

Otherwise, set the Answer Box False.

44. Query Search

QSRCH S

Stack S is searched for a group matching COMMON. If one is found, the Answer Box is set True and a pointer to the group is placed in a new WO. Otherwise, the Answer Box is set False.

2. Set the Answer Box to True

3. Set COMMA FLAG if ',' or ';' follows the word

4. The current scanning pointer, CHPOS, is advanced to the next space character

Otherwise, set the Answer Box Fals

44. Query Search

QSRCH S

Stack S is searched for a group matching COMMON. If one is found, the Answer Box is set True and a pointer to the group is placed in a new W0. Otherwise, the Answer Box is set False.

45. Query Take off Top

QTT S

If the Stack S is empty, set False. Otherwise, pop the top rung of Stack S and place it in new W0 and set Answer Box True.

46. Register and Fetch Pointer

REGF S

The content of COMMON is placed on the top of Stack S as a stack group and a pointer to ~~the first word of~~ the group entry is placed in a new W0.

47. Reset Flag

RFL F

Reset the flag specified by F.

$0 \leq F \leq 63$

48. Replace

RPL A

$(A) \longrightarrow W0$

49. Replace Relative

RPLR R

$((\overrightarrow{W0})+R) \longrightarrow W0$

50. Set Flag

SFL F

Set the flag specified by F.

$0 \leq F \leq 63$

51. Shift Logical Left

SLL N

Shift (W0) left N bits

52. Shift Logical Right

SLR N

Shift (W0) logical right N bits

53. Store Relative

SR R

$(W0) \longrightarrow (\overrightarrow{W1}) + R$ and pop one

54. Store

ST A

$(W0) \longrightarrow A$

55. Store and Pop

STP A

(W0) ⟶ A and pop one

56. Store Relative and Pop

SRP R

(W0) ⟶ ((W1)+R), then pop two

57. Subtract

SUB A

(W0) - (A) ⟶ W0

58. Switch

SWT A

(W0) ⟶ A and original (A) ⟶ W0

59. Tab

TAB N

Set current output position to column N

60. Tally

TLY A

(A) + 1 ⟶ A

61. Execute

XEC N

XEC is a generic name for any interpretive instruction which does not require an operand. The instruction subroutine address located at the Nth element of the XEC jump vector is executed just as any other interpretive instruc-

tion and control continues with the next MOM

62. Index Next Instruction

XNI A

(A) is added to the operand of the next MOM prior to execution

63. Zero Memory

ZM A

0 → A

3.2 Stack Management Concept

One of the major advantages of a compiler that operates interpretively is that the procedural code takes less memory than is the case with compilers that employ in-line assembly language code. This means that more memory is available for keeping tabular information about the source program and that much less data needs to be written out to secondary storage. This saves time and secondary storage space.

The MOM language is specifically designed for management of information kept in "stacks" in memory. Some of the major stacks kept by the compiler are:

Symbol Table	A record of the definition of each unique symbol presented in the COBOL source program
Data Stack	Describes the characteristics of data items in the source program
Procedure Stack	Describes all procedure names defined in the source program
File Stack	Describes all of the file names used in the source program

Stacks are primarily constructed in Pass 1. They are utilized in all of the passes. There are generally three classes of information needed by the compiler during its operation: (1) information of global interest to more than one phase, (2) information of local interest, usually needed only sequentially and infrequently, and (3) compilation control information

(compiler options). The first type (type 1 information) is kept in stacks. Type 2 is output to files to be used in subsequent phases. Type 3 is kept in bit tables in special data bases and queried as required. These tables are also kept in memory and are utilized throughout the compilation.

MOMs work by moving information between stacks and a work stack. Items in the work stack can be examined, stored, tested and evaluated. The work stack operates like a series of registers but with push/pull capability. The MOM language has a great deal of machine independence built into it.

The stacks all share a common memory pool as shown in Figure 3-3.

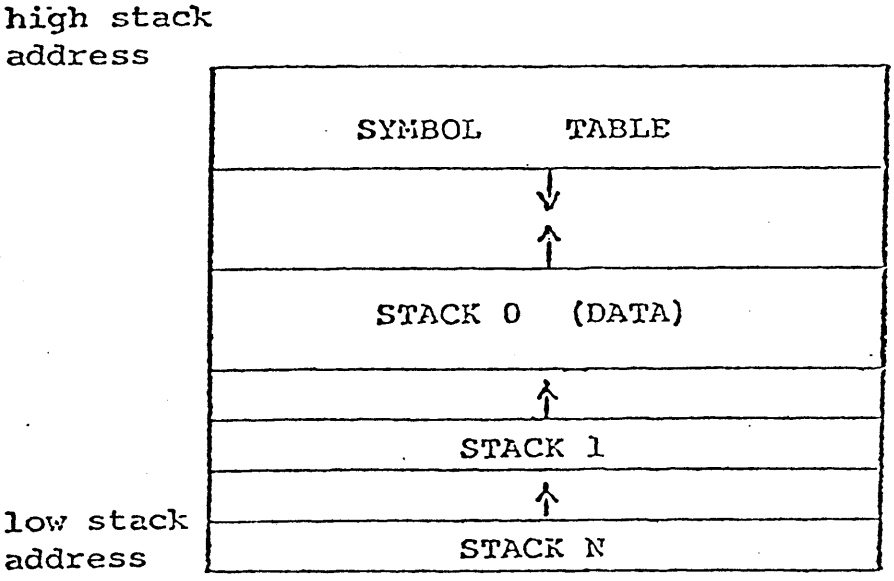


Figure 3-3. STACK ALLOCATION

The Symbol Table, shown in Figure 3-3, grows down from the top of stack memory; the other stacks grow up from the bottom of memory. Stacks can be dynamically rearranged (squeeze) when more space is required for an individual stack. If this operation does not provide enough space, then the Data Stack, ~~is spilled to disk.~~ Symbol Table, and Procedure Stack can be spilled to disk, so that these three stacks can have "virtual memory".

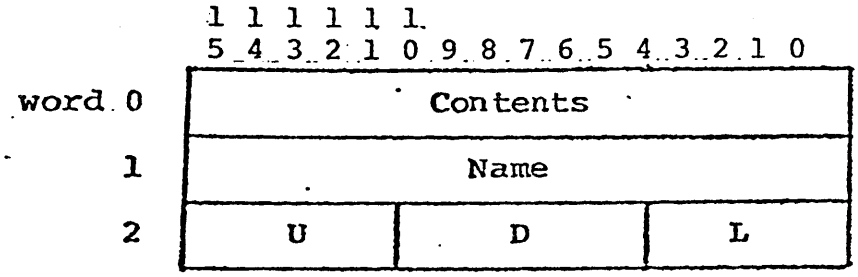
some type of register management?

3.3 Stack Descriptions

3.3.1 ACC Stack (Stack 9, Phase 6)

The ACC stack contains information for each binary pseudo-register whose contents are recognized by code gen phase for optimization.

There are eight groups on the stack. Each pseudo-register is represented by the corresponding numbered group.



Contents - a pointer to the item which is in the register. In most cases, the 'contents' pointer is either a Data or Triad stack pointer.

Name - contains the stored pointer of an assignment.

U = Used count. The number of times the value in the register is used.

D = decimal position

L = length

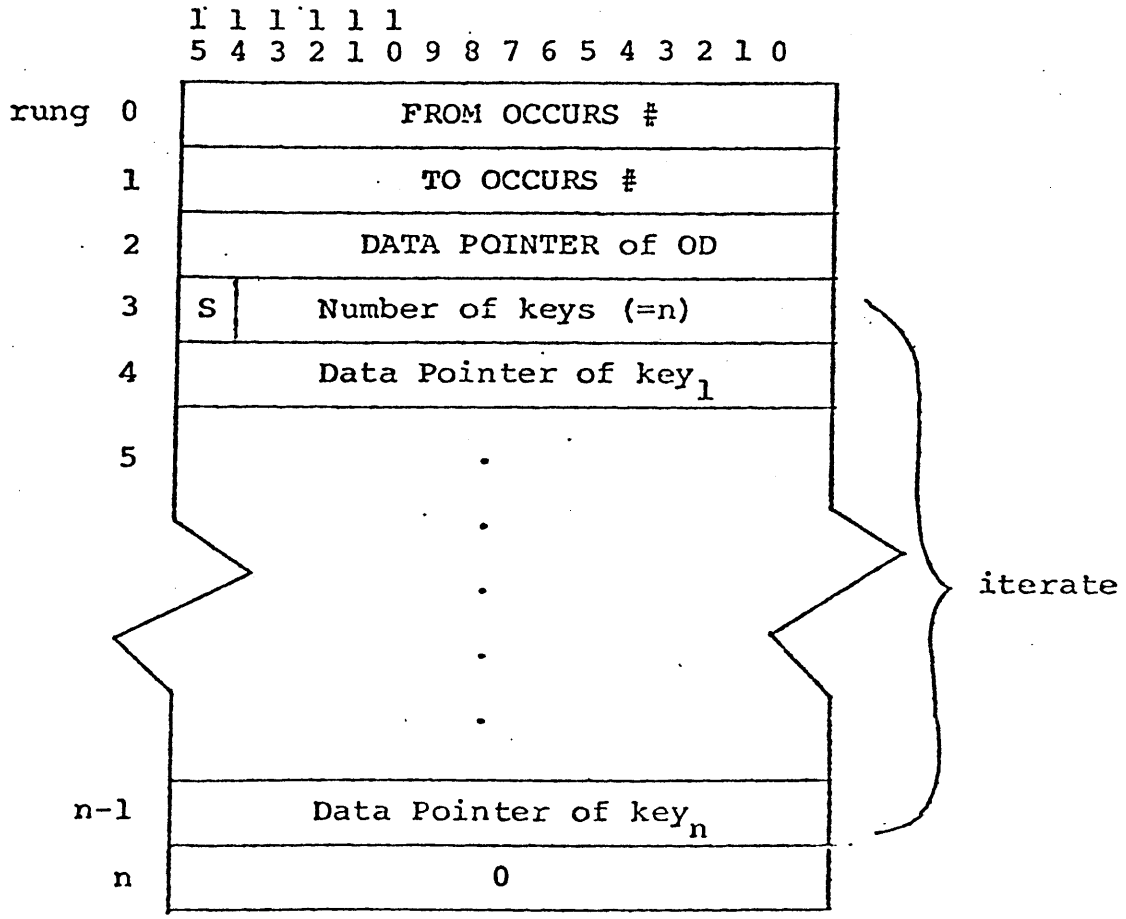
For example, A = B + C is recorded in the stack as follows:

Triad pointer of B + C in 'contents'

Data pointer of A in 'Name'

3.3.2 Array Stack (Stack 5, Phase 1, 2, 3, 6)

The Array stack is used to collect the OCCURS clause information. It is built along with the Data stack for table items and is accessed via the Data Stack.



Where FROM occurs # = minimum occurrence number

TO occurs # = maximum occurrence number

= 0 if no TO option

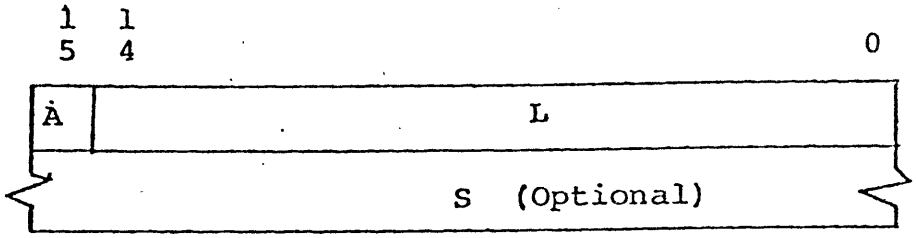
Data Pointer of OD = DEPENDING ON data item pointer ∞ 0

S = 1 ASCENDING

= 0 DESCENDING

3.3.3 Copy Replacing Stack

The Copy Replacing Stack is used as a temporary hold area for strings to be replaced and the replacing strings. As each word in the source stream is encountered, it is compared to all of the "replaces" stored in the stack. When a match is made the "replacer" is substituted in the source input text. The Copy Replacing Stack is built and used in the first pass.

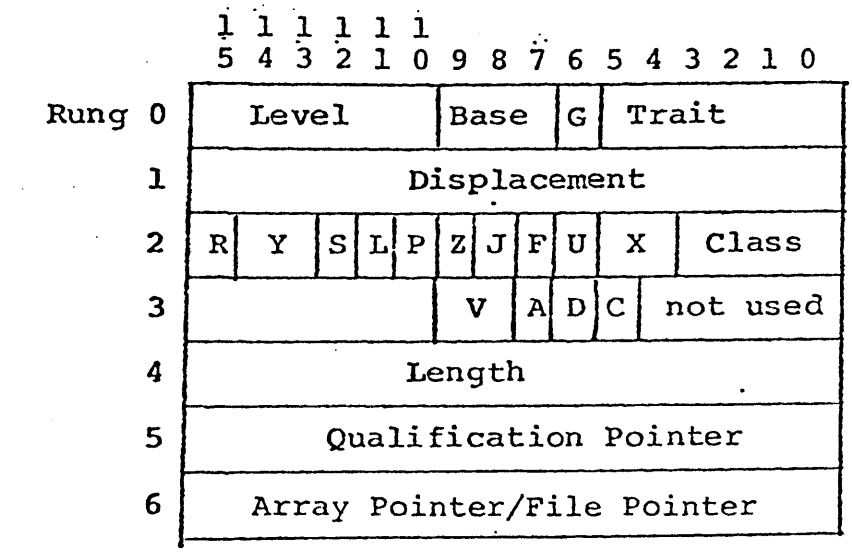


Where A = 0 REPLACING string
= 1 BY string
L = Byte size of S
S = String (up to 256 bytes)

3.3.4 Data Stack (Stack 0, Phases 1-7)

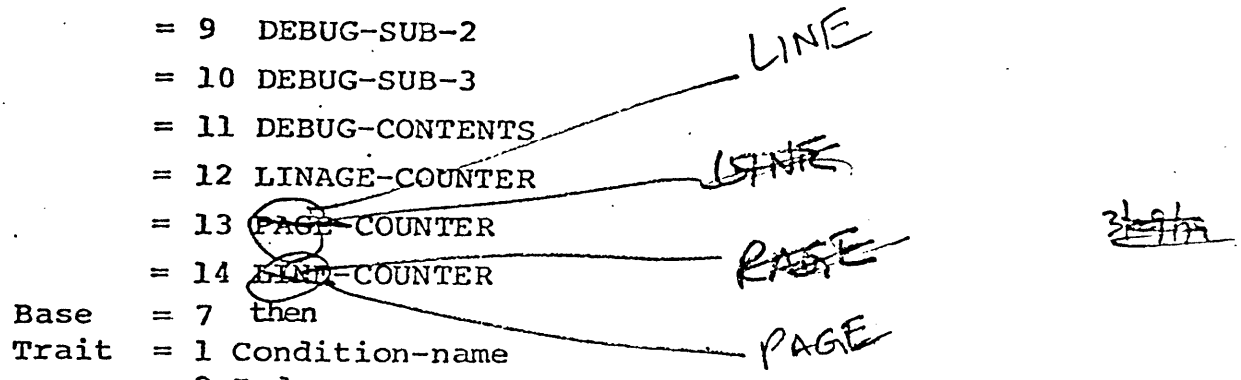
This stack records the definition of each data-name/condition-name/index-name in the order of COBOL source presentation. The stack is used by all phases and is likely to have the highest activity. In addition to named data-names, the FILLER items described with 01 level, OCCURS and/or VALUE clause are also registered. Group slot 0 is physically not used for this stack only!! Group 1 is the first real entry.

3.3.4.1 Data-name, parse



Level	= 0	level 77
	= 1-49	level 1-49
	= 50	level 66 (RENAMES)
	= 51	RENAMES edited item
Base	= 0	Linkage Section
	= 1	File Section
	= 2	Working-Storage Section
	= 5	Report Section
	= 6	Compiler-generated registers
	= 7	Condition-name/Index-name
G	= 1	Procedure-name reference

Trait = 0 not used
= 1 DATE
= 2 DAY
= 3 TIME
= 4 PRINT-SWITCH
= 5 DEBUG-ITEM
= 6 DEBUG-LINE
= 7 DEBUG-NAME
= 8 DEBUG-SUB-1
= 9 DEBUG-SUB-2
= 10 DEBUG-SUB-3
= 11 DEBUG-CONTENTS
= 12 LINAGE-COUNTER
= 13 PAGE-COUNTER
= 14 LINE-COUNTER
Base = 7 then
Trait = 1 Condition-name
= 2 Index-name
= 5 Condition-switch



Displacement =

- before the record resolution
  a. if R = 1, redefined Data pointer
  b. if Level = 50, Renamed Data pointer
  c. for others, not used
- after the record resolution
  a. if Base = 0 (Linkage Section), contains the byte displacement of the item from the level 01 item
  b. if Base = 1 (File Section), byte displacement from the file base
  c. if Base = 2 (Working-Storage Section), byte displacement from the working-storage base.

R = redefined flag in phase 1
= debugging item flag in phase 3
Y = 0 no SYNCHRONIZED
= 1 SYNC RIGHT
= 2 SYNC LEFT

S = ~~signed~~ 1 signed

L = 0 sign is TRAILING

= 1 sign is LEADING

P = 0 sign is not separate

= 1 sign is SEPARATE

Z = 1 BLANK WHEN ZERO

J = 1 JUSTIFIED RIGHT

F = 1 FILLER item

U = 1 USAGE specified

X = subscript level, $0 \leq X \leq 3$

Class = 0 group

= 1 alphabetic

= 2 alphanumeric

= 3 numeric (DISPLAY)

= 4 packed decimal (COMP-3)

= 5 binary (COMP & COMP-4)

= 6 index (INDEX)

= 10 alphabetic edited

= 11 alphanumeric edited

= 12 numeric edited

V = Value class

0 Value not specified

1 not used

2 alphanumeric

3 numeric

A = Variable length

D = OCCURS DEPENDING item

C = PICTURE clause specified

Length =

- for variable group items, it contains the maximum group length
- for numeric items,
  - Bits 15-10 contains the number of decimal positions.
    $-18 \leq$ Decimal $\leq +18$ when Decimal is <0, the assumed decimal position is $|D|$ digits to the right of the item.
  - Bits 4-0 contains the logical length. Maximum is 18.
- for non-numeric items, it contains the logical length. Maximum is 32K characters.

Qualification pointer =

- for qualified data items, it contains the Data pointer of a synonym item that precedes the item
- for all others, it contains zero

Array Pointer/File Pointer

- for OCCURS item, it contains the Array pointer.
- for Level 01 items in a file record, it contains the file pointer.

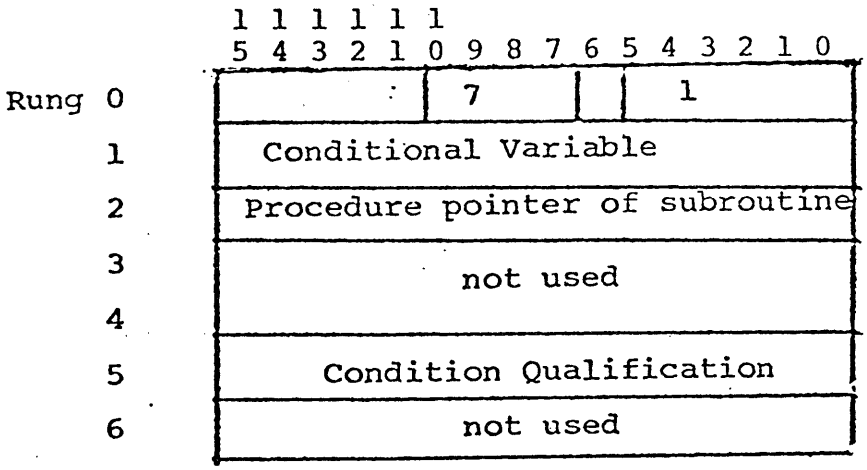
3.3.4.2 Data-name, after allocate phase

Rung 0	Same as Parse
1	Location
2	Same as Parse
3	DD Location
4	Same as Parse
5	Mask Location
6	Same as Parse

where Location is

- byte address of the item if not in the Linkage section

3.3.4.3 Condition-name (level 88 item):



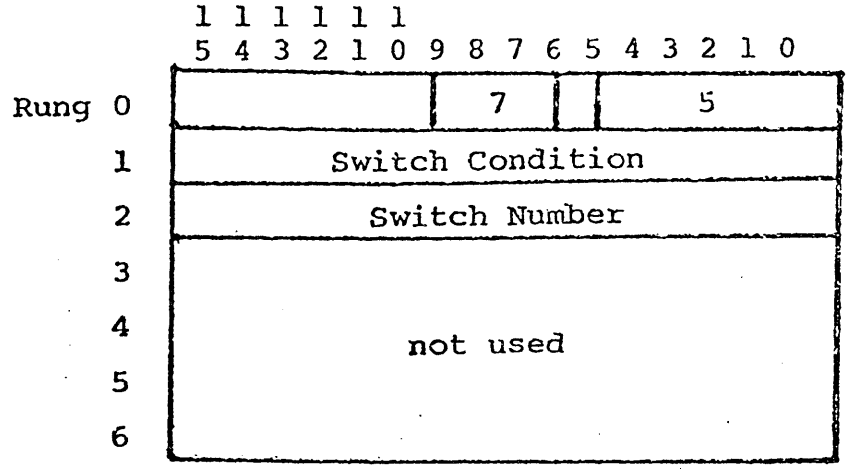
where

Conditional Variable = Data stack pointer of conditional variable item.

Condition Qualification = Data stack pointer of condition-name qualification chain.

Procedure pointer of subroutine = compiler-defined procedure pointer to the condition closed subroutine.

3.3.4.4 Condition-name switch status:



where

Switch Condition

= 0 , OFF status

= 1 , ON status

Switch Number

= 1 ~~SW1~~ SWITCH-1

= 2 ~~SW2~~ SWITCH-2

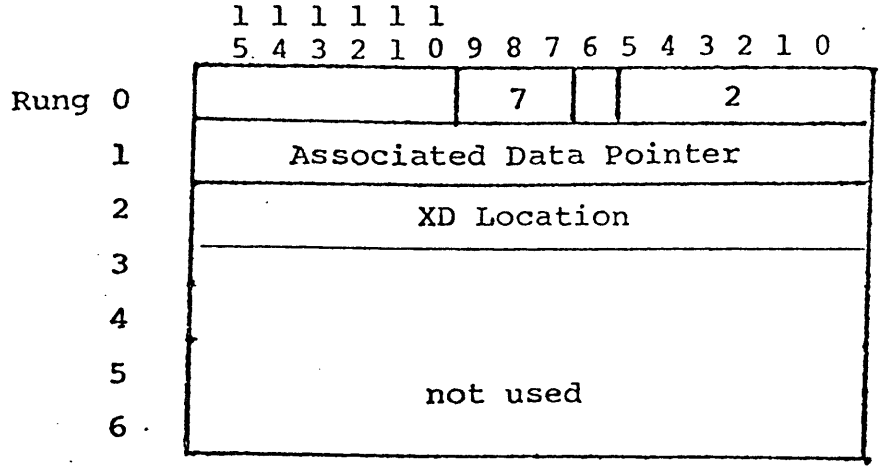
.

.

.

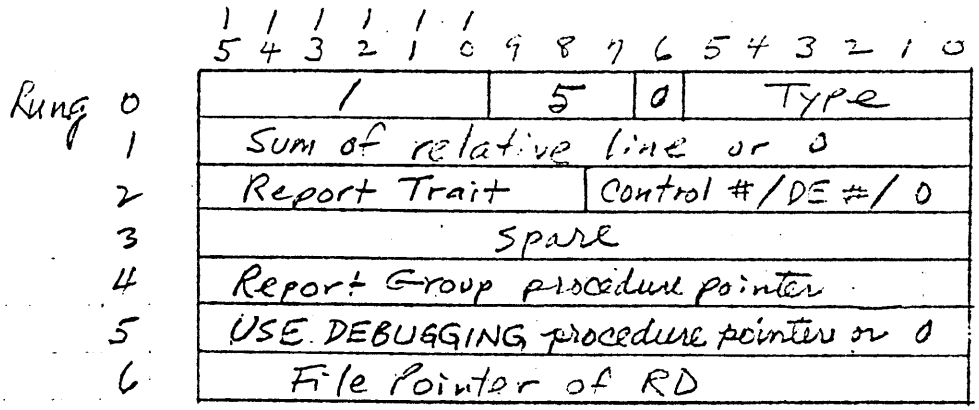
= 16 ~~SW16~~ SWITCH-16

3.3.4.5 Index-name (INDEXED BY)



3.3.4.6 Report Writer Items

3.3.4.6.1 Report Group (01 Level in Report Section)



Type = X'00' DE (Detail)
X'01' RF (Report Footing)
X'02' PF (Page Footing)
X'04' CF (Control Footing)
X'08' CH (Control Heading)
X'10' PH (Page Heading)
X'20' RH (Report Heading)

Sum of relative line

= 0 absolute LINE NUMBER is specified in the report group

≠ 0 sum of relative line number specified in the report group. (i.e., sum of PLUS LINE integers.) The number is used by report writer routines to perform the report group fit test.

Report Trait

Bit 15 = CODE
Bit 14 = CONTROL
Bit 13 = PAGE
Bit 12 = TYPE
Bit 11 = NEXT GROUP
Bit 10 = LINE

Control #

When TYPE = CH or CF, the CONTROL numbers are sequentially assigned from major to minor. (i.e., FINAL will always be 0 if specified.)

DE #

When TYPE = DE, the number associated with each DEs is assigned in the order of presentation. DE # > 0.

USE REPORTING procedure pointer

Procedure pointer to USE FOR REPORTING declarative section

Report Group procedure pointer

Procedure pointer of compiler-generated subroutine for the report group that performs the moves and writes as implied by VALUE, SOURCE, COLUMN and LINE clauses.

3.3.4.6.2 Non Report Group

- Other report writer levels (not level 01)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rung 0	Level						5			1	0					
1	Column number															
2	L	0		S	0		Z	J	0	U	0			Class		
3	Decimal						Report Trait									
4	Length															
5	Qualification pointer															
6	Data pointer of report group															

Report trait

Bit 9 = VALUE
8 = PICTURE
7 = JUSTIFIED
6 = BLANK WHEN ZERO
5 = COLUMN
4 = GROUP INDICATE
3 = SOURCE
2 = SUM
1 = RESET
0 = UPON

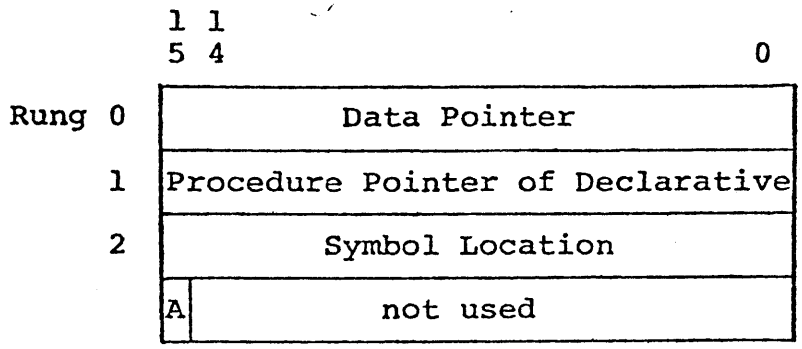
3.3.4.6.3 Control Save Item - Report Writer

The data stack group of CONTROL item is copied to the control save group and a new data location is assigned.

Rung 0 1 2 3	Same as CONTROL item
4	DD Location
5	Same as CONTROL item
6	Procedure pointer to RESET or 0

3.3.5 Data Debug Stack (Stack 8, Phase 5)

In ~~allocate~~ optimization phase, the symbol string of all identifiers (data-name and its qualifier) which are specified in the USE FOR DEBUGGING statements are allocated. The stack is used to collect the addresses of these symbol strings.



where A = ALL REFERENCES

3.3.6 Data Line Number Stack (Stack 9, Phase 1)

The Data Line Number Stack is used to temporarily hold the line number and the column number of each data and filler item described in a given 01 level record. Although the stack is emptied after each level 01 record resolution, a direct correlation exists between this stack and the Data stack.

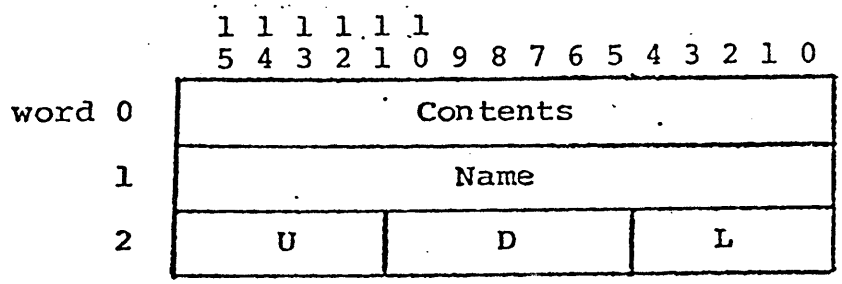
Rung 0	Line Number
1	Symbol table pointer
2	Column Number

register management?

3.3.7 DECA Stack (Stack 8, Phase 6)

The DECA stack contains information for each decimal pseudo-register whose contents are recognized by code gen phase for optimization.

There are eight groups on the stack. Each pseudo-register is represented by the corresponding numbered group.



Contents - a pointer to the item which is in the register. In most cases, the 'contents' pointer is either a Data or Triad stack pointer.

Name - contains the stored pointer of an assignment.

U = Used count. The number of times the value in the register is used.

D = decimal position

L = length

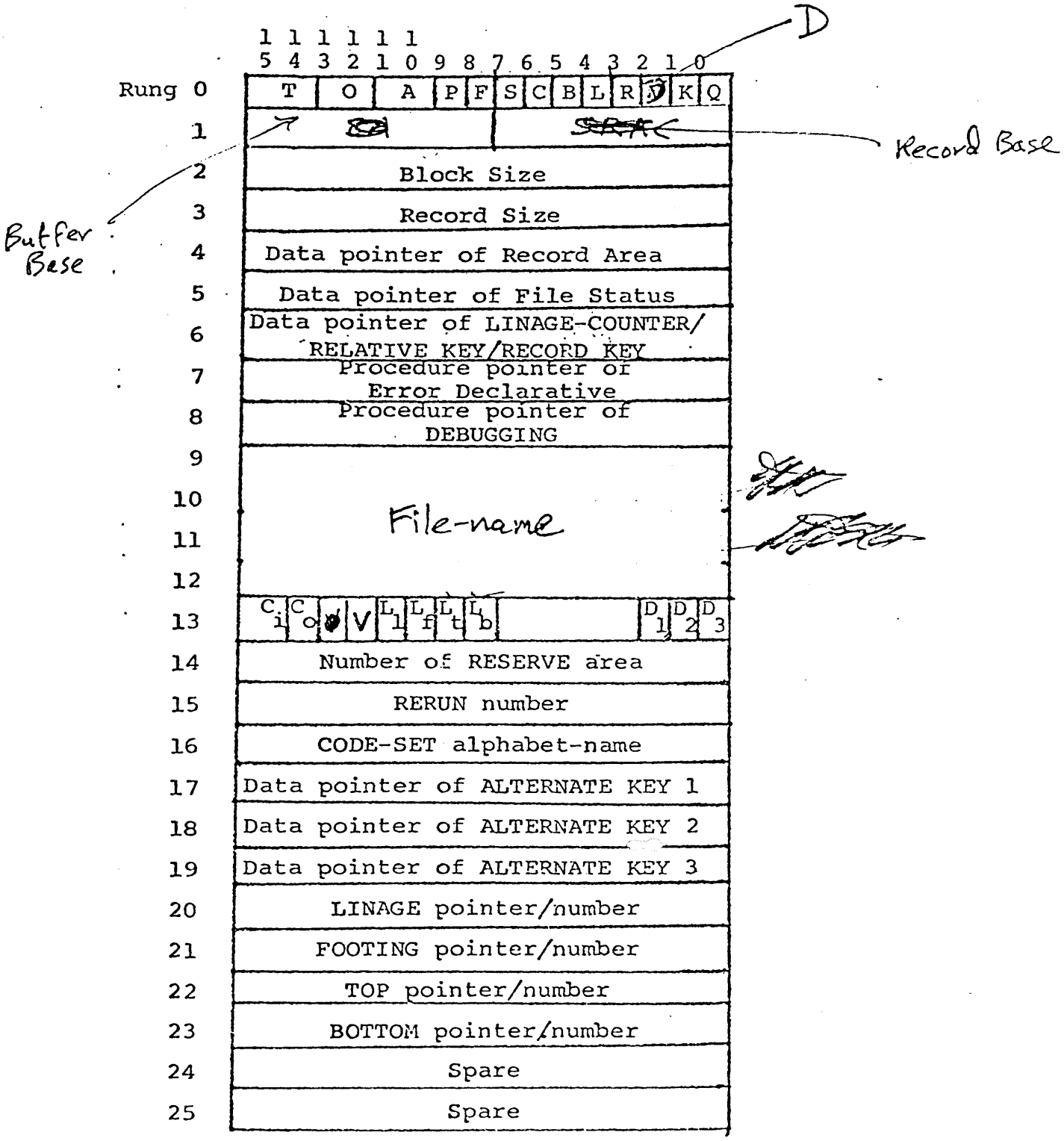
For example, A = B + C is recorded in the stack as follows:

Triad pointer of B + C in 'contents'

Data pointer of A in 'Name'

3.3.8 File Stack (Stack 2, Phases 1-7)

The File Stack is created in parse phase and used by others. It contains all pertinent information relating to the files declared by the user in the source program.



where

T = Type

- 0 FD (File Description)
- 1 SD (Sort Description)
- 2 RD (Report Description)

O = Organization

- 0 Sequential I/O
- 1 Relative I/O
- 2 Indexed I/O

A = Access Mode

- 0 Sequential access
- 1 Random access
- 2 Dynamic access

P = OPTIONAL specified

F = LABEL RECORD

- 0 Omitted
- 1 Standard

S = START specified for the file

C = ADVANCING specified

L = LINAGE specified

R = RERUN specified

D = RESERVE specified (double buffered)

B = Block mode

- 0 for RECORDS
- 1 for CHARACTERS

K = ALTERNATE KEY specified

Q = CODE-SET specified

~~SA = Base Control~~

- 0 Not specified
- 1 SAME AREA
- 2 SAME RECORD AREA
- 3 SAME SORT AREA
- 4 SAME SORT-MERGE AREA

Note

File Name = ~~8~~ 8-character logical name to be used for physical ~~device~~ name connection.

C_i = Opened as INPUT (Sequential I/O only)

C_o = Opened as either OUTPUT or EXTEND (Sequential I/O only)

V = variable length file

L_l = LINAGE Type

- 0 Data pointer
- 1 number

L_f = FOOTING Type

- 0 Data pointer
- 1 number

L_t = TOP Type

- 0 Data pointer
- 1 number

L_b = BOTTOM Type

- 0 Data pointer
- 1 number

D_1 = DUPLICATES for alternate key 1

D_2 = DUPLICATES for alternate key 2

D_3 = DUPLICATES for alternate key 3

3.3.8.1 Report Writer group - RD group

Rung		15 14 13 ... 0
	0	2 | not used
	1	CODE literal value
	2	Data Pointer of Control Save area
	3	Data Pointer of first item of group
	4	Data Pointer of last item of group
	5	Data Pointer of record area
	6	Data Pointer of LINE-COUNTER
	7	Data Pointer of PAGE-COUNTER
	8	File Pointer of FD
	9	# of Controls (CONTROLS ARE)
	10	# of detail groups (DE$)
	11	spare
	12	spare
	13	PAGE-LIMIT #
	14	HEADING #
	15	FIRST DETAIL #
	16	LAST DETAIL #
	17	FOOTING #
	18	RCB Location
	19	Procedure pointer of Control break
	20	Procedure pointer of save move
	21	Data pointer of Control variable
	22	CH Table Location
	23	CF Table Location
	24	Reset Flags Table Location
	25	DE Flags Table Location

3.3.9. File Base Stack (Stack 8, Phase 4)

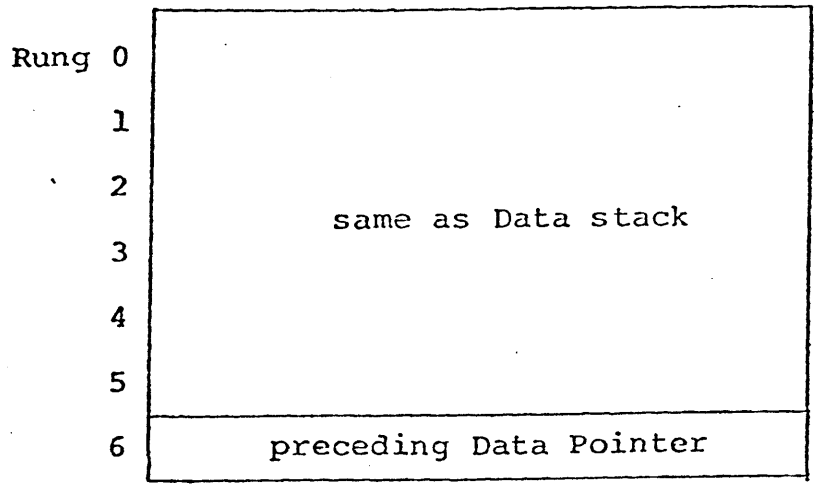
The File Base Stack is used to record the buffer and record size needed for each file base. The files specified in the SAME AREA clause are assigned the same base number. Likewise, the records within the same file are assigned the same record number. The order of the group registration corresponds to the assigned base number of ~~the~~ files.

The stack is exclusively used by allocate phase for allocation. After all files are recorded, a run of the stack is made to allocate the buffer and record areas and to replace contents of rungs with appropriate location value.

Rung 0	Block Size/Location
1	Alt. Block Size/Location
2	Record Size/Location

3.3.10 Filler Stack (Stack 10, Phase 1)

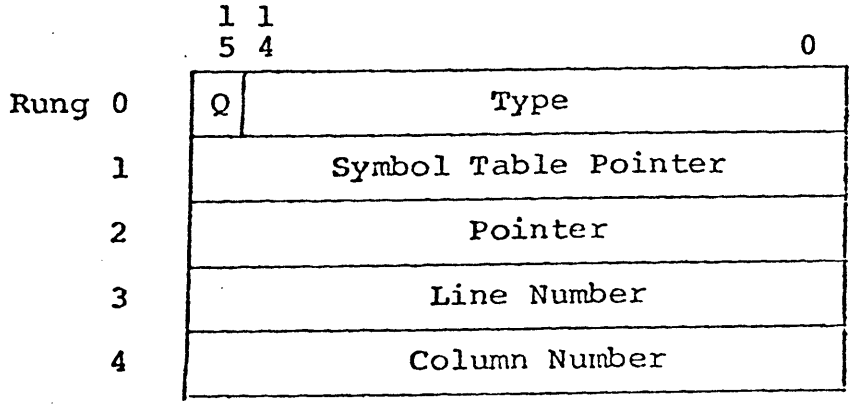
The Filler Stack is used to temporarily accumulate FILLER item information prior to its record group (01 Level) resolution. The format of each group is identical to that of Data Stack except that rung 6 contains a Data stack pointer of the nearest preceding data name. The stack is used only in data parse phase, and after each 01 level record group resolution, it is emptied.



3.3.11 Forward Reference Stack (Stack 7, Phase 1)

The Forward Reference Stack is created during data parse phase to hold the forward data-name reference attribute. At the end of phase 1, a run is made on this stack to

- 1. Update the appropriate stack with data stack pointer
- 2. Diagnose undefined or non unique qualification, etc.
- 3. Output cross-reference records.

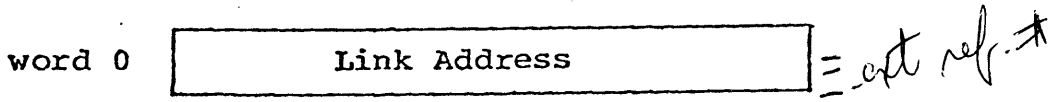


Q = qualifier

Type = 0 FILE STATUS
= 1 spare
= 2 RECORD KEY
= 3 ALTERNATE KEY
= 4 spare
= 5 RELATIVE KEY
= 6 spare
= 7 DATA RECORD
= 8 LINAGE
= 9 FOOTING
= 10 TOP
= 11 BOTTOM
= 12 ASCENDING/DESCENDING KEY
= 13 OCCURS DEPENDING

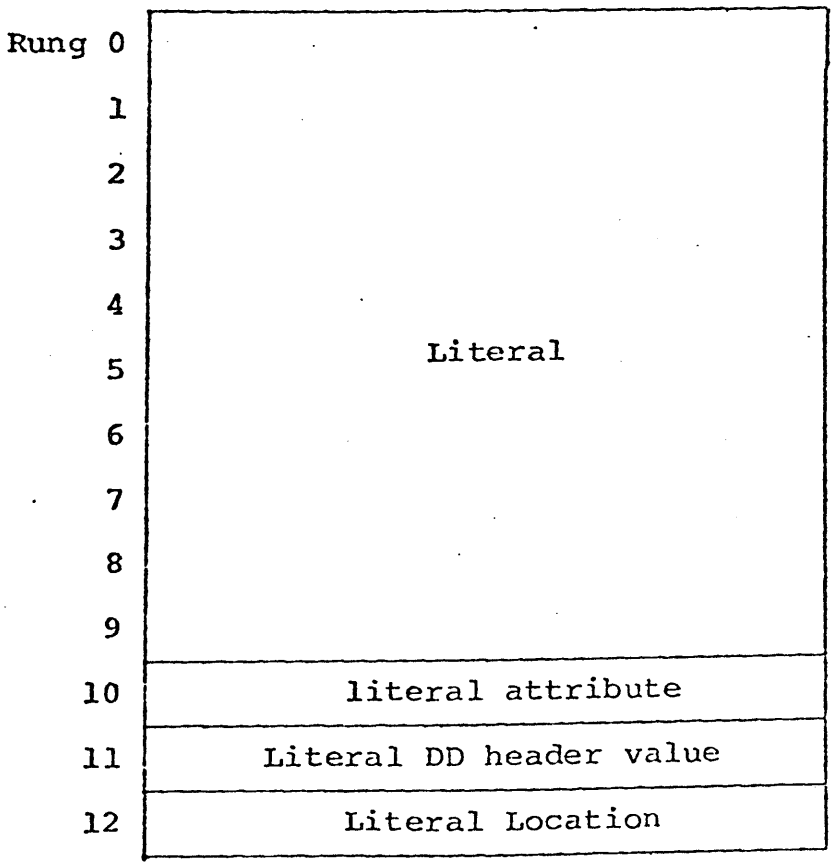
3.3.12 Library Ref Stack (Stack 12, Phase 6)

The Library Ref Stack contains the link addresses of all runtime library routine references that a COBOL object module may require.



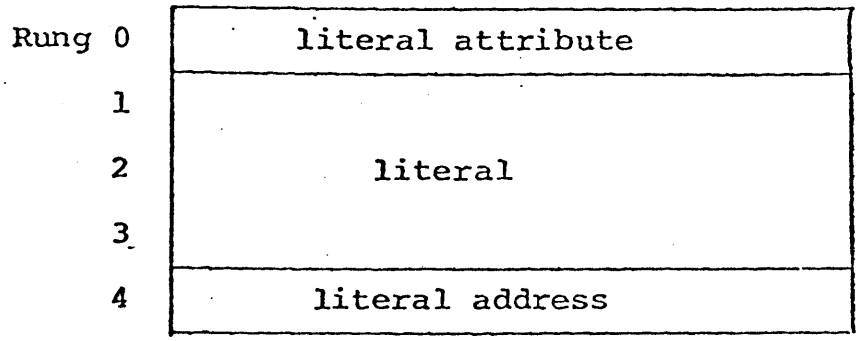
3.3.13 Literal Stack (Stack 11, Phases 4-6)

The Literal stack is used to hold the current batch of edit picture masks and procedure literals. The stack is emptied when processing of each functional block is done.



3.3.14 Literal Optimize Stack (Stack 12, Phases 4, 5)

Literals and edit masks whose length is less than or equal to 6 characters are collected in this stack to eliminate the repetitive generation of literals.



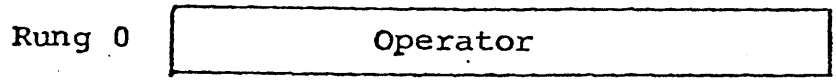
3.3.15 Messenger Stack (Stack # 6, Phases 1, 2, 3, 5)

The Messenger stack is used in parse phases to temporarily hold MS file (Messenger File). At the end of the encoding of each statement, the contents of the Messenger Stack are transferred to the MS file via XEC MOM and the stack is emptied.

Rung 0	Encoded Text

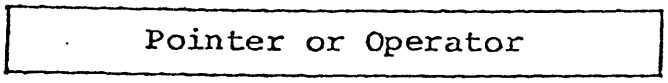
3.3.16 Operator Stack (Stack 14, Phase 2)

The Operator stack is used during the expression analyzer to facilitate the construction of Polish strings.



3.3.1.7 Polish Stack (Stack 11, Phase 2)

The Polish Stack is used for the necessary rearrangement of expressions. It is created and used in procedure parse.



3.3.18 Procedure Stack (Stack 1, Phases 2-7)

The Procedure Stack records the definition and/or references of each procedure-name specified in the Procedure Division.

- in procedure phase

	1 1 1 1 1 1 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
Rung 0	Section Pointer
1	S | | C | B | E | G | A | U | D | Segment #
2	Qualification Pointer
3	Not Used
4	I | O | F | T | Definition Number

Where Section pointer - contains Procedure pointer of section-name in which this paragraph is defined.

S = Section-name

C = Defined in Declarative Section

B = Defined in Debugging Section

E = Procedure-name of PERFORM exit

G = Procedure-name of simple GO TO

A = Referenced by ALTER subject

U = Referenced by USE FOR DEBUGGING

D = Defined

Rung 2 contains a circular synonym chain for qualification

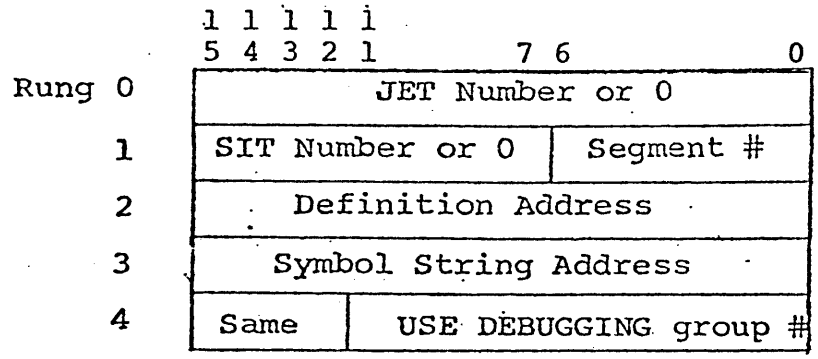
I = Input Procedure of SORT

O = Output Procedure of SORT

F = FROM procedure-name of SORT procedure

T = TO procedure-name of SORT procedure

- After parse



JET number = Index value into the Jump Exit Table, JET

SIT number = Index value into the Segment Interface Table, SIT. The SIT facilitates branching between segments.

For further descriptions of JET and SIT, see Section 5.6, Procedure Code Generation.

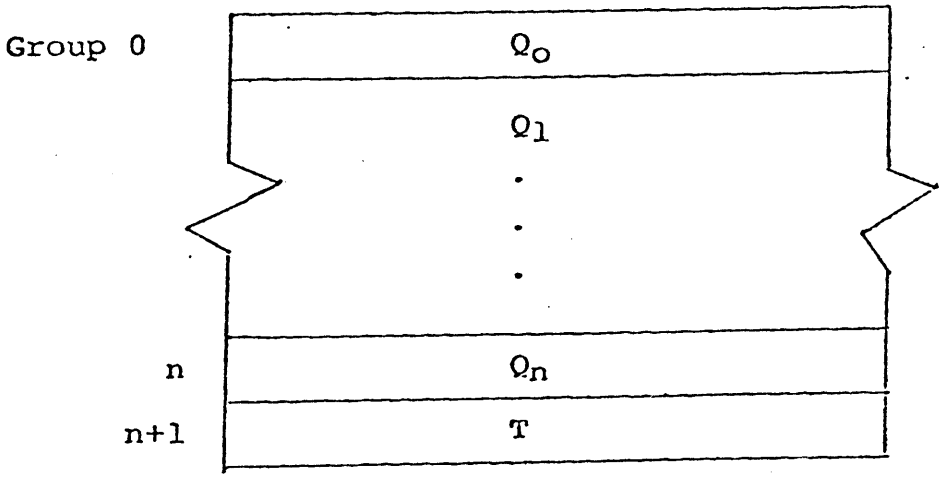
3.3.19 Qualification Stack (Stack 12, Phase# 3)

The Qualification Stack is used during procedure parse phase to hold the order of data-name and its qualifiers. It is used to locate the qualified name in the Data stack. The information in this stack is also used in generating the cross reference record.

The group format of both qualified and qualifier is:

Rung 0	Symbol Table Pointer
1	Line Number
2	Column Number

The order of registration of each group is:



Where Q_o = Qualified group

$Q_1 \sim Q_n$ = Qualifying groups

T = Termination group which contains 0

3.3.20 Renames Stack (Stack 8, Phase 1)

The Renames Stack is used to temporarily hold the Data Stack pointers of "from" and "through" renamed data-name. This stack is used to resolve the displacement of renaming data-name during the 01 Level record resolution. The stack is emptied at the end of each 01 Level record resolution.

Rung 0	FROM Data Pointer
1	THRU Data Pointer or 0

3.3.21 Report Stack

A group is registered in the stack for each record-name specified in the REPORTS ARE clause of FD. This stack is used to make the connection between the file name and its report names. The stack is active during data parse and report writer parse.

Rung 0	File Pointer
1	Symbol Table Displacement
2	Line Number
3	Column Number

3.3.22 Script Stack (Stack 10, Phases 4-6)

The Script Stack is constructed and used in Phase 4. A search of the Script Stack is made for each array reference and if a match is found, the pointer to the matched group is used. Otherwise, a new group is registered onto the stack and its pointer is used.

- Array

Rung 0	Data Pointer of Array
1	Sum of Constant Indexes
2	$Script_1/0$
3	$Script_2/0$
4	$Script_3/0$

- Variable data

Rung 0	0
1	Data Pointer of Variable Item
2	Data/Triad Pointer of OCCURS DEPENDING
3	0
4	0

- Array Condition-name

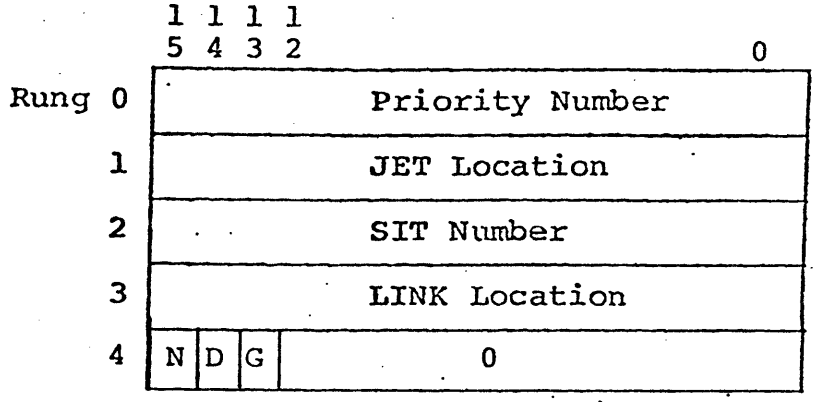
Rung 0	Data Pointer of Condition-name
1	Sum of Constant Indexes
2	$Script_1/0$
3	$Script_2/0$
4	$Script_3/0$

- Variable Group Condition-name

Rung 0	0
1	Data Pointer of Variable Item
2	Data/Triad Pointer of OCCURS DEPENDING
3	Data Pointer of Condition-name
4	0

3.3.23 Segment Stack (Stack

An entry is made to the Segment Stack for each unique priority number defined. The stack is used to hold information for code generation needed in the segment interface.



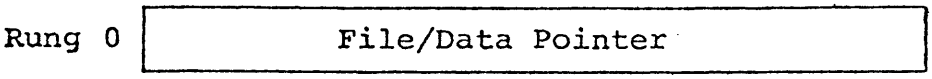
where

N = Noncontiguous segment
D = Defined (used in Collapse phase to detect non-contiguous segments)
G = Generated (used in Code Generation phase)

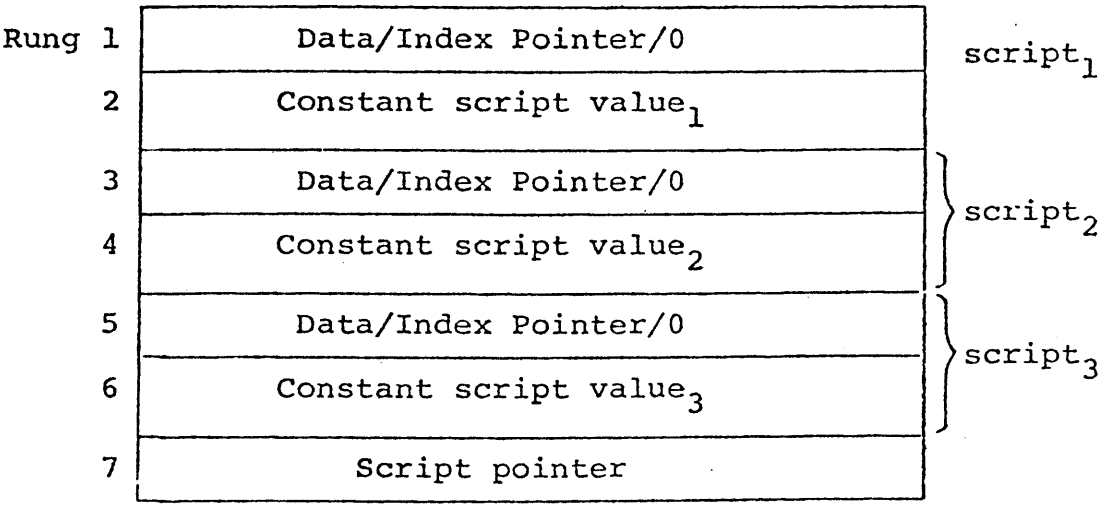
3.3.24 Sta Debug Stack (Stack 3, Phase 5)

The Sta Debug Stack is used to collect debugging information for each statement. At the end of verb generation, debugging codes are produced through this stack.

General format is



If "Data pointer" is an array item, then two additional rungs are pushed onto the stack for each subscript level. In addition, a script pointer is pushed as follows:

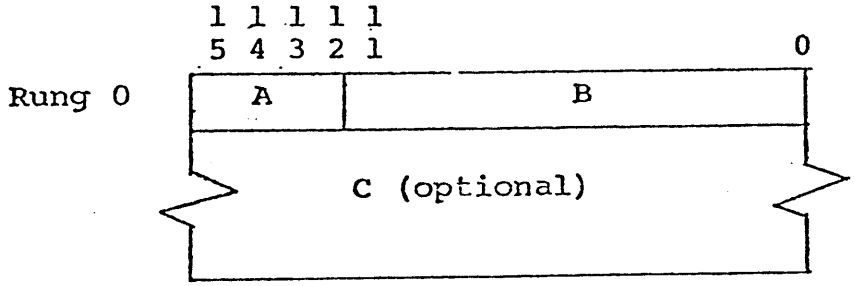


This stack is in plex form. In other words, a pointer to the current top of stack is pushed onto the stack for each stack group and is used to distinguish one statement from another when nested.

intermediate text

3.3.25 Sta Polish Stack

The Sta Polish Stack is produced along with the Triad and Script stacks for each procedural block in optimize phase. At the end of each block, all three stacks are written out to the OP-file as an intermediate text. The code gen phase, in turn, places the contents of the file back into their respective stack area and the code generation for the block is done by driving through this stack.



	A	B	C
0	Data Pointer	Group Number	-
1	Procedure Pointer	" Group Number	-
2	~~Spare~~ File Pointer	" Group Number	-
3	Spare	-	-
4	Spare	-	-
5	Spare	-	-
6	Absolute String	String Length in words	String
7	Spare	-	-
8	Spare	-	-
9	Sta Polish Pointer	Group Number	-
10	Script Pointer	"	-
11	Literal Pointer	"	-
12	Procedure Definition	"	-
13	Statement	Sentence #	Line Number
14	Triad Pointer	Group Number	-
15	Verb	Verb Number	Operands

A = 15, verb number is

0 = end of block
1 = Accept console
2 = Accept date
3 = Accept day
4 = Accept time
5 = Alter
6 = Alter segment
7 = Branch LE
8 = Branch NE
9 = Branch GE
10 = Branch GT
11 = Branch EQ
12 = Branch LT
13 = Report Writer Definition
14 = Binary Compare
15 = Binary Store
16 = Binary Store Index
17 = Call
18 = Class Alpha Test
19 = Class Numeric Test
20 = Close Sequential I/O
21 = Close Relative I/O
22 = Close Indexed I/O
23 = Compare Alpha
24 = Compare Figcon
25 = Compare Group
26 = Compare Numeric
27 = Debug Setup
28 = ~~spare~~ Enter
29 = Display Console
30 = Display ~~Sysout~~ sysout
31 = Delete Relative I/O
32 = Delete Indexed I/O
33 = Test switch-name
34 = Report Writer end
35 = Linage procedure definition
36 = Gc
37 = Go Depending
38 = Go Depending Segment
39 = Go Indirect
40 = Go Segment
41 = Inspect
42 = ~~spare~~ Go initializ
43 = Sort EOF
44 = Link
45 = ~~spare~~ Prepare Data

86 = AND
87 = OR
88 = Convert Binary to Numeric
89 = Convert Binary to Packed
90 = Convert Numeric to Packed
91 = Convert Numeric to Binary
92 = Convert Packed to Numeric
93 = Convert Packed to Binary
94 = Start Relative I/O
95 = Start Indexed I/O
96 = Made Label Definition
97 = Stop
98 = ~~Linage-Procedure Exit~~ Declarative reference
99 = ~~Program Entry~~ Segment Setup
100 = Move Figcon Edited
101 = Return
102 = Search Initialize
103 = Search
104 = Search All
105 = Search All Direction
~~106 = Set~~
~~106~~ 107 = String
~~107~~ 108 = Unstring
~~109~~ = Segment Branch
110 = Exit Program
111 = Unstring Control
112 = Unstring Into
113 = Unstring Into End
114 = Call Variable
115 = Cancel
116 = Merge

106 = String
107 = Unstring
108 = Report Writer Initialize
109 = Report Writer Generate
110 = Report Writer Terminate
111 = Report Writer Write
112 = Report Writer Group Indicate
113 = Segment Branch
114 = Unstring Control
115 = Unstring Initialize
116 = Unstring End
117 = Set
118 = Exhibit
119 = Exit
120 = Trace
121 = Trace Off
122 = Trace On
123 = Declarative Definition
124 = Condition Definition
125 = External Data
126 = Unlock
127 = Inquire
128 = spare
129 = Script
130 = Index
131 = Report Writer Advance

46 = Move Figcon
47 = Move Alpha
48 = Move alpha just-right
49 = Move alpha edited
50 = Move alpha figcon just-right
51 = Move group
52 = Null
53 = Open Sequential I/O
54 = Open Relative I/O
55 = Open Indexed I/O
56 = Perform
57 = Perform Terminate
58 = Read Sequential I/O
59 = Read Relative I/O
60 = Read Indexed I/O
61 = Release
62 = Rewrite Sequential I/O
63 = Rewrite Relative I/O
64 = Rewrite Indexed I/O
65 = Sort Control Block Setup
66 = Set Adjust Index
67 = Set Store
68 = Sort
69 = Store Edited
70 = Set False
71 = Store numeric
72 = Stop Literal
73 = Set True
74 = Size Jump
75 = Size Reset
76 = Debug Procedure Test
77 = Declarative Procedure end
78 = Perform Segment
79 = Write Sequential I/O
80 = Write Relative I/O
81 = Write Indexed I/O
82 = ~~spare~~ Transform
83 = Program Collating Sequence
84 = Test Numeric
85 = Test Binary

3.3.26 Sum Upon Stack

For each item to be summed (addend), in the report writer, a group is registered onto the stack. The Sum Upon stack is used to construct appropriate RESET and SUM routines for each control level in a report.

Rung 0	Stack group type
1	Data Pointer of addend
2	Data Pointer of sum-counter
3	Data ptr of sum-ctr's report group
4	Data Pointer of RESET control
5	Data Pointer of addend report group

where

rung 4 = 0 FINAL

≠ 0 Data pointer of RESET control

rung 5 = 0 addend is not in the report section

Stack group type

= 0 non-array

= 1, 2 or 3 Number of subscripts/indexed needed for addend. The next group in the stack contains subscript/indexes information.

Rung 0	Data Pointer of $subscript_1$
1	Relative Index $integer_1$ or 0
2	Data Pointer of $subscript_2$
3	Relative Index $integer_2$ or 0
4	Data Pointer of $subscript_3$
5	Relative Index $integer_3$ or 0

3.3.27 Temp Stack (Stack 13, Phases 1-7)

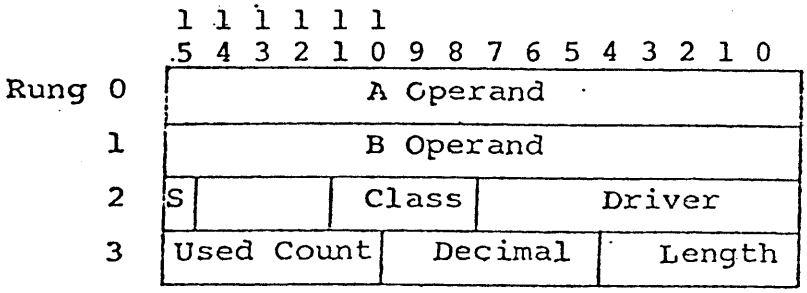
The Temp Stack is used by all phases to hold temporary information.

Rung 0 

3.3.28 Triad Stack

For each collapsible operator, a Triad group is constructed in the Common area and is searched against the Triad Stack. When a match is found, the matched group in the stack is marked as being used one more time. If a match is not found, the group in Common is registered onto the stack as a new entry. In either case, a pointer to the Triad Stack is used to describe the operand in Sta Polish Stack. Collapsible operations are any operations that can be optimized; that is, arithmetic and mode conversions.

In optimize phase, each use of the Triad group is noted on the group by decrementing the 'used'count. As long as the 'used' count is non-zero, the integrity of the Triad group is maintained in either a pseudo register or a temp cell.



A and B operands contain one of following pointers or zero.

- Data
- Index
- Script
- Literal

Two operands of an operation, A and B, are canonized when possible. That is, operands are reordered to some logical sequence so that an operation, X * Y, will match with Y * X operation.

S = Spoiled flag. The operation is "spoiled" when either of the operands is modified.

Class = see Data Stack Description.

Driver = 1 Unary minus, -
= 2 Exponentiation, **
= 3 Multiplication, *
= 4 Division, /
= 5 Addition, +
= 6 Subtraction, -
= 7 Binary load
= 8 Binary load of index-name
= 9 Load
= 10 Load figurative constant
= 11 Round
= 12 SET load (load occurrence number)
= 13 Numeric to packed conversion
= 14 Numeric to binary conversion
= 15 Packed to numeric conversion
= 16 Packed to binary conversion
= 17 Binary to numeric conversion
= 18 Binary to packed conversion
= 19 Binary Load immediate

Used Count = Number of times the Triad result is used.

Decimal = Decimal position of Triad result

Length = Logical length of Triad result.

3.3.29 True Label Stack (Stack 7, Phase 5)
False Label Stack (Stack 15, Phase 5)

These two stacks are used during the logical expression analyze to record branches for relationals.

Rung 0	Sta Polish Pointer
1	Label Code

Where Label Code

= 0 Synonym Spoil

= 1 Synonym Label Start

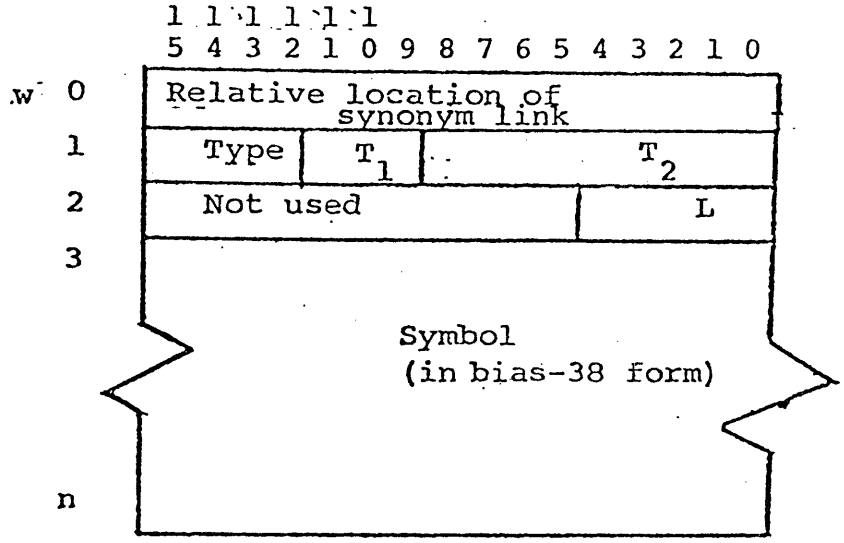
3.3.30 USE Section Stack (Stack 8, Phase 3)

The USE Section stack is used during procedure parse to temporarily hold USE FOR DEBUGGING identifier or USE FOR REPORTING information. At the conclusion of procedure parse, the section pointers held in this stack are transferred to rung 5 of appropriate data stack groups and the stack is popped.

Rung 0	Section Pointer
1	Data Pointer of USE item

3.3.31 Symbol Table

The Symbol Table records the definitions of every unique symbol presented by a COBOL compilation. The implementor-names are predefined and initialized in the table.



For Type = 0 - 14,

Type = stack number

T_1 and T_2 = group number

For Type = 15

T_1 = 0 mnemonic-name, T_2 = file number

T_1 = 1 switch-name, T_2 = switch number

T_1 = 2 device-name, T_2 = device number

T_1 = 3 alphabet-name, T_2 = 0 STANDARD-1

= 1 NATIVE

= 2 EBCDIC

T_1 = 4 language-name $T_2 \geq 3$ literal

L = number of characters in symbol. (Maximum 30 characters)

3.3.32 Reserved Word Table

The reserved word table consists of four distinct tables.

A. Hash Linkage Table

This table contains 128 one-word entries. Each word contains word displacement of a reserved word in the Reserved Word Table (RWT) for the first hash synonym. It contains zero for a null hash synonym. This table is used to reduce the search time required to distinguish user-defined words from reserved words. Each word is accessed via its hash number.

B. Word Number Table

The word number table consists of five entries. Each entry is the number (in alphabetical order) of the first reserved word requiring a given amount of space in the reserved word table. These numbers are compared to QQ and QQA operands to determine the size of the word being asked for.

C. Word Displacement Table

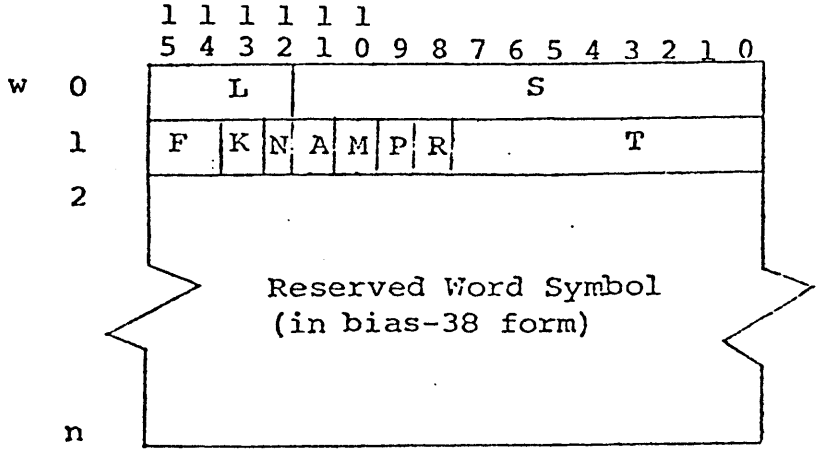
Each word in the five-word table contains the word displacement into the Reserved Word Table (RWT) for the first entry of a given size. It is used to locate and obtain required information of the reserved word in question.

To locate a word in the RWT from a QQ or QQA operand, the following steps are performed:

1. Search Word Number Table for largest entry less than or equal to operand.
2. Subtract WNT entry from operand.
3. Multiply result by RWT entry size (2 + subscript of WNT entry).
4. Add to corresponding WDT entry to get RWT displacement.

D. Reserved Word Table (RWT)

This table contains the information of each reserved word, such as length, forward synonym linkage, reserved word characteristics and packed reserved word symbol. This table is accessed via the Hash Linkage Table or Word Number Table and Word Displacement Table. The format of each entry is:



Where L = character count of reserved word symbol

S = relative halfword location of the next hash synonym in chain. A value of zero signifies the end of chain.

F = Federal Standard level indicator

= 0 low

= 1 low-intermediate

= 2 high-intermediate

= 3 high

K = Keyword. This flag is used when the syntax analyzer is in the recovery mode.

N = Reserved word used as a name (e.g., LINAGE-COUNTER).

A = Area A required.

M = Both area A and area B allowed.

P = Period required after this reserved word.

R = Clause recovery flag.

T = Trait which is division dependent or a pre-assigned value for the reserved word.

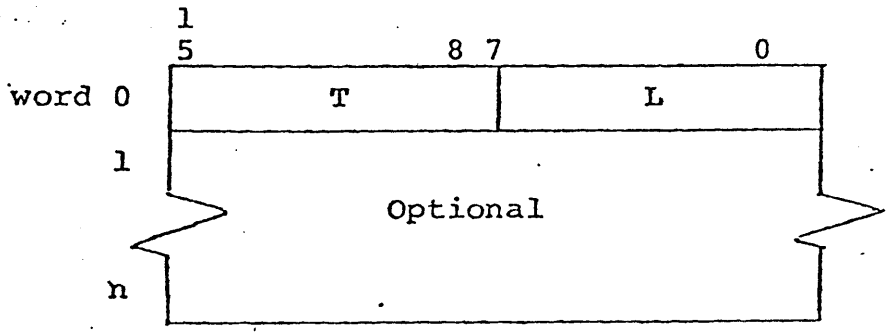
1. For a figurative constant - it contains the actual figurative constant value.
2. For Division, Section, Paragraph and Clause header - the last four bits are "encountered bits" associated with its occurrence.
3. For a verb in the Procedure Division - it contains the "Verb Control" value.
4. For a reserved word used as a name in the Procedure Division - it contains a preassigned number.

3.4 Compile Work File Descriptions

3.4.1 Data Text (DT-Text)

The Data Text is used to record the information in Data Division in order to minimize the size of various stack for those information not directly used during the parsing phases.

The format of the text is:



Where T = Type of the text
L = Word length of B
B = Body of the text (word 1 through word n)

The following is a list of texts being generated.

1. Data Item Header (DIH)

   This text is generated for each data item defined in the Data Division. The DIH precedes any text that pertains to it.

   T = X'00'
   L = n
   w 1 = Data Stack Pointer
   w 2-n = Packed symbol of the data item, if required

2. Line Number (LIN)

T = X'01'

L = 1

w 1 = Source image record number in bits 15-1
'Copied from library' flag in bit 0

3. Initial Value (VAL)

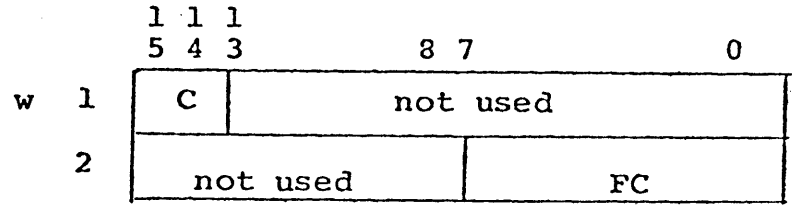
This text is generated for any data item with an initial value specified.

T = X'02'

L = n

w 1 = Literal attribute as follows:

a. Figurative Constant



Where C = 0 for figurative constant

FC = figurative constant value

X'00' = LOW-VALUE

X'20' = SPACE

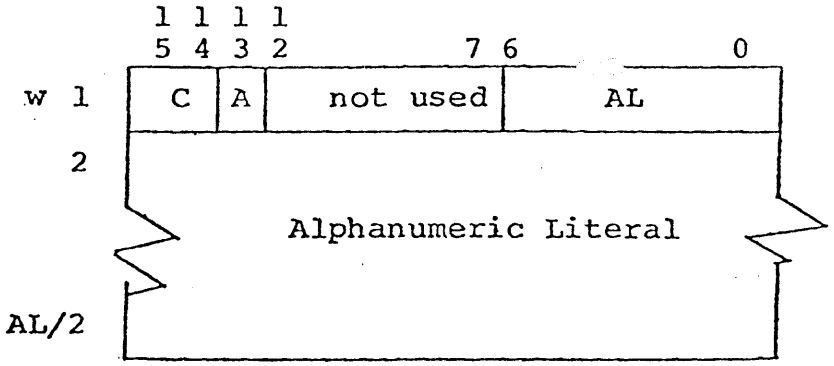
X'30' = ZERO

X'22' = QUOTE - double

X'27' = QUOTE - single

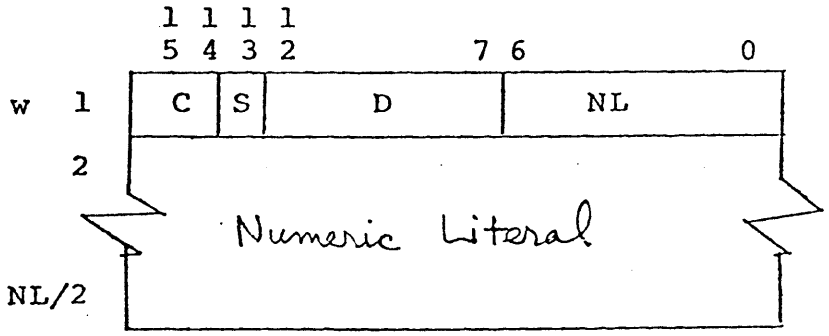
X'FF' = HIGH-VALUE

b. Alphanumeric



Where C = 2 for alphanumeric
A = ALL
AL = alphanumeric literal length ($\leq$120)

c. Numeric



Where C = 3 for numeric
S = negative signed
D = decimal positions
NL = numeric literal length

4. Edit Mask String (EMS)

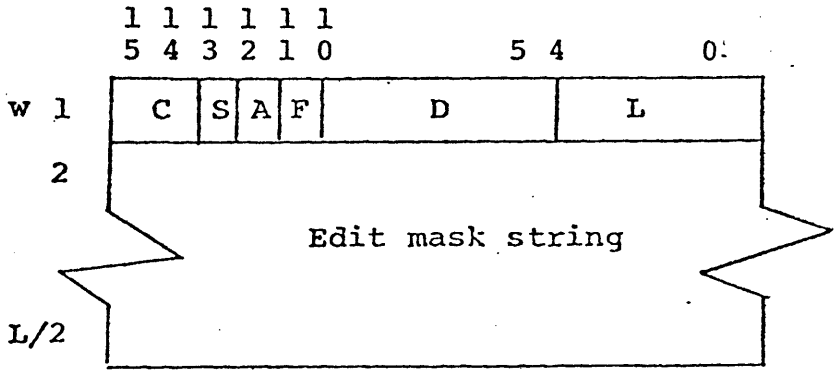
This text is generated for the data item with edit picture.

T = X'03'

L = n

w 1 = Edit mask string attribute

w 2-n = Edit mask string



Where C = class

0 for alphanumeric

1 for numeric

S = digit select not present (i.e. no 9's)

A = asterisk protect

F = floating character present

D = replaceable decimal position

R = number of replaceable positions ($\leq$ 18)

5. Condition Name Header (CNH)

This text is generated for each 88 level item and is followed by condition literal text.

T = X'04'
L = n
w 1 = Condition-Name Stack Pointer
w 2-n contains the packed symbol of the condition-name, if required.

6. Condition Literal Single (CLS)

This text is generated for each condition literal without "THRU" option.

T = X'05'
L = n
w 1 = Literal attribute, same format as VALUE literal
w 2-n = Literal string

7. Condition FROM literal (CFL)

This text is generated for each FROM condition literal:

T = X'06'
L = n
w 1 = Literal attribute
w 2-n = Literal string

8. Condition TO literal (CTL)

This text is generated for each TO condition literal:

T = X'07'

L = n

w 1 = Literal attribute

w 2-n = Literal string

9. Condition Terminator

T = X'08'

L = 0

10. Data Item Terminator

T = X'09'

L = 0

11. Report Name Header

This text is generated to identify RD (report file).

T = X'0A'

L = 1

word 1 = File Pointer

12. Report Name Terminator

T = X'0B'

L = 0

13. Alphabet-name definition

T = X'0C'

L = 0 or 129

w 1 = mnemonic number (see Symbol Table description)

w 2-129 = 256-byte translation table if not 'STANDARD'.

14. Program Collating Sequence

    T = X'0D'

    L = 1

    w 1 = mnemonic

15. Line Number

    T = X'D0'

    L = 1

16. DT-Text Terminator

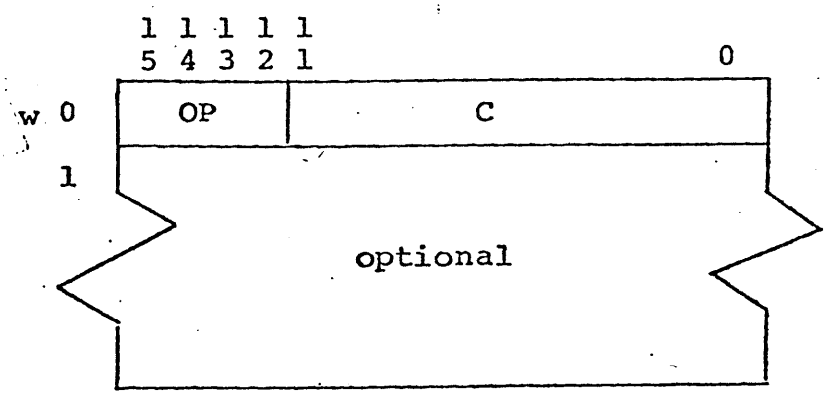
    This text is generated to flag the termination of DT-text

    T = X'FF'

    L = 1

3.4.2 Encoded Procedure Text, EP-Text

The Encoded Procedure Text is a simple encoding of PROCEDURE DIVISION. The format is:



Where OP = 0 Dataname/Condition-name/Index-name
= 1 Procedure-name
= 2 File-name
= 3 Spare
= 4 Spare
= 5 Mnemonic-name
= 6 Miscellaneous
= 7 Qualified Procedure reference
= 8 Expression
= 9 Spare
=10 Compiler-generated tag
=11 Literal or Figurative constant
=12 Procedure Definition
=13 Source Line Number
=14 Procedure Syntax information
=15 COBOL verb

Control field, C, is as follows:

- For OP = 0,1,2,3,4,7,12
  C = group number of stack
- For OP = 5, C = '8nn' where nn is collating sequence table number.
- For OP = 6
  C = 0 for implied subject in an expression
- For OP = 8, C consists of two sub fields; Order and Operator #. The Order field is bits 4-7 and the Operator # field is bits 8-15.

Operator	Order	Operator #
expression end	0	0
unary minus	1	1
**	2	2
*	3	3
/	3	4
+	4	5
-	4	6
>	6	7
=	6	8
<	6	9
condition relational	6	12
numeric test	7	13
alphabetic test	7	14
positive test	7	15
zero test	7	16
negative test	7	17
NOT	8	18
AND	9	19
OR	10	20
expression start	15	0

(4-1)

- For OP = 12

  C = group number of procedure stack
  w1 = Symbolic (ASCII) length of procedure-name being defined
  w2 → n = Symbolic string of procedure-name

- For OP = 13

  C = 0
  w1 = Bits 15-1 contains the source line number
  Bit 0 contains the 'copied' flag

- For OP = 10,
  C = 0 next sentence
  = 1 true label
  = 2 false label

- For OP = 11, C consists of two subfields: C_1 and C_2 where C_1 is bits 4-7 and C_2 is bits 8-15.
  C_1 = 0 literal attribute and literal is words 1-n.
  = 4 integer in word 1 (= 2 alpha literal, = 3 Numeric literal in ASCII)
  = 5 + self-relative integer in word 1
  = 6 - self-relative integer in word 1
  C_2 = word length of optional words (words 1-n)

- For OP = 15, C contains verb number

X'00'	USING (Procedure Division Header)
X'01'	DECLARATIVES
X'02'	USE DEBUGGING
X'03'	USE REPORTING
X'04'	USE STANDARD
X'05'	END DECLARATIVES
X'06'	REPORT NAME HEADER
X'07'	REPORT GROUP HEADER
X'08'	REPORT NAME END
X'09'	~~SPATH~~ spare
X'10'	ACCEPT
X'11'	ADD
X'12'	ALTER
X'13'	CALL
X'15'	CLOSE
X'16'	COMPUTE
X'17'	DELETE
X'19'	DISPLAY
X'1A'	DIVIDE
X'1C'	ENTER

X'1D'	ENTRY
X'1E'	EXAMINE
X'1F'	EXHIBIT
X'20'	EXIT
X'21'	GENERATE
X'22'	GO
X'23'	GOBACK
X'24'	IF
X'25'	INITIATE
X'26'	INSPECT
X'27'	MERGE
X'28'	MOVE
X'29'	MULTIPLY
X'2A'	ON
X'2B'	OPEN
X'2C'	PERFORM
X'2D'	READ
X'2E'	READY TRACE
X'30'	RELEASE
X'31'	RESET TRACE
X'32'	RETURN
X'33'	REWRITE
X'34'	SEARCH
X'37'	SET
X'38'	SORT
X'39'	START
X'3A'	STOP
X'3B'	STRING
X'3C'	SUBTRACT
X'3E'	TERMINATE
X'3F'	TRANSFORM
X'40'	UNSTRING
X'41'	WRITE
X'42'	REPORT WRITE
X'43'	GROUP INDICATE
X'46'	LINE ADVANCE (Report Writer)

X'42' UNLOCK
X'43' INQUIRE

. For OP = 14, this field is a statement option increment.

USE DEBUGGING

X'00'	ALL PROCEDURES
X'01'	ALL REFERENCES

USE STANDARD

X'100'	EXTEND
X'200'	I-O
X'400'	OUTPUT
X'800'	INPUT

ACCEPT

X'01'	DATE
X'02'	DAY
X'04'	TIME
~~X'08'~~	~~SYSIN~~
X'10'	CONSOLE

ADD

X'00'	TO
X'02'	GIVING
X'04'	ROUNDED
X'10'	CORRESPONDING IDENTIFIER
X'40'	ON SIZE ERROR
X'80'	CORRESPONDING

CALL

X'80'	ON OVERFLOW

CLOSE

X'01'	UNIT/REEL
X'02'	WITH LOCK
X'04'	NO REWIND
X'20'	REMOVAL

COMPUTE

X'04'	ROUNDED
X'20'	=
X'40'	ON SIZE ERROR

DISPLAY

X'10'	CONSOLE
~~X'01'~~	~~SYSPUNCH~~
~~X'02'~~	~~SYSOUT~~

DIVIDE

X'00'	INTO
X'01'	BY
X'02'	GIVING
X'04'	ROUNDED
X'08'	REMAINDER
X'40'	ON SIZE ERROR

EXAMINE

X'01'	ALL
X'02'	LEADING
X'04'	FIRST
X'08'	UNTIL FIRST
X'20'	REPLACING

EXHIBIT

X'00'	NAMED
X'01'	CHANGED
X'02'	CHANGED NAMED

EXIT

X'00'	PROGRAM

INQUIRE

X'01'	EXIST
X'02'	KEY
X'04'	OPEN
X'08'	ACCESS
X'10'	ORGANIZATION
X'20'	SHARED
X'40'	LOCK
X'80'	LAST RECORD

INSPECT

X'00'	CHARACTERS
X'01'	ALL
X'02'	LEADING
X'04'	FIRST
X'08'	BEFORE INITIAL
X'10'	AFTER INITIAL
X'20'	REPLACING
X'40'	TALLYING data-name

MERGE

X'00'	ASCENDING KEY
X'01'	DESCENDING KEY
X'04'	USING
X'02'	GIVING
X'20'	COLLATING SEQUENCE (followed by mnemonic-name)
X'40'	OUTPUT PROCEDURE

MOVE

X'10'	CORRESPONDING Identifier
X'80'	CORRESPONDING

MULTIPLY

X'01'	GIVING
X'04'	ROUNDED
X'40'	ON SIZE ERROR

OPEN

X'04'	NO REWIND
X'08'	REVERSED
X'10'	EXTEND
X'20'	I-O
X'40'	OUTPUT
X'80'	INPUT
X'400'	LOCK
X'200'	SHARED

PERFORM

X'00'	FROM
X'01'	BY
X'10'	AFTER
X'20'	UNTIL
X'40'	VARYING

READ

X'04'	NEXT
X'01'	INTO
X'02'	KEY
X'20'	AT END
X'40'	INVALID KEY

X'400' LOCK
X'200' SHARED

RELEASE

X'00'	FROM

RETURN

X'01'	INTO
X'20'	AT END

REWRITE

X'40'	INVALID KEY

X'400' LOCK
~~X'200' SHARED~~

SEARCH

X'00'	VARYING
X'02'	WHEN
X'04'	NEXT SENTENCE
X'10'	Next Sentence Tag
X'20'	AT END
X'40'	ALL

SET

X'00'	TO
X'01'	UP BY
X'02'	DOWN BY

SORT

X'00'	ASCENDING
X'01'	DESCENDING
X'04'	USING
X'02'	GIVING
X'20'	COLLATING SEQUENCE (followed by mnemonic-name)
X'40'	OUTPUT PROCEDURE
X'80'	INPUT PROCEDURE

START

X'02'	USING KEY
X'04'	EQUAL
X'08'	GREATER
X'10'	NOT LESS
X'40'	INVALID KEY

X'400' LOCK
X'200' SHARED

STRING

X'01'	INTO
X'02'	SIZE
X'04'	DELIMITED
X'08'	POINTER
X'80'	ON OVERFLOW

SUBTRACT

X'00'	FROM
X'02'	GIVING
X'04'	ROUNDED
X'10'	CORRESPONDING Identifier
X'40'	ON SIZE ERROR
X'80'	CORRESPONDING

UNLOCK X'02' KEY

UNSTRING

X'01'	INTO
X'02'	ALL
X'04'	DELIMITED
X'08'	POINTER
X'10'	COUNT
X'20'	DELIMITER
X'40'	TALLYING
X'80'	ON OVERFLOW

WRITE

X'20'	EOP, END-OF-PAGE
X'02'	PAGE ~~X'04' POSITIONING~~
X'08'	BEFORE
X'10'	AFTER ~~X'20' POSITIONING~~
X'40'	INVALID KEY

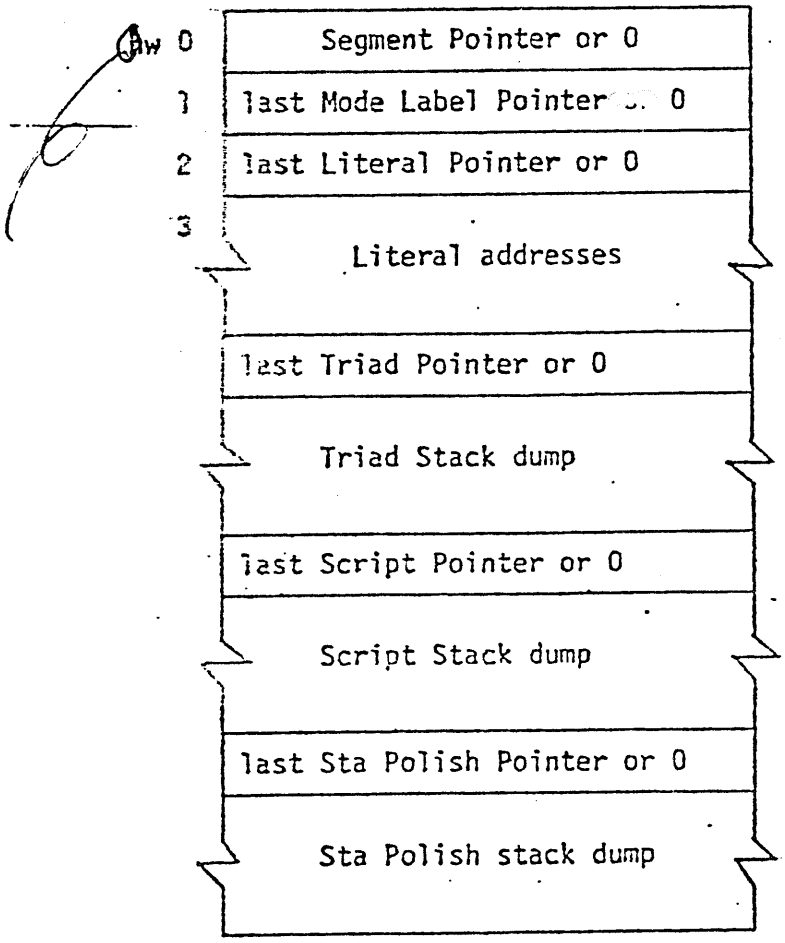
X'400' LOCK

intermediate text file ??

3.4.3 Optimized Procedure Text, OP-Text

This text is a conglomeration of the contents of several stacks used in phase 5 for collapsing and analyzing of EP-Text. Since the collapsing occurs from one procedure-name definition to the next definition (a functional block), information collected in these stacks are no longer needed at the end of each "block". At this time, these stacks are copied out to the OP-file before being popped.

The general format of OP-text is as follows for each block:

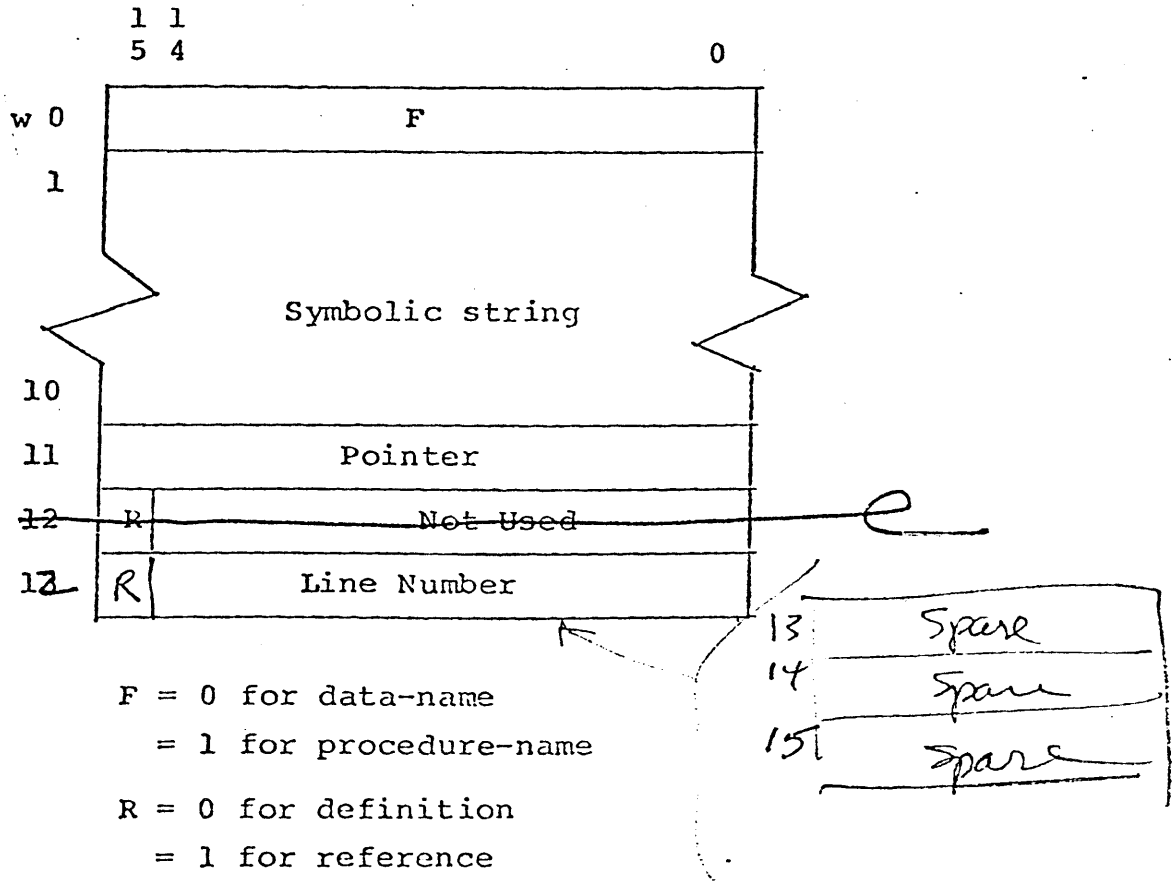


3.4.4 Cross Reference and Diagnostic Text

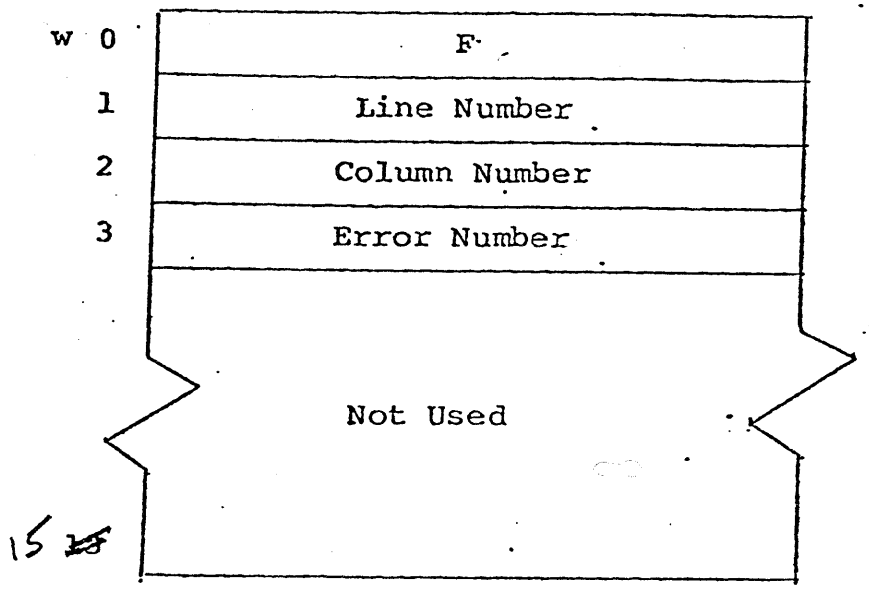
The Cross Reference and Diagnostic Text consists of Cross Reference Records and Diagnostic Records. Each record contains ~~14~~ 16 words and the whole record is used as a sort key.

3.4.4.1 Cross Reference Text, XR-Text

The Cross Reference Record is created by the parsing phases when the CR option is specified on COBOL ~~Job Control Card~~ RUN Command.



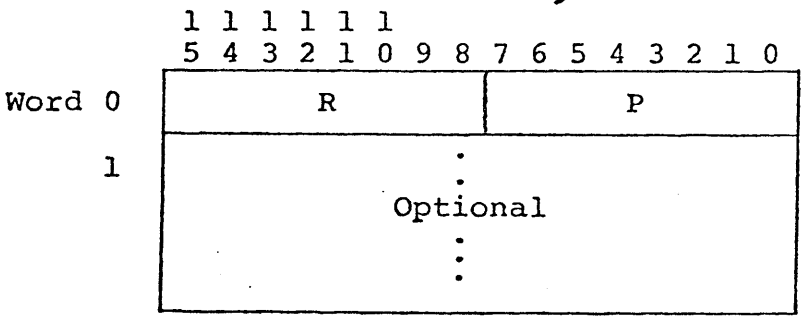
3.4.4.2 Diagnostic Text, ER-Text



Where F = -1 for diagnostic

3.4.5 RW-Text

The RW-Text is identical to DT-Text with following extensions and is used to temporarily hold the report writer information between the passes in Phase 2. The general format is:



1. CODE

   R = X'E0'
   P = 2

   Word 1 = CODE literal value

2. CONTROL

   R = X'E1'

   P = 0 FINAL not specified
     = 1 FINAL specified

   Words 1-n = Data pointers of CONTROL items
               in order of sequence.

3. Lineage

   R = X'E2'
   P = 0 PAGE-LIMIT is specified
     = 1 HEADING is specified
     = 2 FIRST DETAIL is specified
     = 3 LAST DETAIL is specified
     = 4 FOOTING is specified

   The integer value associated with above clauses are placed in the File Stack.

4. TYPE

   R = X'E3'
   P = X'00' DE
     = X'01' RF
     = X'02' PF
     = X'04' CF and word 1 = Data Pointer
     = X'04' + X'40' CF FINAL
     = X'08' CH and word 1 = Data Pointer
     = X'08' + X'40' CH FINAL
     = X'10' PH
     = X'20' RH

5. NEXT GROUP

```
R = X'E4'
P = 1  word 1 contains integer
  = 2  NEXT PAGE
  = 3  integer with NEXT PAGE
  = 4  PLUS and word 1 contains integer
```

6. LINE

```
R = X'E5'
P = 1  word 1 contains integer
  = 2  NEXT PAGE
  = 3  integer with NEXT PAGE
  = 4  PLUS and word 1 contains integer
```

7. COLUMN

```
R = X'E6'
P = not used
word 1 = integer
```

8. GROUP INDICATE

```
R = X'E7'
P = 0
```

9. SOURCE

```
R = X'E8'
P = 0
words 1-n = Data pointer and subscript/index
            information if an array.
```

10. VALUE

```
R = X'E9'
P = 0
words 1-n = value literal
```

11. RESET

```
R = X'EA'
P = 0  word 1 contains Data Pointer
  = 8  FINAL
```

12. Forward Referenced Data Item

```
R = X'70'
P = number of data-names used for the reference
  = 0  for each of qualifying data-names
word 1 contains
 .  if bit 15=0, symbol table displacement
 .  if bit 15=1, compiler-generated register number
    (e.g., PAGE-COUNTER reference is X'800D')
```

APPENDIX D

OPERATIONAL CONSIDERATIONS

Compiler Control Options

Compiler control options are listed on the RN. COBOL line. A comma or space may be used to separate options. A minus sign turns an option off. Compiler control options and their meanings are the following:

Option	Meaning
ANS	Use ANSI mode in the compiler: flag any nonstandard clause or syntax as invalid.
CR	Produce cross-reference listing.
DM	Produce data map.
DQ	Use the double quotation mark (") instead of the apostrophe (') to delimit alphanumeric literals.
DW	Display warning messages.
GO	Generate object code file.
LL	Long listing: use this option for 11-inch paper or 8 lines/inch printing.
LO	List object code. LO may be followed by a specification of lines for which object code is to be listed. This specification takes the form $(R_1, R_2, R_3, R_4, R_5)$. Each R_n may be a single line number or a range lo-hi, where lo $\leq$ hi.
LS	List source code.
R80	Allow 80 columns for source. If R80 is not specified, columns 73-80.

ANS ~~Causes~~ informs the compiler ~~to~~ to ~~issue diagnostics for~~ delete
CR non-~~standard~~ ANSI features and issue
DM diagnostics for them.
DQ
DW
GO
LO ($n_1 - n_2$)
LS
R80
SEG
SUB
SYN
TRACE

Option	Meaning
SEG	Use of the segmentation feature is permitted.
SUB	This is a subroutine.
SYN	Check source program syntax only.
TRACE	Enables trace on and trace off statements.

The options CR, GO, and LS are on unless they are explicitly turned off by a minus sign. All other options are off unless they are explicitly turned on in the RN. COBOL line.

FILE STATUS Data Item

The FILE STATUS data item is a two-character data item which indicates the status of an OPEN, CLOSE, READ, START, WRITE, or REWRITE statement during the execution of the statement and before any applicable USE procedure is executed. The data item has a valid code only if the FILE STATUS clause is specified in the file control entry for a file.

The codes and their meanings are given in table D-1.

3.5 COMPILATION OPTIONS

The compilation options (through $OPTION statement) provide the user a wide range of capabilities.

1	specifies that the source and any accompanying diagnostics are not to be listed.
2	causes the object program to be not written out on the system binary out file.
3	causes a map of the Data Division to be not produced.
4	causes a cross reference list to be not produced.
5	causes the object program to be written out on the system load-and-go file.
6	causes an object listing to be produced.
7	signals the compiler to list warning diagnostics along with the other diagnostics.
8	Source program is checked for syntax only.
9	informs the compiler that the source program has a double quotation mark instead of apostrophes that are to be used as enclosing characters for alphanumeric literals.
10	Federal Information Standard (FIPS) low level diagnosis.
11	Federal Information Standard (FIPS) low-intermediate level diagnosis.
12	Federal Information Standard (FIPS) high-intermediate level diagnosis.
13	causes segment numbers to be ignored so that the object program is not segmented.
14	informs the compiler to produce a subprogram object. This option is required when no argument is being passed to a subprogram.
15	informs the compiler that all 80 columns of input are significant, rather than just the first 72.
16	produces a debug file to be interfaced with the interactive debugger.

word 2 = Line number
word 3 = Column number

For example, a forward reference of A of B will produce following RW-Text:

```
word 0 = X'7002'
  "  1 = Data pointer of A
  "  2 = line #
  "  3 = column #
  "  4 = X'7000'
  "  5 = Data pointer of B
  "  6 = line #
  "  7 = column #
```

3.5 Compilation Options

The compilation options provide the user a wide range of capabilities.

A brief description of each option follows: Any of the options can be negated by preceding the option with a '-' (minus) character, e.g., -LS. If the option is a default value it is underlined.

TEST	produces a debug file to be interfaced with the interactive debugger.
DW	signals the compiler to list warning diagnostics along with the other diagnostics.
DM	causes a map of the Data Division to be produced.
PM	causes a map of the Procedure Division to be produced.
CR	causes a cross reference list to be produced.
LS	specifies that the source and any accompanying diagnostics are to be listed.
LO	causes an object listing to be produced.
DQ	informs the compiler that the source program has double quotation marks instead of single quotation marks that are to be used as enclosing characters for alphanumeric literals.
GO	causes the object program to be written out on the system GO file.
BO	causes the object program to be written out on the system BO file.
SEG	informs the compiler to honor the segment numbers that are associated with each section-name.
SUB	informs the compiler to produce a subprogram object. This option is required when no argument is being passed to a subprogram.

3.6 Compiler Output

3.6.1 Overview

The COBOL compiler optionally produces the following outputs:

- Source listing
- Diagnostics listing
- Binary loader text
- Object listing
- Data name map listing
- Cross Reference listing
- Procedure map listing

3.6.2 Source Listing

The Source listing is produced with the LS option, and each source line consists of a line number, location of generated code, and the 80-column image of a source input. Column 1-6 of each source record is sequence checked and if it is out of sequence, "#" is printed to the left of the source record. If blanks appear in column 1-6, the record is assumed to be in sequence. Sequence checking is done alphanumerically.

3.6.3 Diagnostics Listing

The compiler collects the diagnostics throughout the parsing and code generation phases and produces the list at the end of compilation. Warning diagnostics are inhibited from printing unless specifically requested with the DW option. Critical diagnostics are always produced.

Each diagnostic consists of line and column numbers which pinpoints the position which the diagnostic refers to and a description of the error along with its number.

3.6.4 Object Listing

The Object program listing is invoked with the LO option and appears in the source program listing. The code generated for each source line follows the line and symbolic verb of that line. It consists of a relative location in hexadecimal, a hexadecimal operation code, a relative operand in hexadecimal, a symbolic operation code and symbolic operand. Object listings are descriptive much like an assembly listing.

3.6.5 Dataname Map Listing

The dataname map listing is produced with the DM option and appears after the source object listing in an alphabetical order.

3.6.6 Procedure Map Listing

The procedure map listing is produced with the PM option and appears after the dataname map listing, if specified. It is in an alphabetic order.

3.6.7 Cross Reference Listing

The CR option produces a cross-reference listing of data and procedure names. If DM or PM is specified also, the maps are intermixed with the cross-reference listing.

4. COMPILER PHASE DESCRIPTIONS

4.1 Compiler Organizations

The COBOL Compiler is organized into seven (7) phases which overlay each other and use a common MOM interpreter with Phase Driver as a root phase as shown in Figure 1. The functions of the Root and seven phases are as follows:

Root - MOM interpreter and phase driver
Phase 0 - Compiler initialization
Phase 1 - Identification, Environment and Data Divisions Parse
Phase 2 - Report Writer Parse
Phase 3 - Procedure Division Parse
Phase 4 - Data Allocation
Phase 5 - Procedure Code Optimization
Phase 6 - Procedure Code Generation
Phase 7 - Cross-Reference List

The compiler can also be described as three required "passes" and an optional fourth "pass": (Phase 1 through Phase 3). Pass 1 parses the source text and encodes it for further processing. Pass 2 (Phases 3 and 4) reads the encoded output of Pass 1, allocates data areas and optimizes the procedure code. Pass 3 (Phase 6) generates the object code. Pass 4 (Phase 7) is optional; it produces the cross-reference listing.

4.1.1 Phase 0

Phase 0 performs compiler initialization. It processes the compilation parameters (options specified on the COBOL Job Control card), determines file requirements for pass 1 and available storage for the Symbol Table

and the Stack area. In addition, Phase 0 initializes the Symbol Table with implementor-name symbols and their attributes.

4.1.2 Phase 1

Input: Source of Identification, Environment and Data Divisions

Output: XR-Text and ER-Text of Ex-File

External Output: Source listing of above mentioned divisions.

Phase 1 performs a syntax analysis of the Identification, Environment and Data Divisions. This checking results in the generation of ER-text (error) if user errors are detected. More importantly, this phase creates stack entries for data-names, index-names, file-names and condition-names so that subsequent phases can readily access this information. In addition, the information relating to initial values, edit mask strings and data map symbols is output as DT-text (data clusters) for input to Pass 2. Phase 1 also optionally produces cross-reference information in the form of XR-text.

4.1.3 Phase 2

Input: Source of Report SEction

Output: DT-Text
EP-Text

External Output: Source listing of Report Section

Intermediate: RW-Text

This is an optional phase and is called only when the REPORT SECTION is recognized. Included in Phase 2 are some of Phase 1 parsing routines and a special set of syntax routines to process tne Report Section. This special set of routines is required because of different syntax rules from their standard Data Division counterparts. Phase 2 is comprised of two parts: syntax analysis and encode.

4.1.4 Phase 3

Input: Source of Procedure Division

Output: EP-Text

XR-Text and ER-Text

Phase 3 is similar to Phase 1 except that this phase operates on procedure statements. Phase 3 performs a syntax analysis of the Procedure Division and creates intermediate text called EP-text (encoded procedure). EP-text contains two major categories: procedure-name definitions and verb strings. A procedure name definition element is simply a control number followed by a pointer to the Procedure Stack. Verb strings consist of a verb identifying number followed by arguments that describe verb operands. These arguments may be stack pointers or some syntactical attributes. For example, the statement MOVE A TO B is translated into a verb string containing a MOVE verb number and Data stack pointers of A and B as its arguments.

Phase 3 creates stack entries for procedure-names and, like Phase 1, produces XR- and ER-text. At the end of source input, Phase 4 is called and after this time the symbol table is no longer required.

4.1.5 Phase 4

Input: DT-Text

Output: OP-Text

External Output: Object list for allocated data area
Object for data area

Phase 4 allocates data ~~structures~~ as described in the Data Division; that is, it assigns locations for the data fields defined and generates code necessary for initial values and data section allocation.

In the first part of the data area, all the information that pertains to file description is generated. To do this Phase 4 makes a run on the File Stack. Each file's record area is allocated at this time and the address of the area is recorded in the File Stack. Following record area allocation, a second pass is made through the File Stack to produce File Information tables.

The primary function of FIT is to provide the addresses of abnormal exit points to various COBOL I/O routines. Furthermore, it provides additional information about file's attributes and status; i.e., block size, address of STATUS item, current lock position, etc.

The next step in Phase 4 processing is to make a run on the Data Stack assigning addresses for each data item defined. The order of allocation is same as the order of source presentation except for items which are re-defined or renamed. After the data item allocation, another run is done on the Data Stack and Data Descriptors are produced. Each Data Descriptor (DD) contains the attributes of the data item and the address where it can be found.

4.1.6 Phase 5

Input: EP-Text

Output: OP-Text

External Output: Object list for procedure-reference literals.

Phase 5 carries out the second stage of 3-stage process of converting the COBOL procedural code as described in the Procedure Division into object program. This stage consists of breaking down certain statements into simpler structures that resemble the final object sequence.

The process of optimization in phase 3 is accomplished when re-translating from Polish notation to triad form. The triad form is especially amenable to analysis for the removal of removable operations. As each triplet is constructed, it is compared to triplets already created within the same functional block. When a match is found, the matched triplet is marked as being used once more and the pointer to this triplet is used to describe the latest operation. In this way, all optimizable operations are collapsed into the least number of operations for a given program.

4.1.7 Phase 6

Input: OP-Text

External Output: Object listing of Procedure Division Object program

Phase 6 can be thought of as the generation or assembler phase, because it prepares a machine language program from a pseudo-language text. In this case, the pseudo language is OP-text that was prepared in Phases 4 and 5.

The main function of Phase 6 is to perform the last stage of the translation process for procedure statements:

a. Translate OP-text into an object module suitable for input to the loader.

b. Create separate object files for each segment module.

c. Generate the code necessary for register housekeeping; i.e., generate register stores and loads of intermediate results when required.

d. Produce object listing, if requested. This listing will be a "one-pass" listing, with forward references being resolved by the loader.

The optimization process in phase 6 is accomplished by developing all arithmetic results in a set of pseudo-registers allocated in memory. For each of the pseudo-registers, the contents and their destination are remembered. The algorithm used for selecting the next register takes advantage of a mark left on each triplet indicating the remaining number of times it is to be used by selecting the register with the least number.

4.1.8 Phase 7

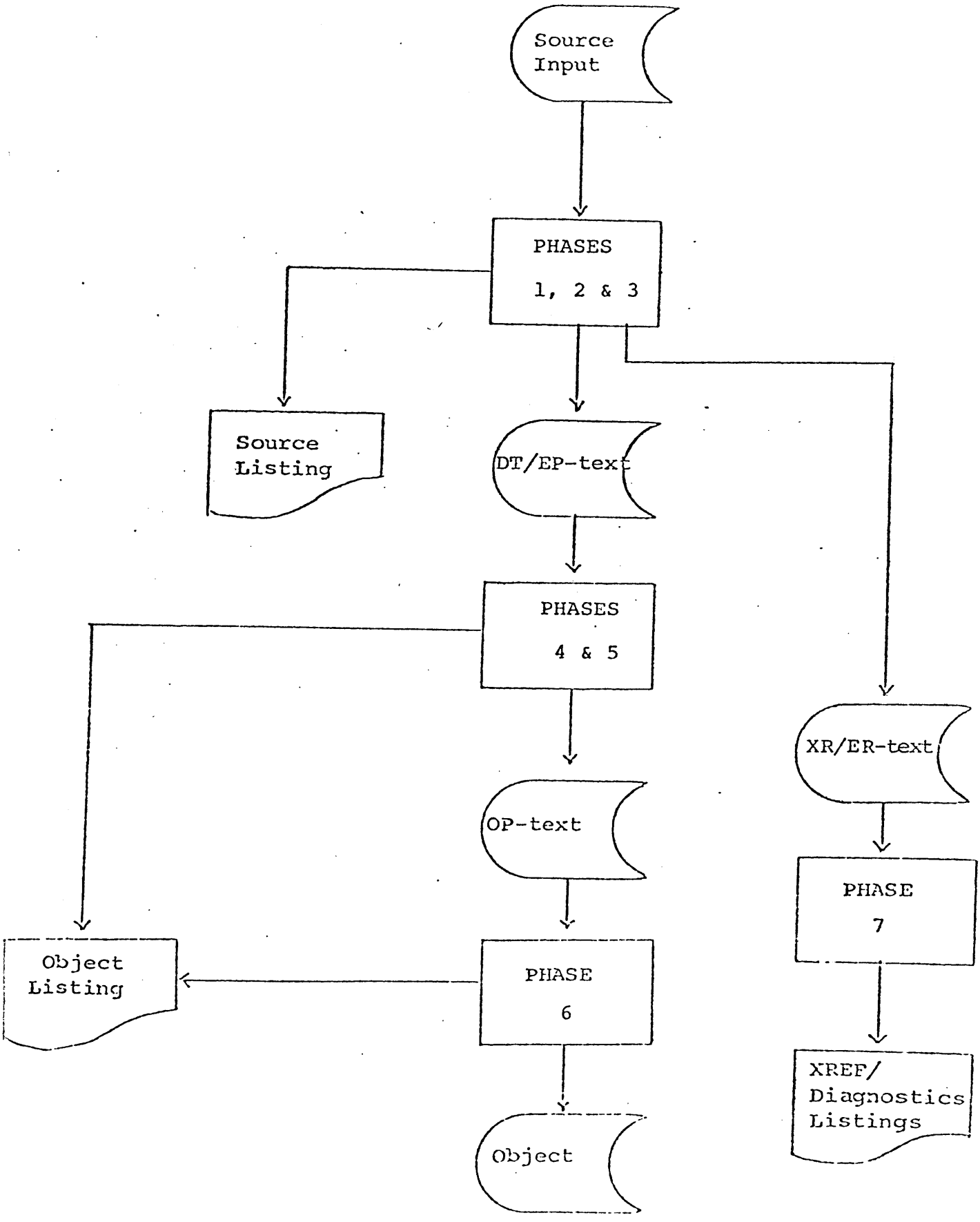
Input: XR-Text and ER-Text of XR-File

External Output: Cross reference listing

The sole function of Phase 7 is to sort the XR-File (XR-Text and ER-Text) and to print out the cross reference and diagnostic listings.

If the CR option is specified, XR-text is generated by Phases 1 through 3. It contains symbolic, name attributes, and references of user-defined names.

4.2 Compiler External Flowchart



5 COMPILER GENERATED OBJECT CODE

5.1 Overview

Due to the nature of the Microdata Express computer, it is not practical to attempt to generate in-line code for the majority of the functions of the COBOL language. The primary reason is that the Microdata Express computer is not a business oriented computer; that is, decimal arithmetic must be done with software. Furthermore, the environment in which the COBOL object must execute in is rather restrictive for good sized COBOL. An average COBOL program is usually in excess of 1000 source lines. Because of this, the design of the Microdata COBOL system includes library routines for performing these functions and compiler generated code consisting of calling sequences to these routines whenever one of the COBOL functions needs to be performed.

5.2 Generation Sequence

5.2.1 Generated order

In the following discussion of the calling sequences for COBOL runtime routines, it is understood that whenever necessary, subscripts, indexes, and data format conversions have been computed, adjusted for type, and placed in the appropriate temporaries or dummies. The code generation for each COBOL verb, in most cases, has the following format:

subscript or index conversion of source	prolog
subscript or index calculation of source	
source data format conversion	
subscript or index conversion of target	
subscript or index calculation of target	
target data format conversion	
COBOL verb processor	
computed data format conversion	epilog

5.2.2 Calling Sequence Conventions

The general calling sequence that the compiler generates for COBOL runtime routines is

```
MARK        "14", routine name
LWL         DD of data-name1
LWL         DD of data-name2
 .
 .
 .
 .
LWL         DD of data-namen
CALL        3+n
```

Since this calling sequence is laborious and unnecessarily redundant to document, a simplified form of describing the calling sequence is used whenever practical.

For example:

```
MARK        "14", C#LOAD
Lr                              where r=DECA number
LWL         DD of id1           source item
CALL        5
MARK        "14", C#STE
Lr
LWL         DD of id2           destination item
CALL        5
```

are documented as

```
LOAD        r,id1
STE         r,id2
```

5.3 External References Naming Conventions

5.3.1 Object Program

The first eight characters of PROGRAM-ID literal are used to identify the root program produced by a COBOL compilation. For the segmented object programs, the last two characters of significant PROGRAM-ID characters are replaced by a segment number. For instance, an object program for the segment 76 of a program called ABCDEF is identified as ABCDEF76.

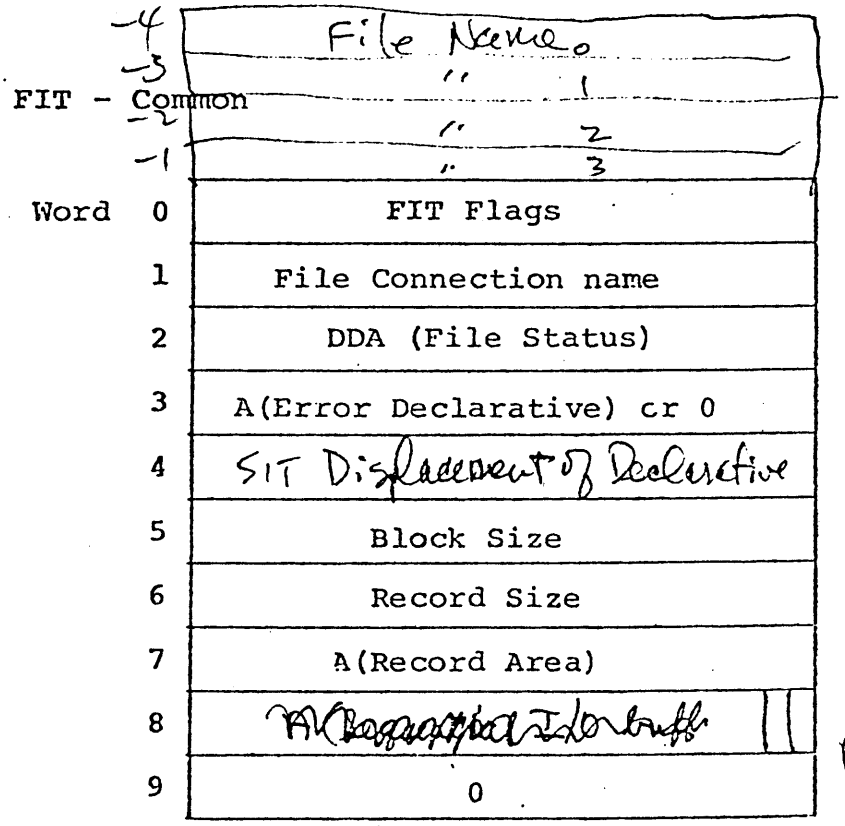
5.3.2 Runtime Library

Each of the COBOL runtime routines is distinguished with C_ prefix. This is done to differentiate COBOL runtime library from other external references which may appear in a load program.

5.4 Generated Data Formats

5.4.1 File Information Table, FIT

The File Information Table, FIT, contains the information necessary to interface COBOL I/O runtime routines. One FIT is produced for each File Description (FD) entry in the source program.



Conventions used are

A( ) ≡ address of

DDA( ) ≡ DD address of

FIT flags are

Bit 15		Current record pointer undefined
Bit 14		Not a read
*Bits 12-13	= 0	SEQUENTIAL access mode
	= 1	RANDOM access mode
	= 2	DYNAMIC access mode
*Bit 11	= 1	SELECT OPTIONAL
Bit 10	= 1	EOF detected
*Bit 9 8	= 1	START specified
Bit 8 9	= 1	Variable length
Bit 7	= 1	LOCK on a close encountered
Bit 6	= 1	Reversed
Bit 5	= 1	first time flag
*Bit 4	= 1	Advancing specified
Bit 3	= 1	Label Declarative specified
Bit 2	= 1	Opened output
Bit 1	= 1	Opened input
Bit 0	= 1	Opened

Bit fields with * are set by COBOL I/O routines.

FIT - Sequential File Extension

Following is continued from common FIT if sequential files.

word c+0	A (Linage Setup Subroutine)/0
c+1	A (Linage Table)/0
C+2	A(Sequential I/O buffer)

If a data-name is specified for any of LINAGE parameters, a subroutine is generated by the compiler to place the binary contents of each data-name in the appropriate entry of the LINAGE table. The LINAGE table is pointed to by word c+1 of the sequential FIT and it is allocated as follows:

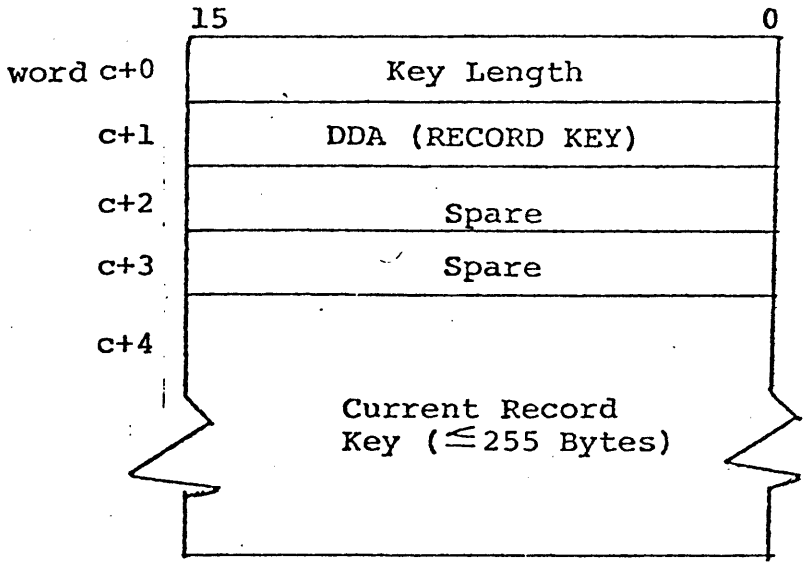
```
77    LINAGE-COUNTER    PIC 9(4)    USAGE COMP-4.
77    LINAGE            PIC 9(4)    USAGE COMP-4.
77    FOOTING           PIC 9(4)    USAGE COMP-4.
77    TOP               PIC 9(4)    USAGE COMP-4.
77    BOTTOM            PIC 9(4)    USAGE COMP-4.
```

If literals are specified for all of LINAGE parameters (FOOTING, TOP, etc.), then the word c+0 of sequential FIT is set to zero by the compiler and the entries in the LINAGE Table are initialized with appropriate literal values.

FIT - Relative File Extension

c+0	DDA (RELATIVE KEY)	in binary form
c+1	0	
c+2 c+3	Current Record Number	

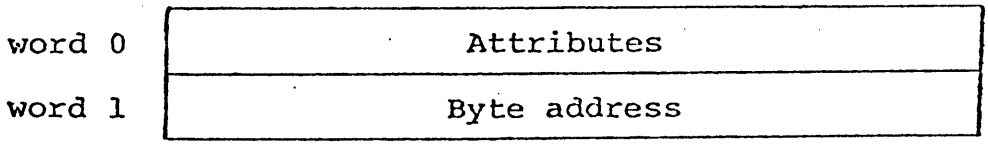
FIT - Indexed File Extension



5.4.2 Data Name Descriptor, DD

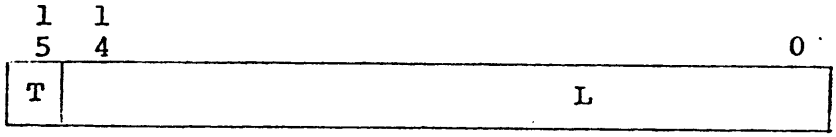
A Dataname Descriptor, DD, is generated by the compiler for each data-name defined in the COBOL source program. Each DD consists of a pair of words; data-name attributes followed by the leftmost address of the item.

The general format of a DD is



The leftmost bit of the attribute word determines whether the DD is of alphanumeric or numeric type.

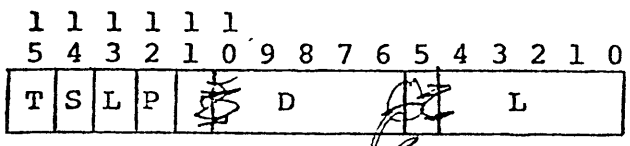
An alphanumeric DD's attribute word is



T = 0 indicating alphanumeric type

L = length in characters. Maximum alphanumeric length is 32767 bytes.

A numeric DD's attribute word is



T = 1 numeric type

S = 0 unsigned

= 1 signed

L = 0 Sign on right (trailing)

= 1 Sign on left (leading)

P = 0 Sign is not separate

= 1 Sign is separate

D = decimal digit count. ($-18 \leq D \leq 18$)

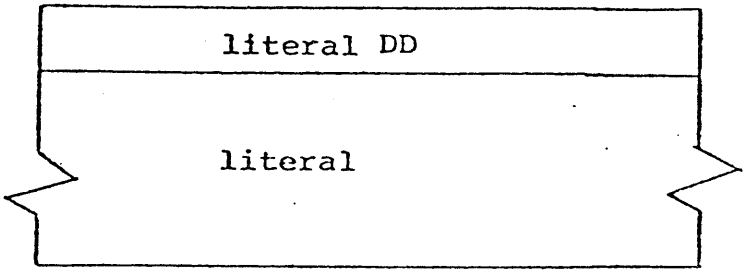
When D is < 0, the assumed decimal point is $|D|$ digits to the right of the item.

e.g. PICTURE 999PP , D = -2, L=3

L = logical digit length. That is, the number of 9's in the PICTURE clause

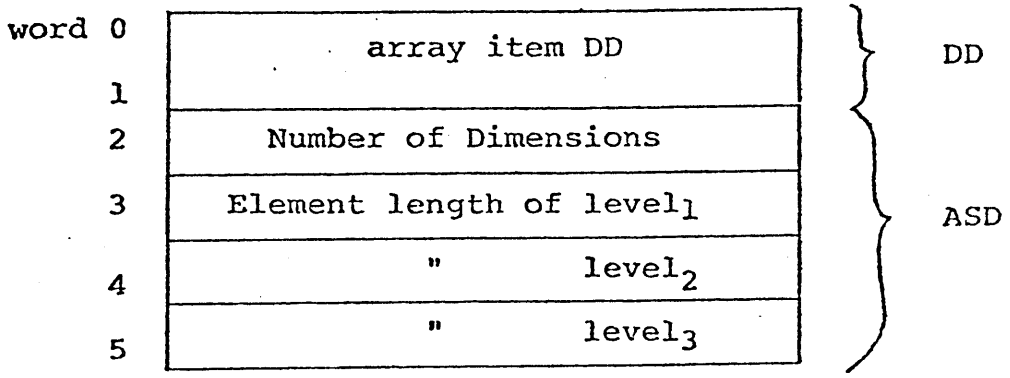
5.4.3 Literals

Each literal being allocated in the data area of the object program is preceded by a DD of the literal. The literal arguments to the runtime routine point to these DD's. Thus, in the runtime, literals are not differentiated from user-defined data-names. The literals are always left justified when allocated.



5.4.4 Array Subscript Descriptor

An Array Subscript Descriptor, ASD, is generated for each array item defined. It contains the number of dimensions and each dimension's element length in memory. The ASD is produced immediately after the array DD to which it pertains.



Where

Number of Dimensions

= 1 one-dimension array

= 2 two-dimension array

= 3 three-dimension array

For instance, if arrays are described as

```
05 ARRAY-X  OCCURS    5
06 ARRAY-Y  OCCURS    3
07 ARRAY-Z  OCCURS    2
08 FILLER   PICTURE  99
```

then ARRAY-X's ASD is

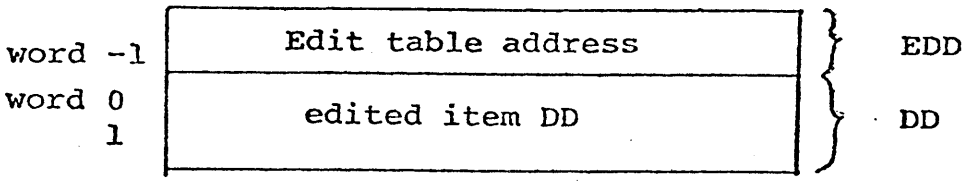
ARRAY-X's DD
1
12

and ARRAY-Z's ASD is

ARRAY-Z's DD
3
12
4
2

5.4.5 Edited Data-name Descriptor, EDD

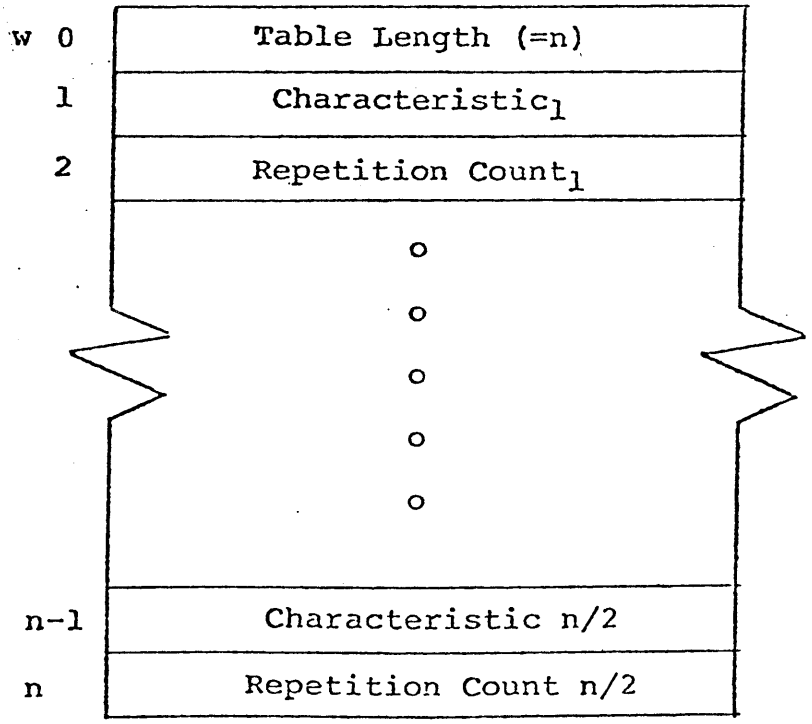
An Edited Data-name Descriptor, EDD, is generated for each edited data-name defined in the source program. It contains all the edit mask information necessary for interface between the object program and the edited move routines. The BLANK WHEN ZERO item is considered edited. The EDD precedes the DD of the edited data item.



The edit table address points to either an alphanumeric or numeric edit table.

5.4.5.1 Alphanumeric Edit Table

The table consists of a word that contains the table length in words followed by edit mask entries. Each two- word entry in the table contains an edit mask characteristic and its repetition count.



Where characteristic is

= 0 for 9, A, X

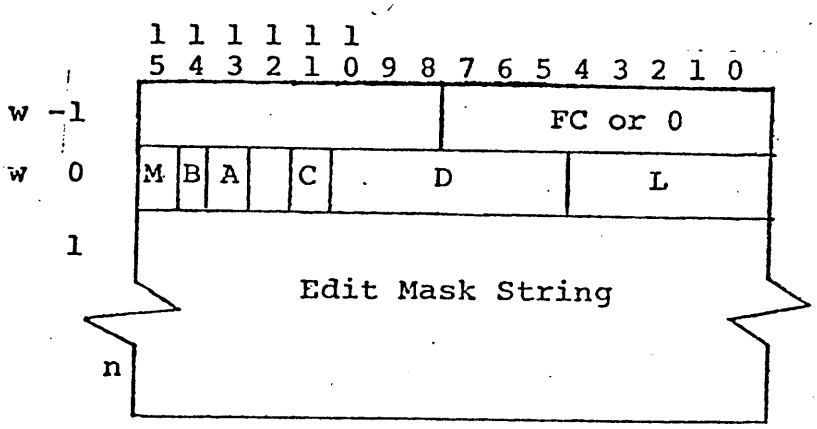
= 1 for B

= 2 for 0

= 3 for /

5.4.5.2 Numeric Edit Table

Each numeric edit table contains a two-word mask characteristic and an edit mask string which interacts with sending numeric data items to produce edited data.



FC = float character
M = 0 no edit mask string
= 1 edit mask editing
B = 0 no BLANK WHEN ZERO
= 1 BLANK WHEN ZERO
A = 0 zero suppress (Z)
= 1 asterisk protect (*)
C = 0 decimal-point is period
= 1 decimal-point is comma
D = replaceable decimal count
I = number of replaceable characters (1-18)

The Edit Mask String consists of following mask codes that represent edit character functions.

Replaceable codes are

"10" = digit select (ds); insert digit if not leading zero

"11" = significant start (ss); same as "ds", but following is significant

"12" = float start (fs); start floating insertion

"13" = start immediate (si); digit is significant

Non-replaceables are

'0' = digit 0

' ' = blank

'/' = stroke

',' = comma

'.' = period

'+' = plus

'-' = minus

'$' = dollar (fixed)

'B' = letter B

'C' = letter C

'D' = letter D

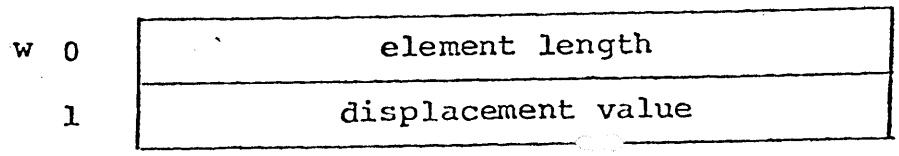
'R' = letter R

The following table lists for a given edit character the condition under which each mask code is used:

Edit Char.	Edit Code Used	Condition
9	si	Always
+	fs	If leading float
+	ss	If float and followed immediately by 9/./V
+	+	If first and non-float
+	+	If trailing
+	ds	All others
-	fs	If leading float
-	ss	If float and followed immediately by 9/./V
-	-	If first and non-float
-	-	If trailing
-	ds	All others
$	fs	If leading float
$	ss	If float and followed immediately by 9/./V
$	$	If non-float
$	ds	All others
Z	ss	If followed immediately by 9/./V
Z	ds	All others
*	ss	If followed immediately by 9/./V
*	ds	All others

5.4.6 Index-name Descriptor, XD

For each index-name defined in the compilation, a pair of words is generated as follows:



During the execution of a COBOL program, the word 1 of XD is used to hold the displacement value as specified by a most recent SET verb.

5.5 Procedure Code Generation

On the pages following, the techniques and rationale used for code generation are described.

5.5.1 Summary

The sections describing the code generation are organized under the following general headings:

1. Arithmetic
   a. ADD
   b. DIVIDE
   c. MULTIPLY
   d. SUBTRACT
   e. COMPUTE
2. Conditions
   a. Class Condition
   b. Sign condition
   c. Relational condition
3. Procedure Branching
   a. Jump Exit Table, JET
   b. Segment Interface Table, SIT
   c. GOTO
   d. GOTO DEPENDING ON
   e. ALTER
   f. PERFORM
4. Subprogram Linkage
   a. Linkage Control Block, LCB
   b. CALL
   c. EXIT PROGRAM
   d. STOP
5. Data Manipulation
   a. MOVE
   b. CONVERSION
   c. INSPECT
   d. STRING
   e. UNSTRING

6. Special Input/Output
    a. ACCEPT
    b. DISPLAY
7. Sequential I/O
    a. CLOSE
    b. OPEN
    c. READ
    d. REWRITE
    e. WRITE
8. Relative I/O
    a. CLOSE
    b. DELETE
    c. OPEN
    d. READ
    e. REWRITE
    f. START
    g. WRITE
9. Indexed I/O
    a. CLOSE
    b. DELETE
    c. OPEN
    d. READ
    e. REWRITE
    f. START
    g. WRITE
10. Subscripting/Indexing
11. Table Handling
    a. SEARCH
    b. SEARCH ALL
    c. SET
    d. OCCURS DEPENDING

12. ANS Debugging

13. IBM Extensions

    a. EXAMINE
    b. EXHIBIT
    c. TRANSFORM

14. Sort

    a. RELEASE
    b. RETURN
    c. SORT/MERGE

15. Report Writer

5.5.2 Arithmetic

The COBOL arithmetic routines develop all arithmetic results in a set of pseudo-registers allocated in memory. A total of 16 pseudo registers are always allocated by the compiler; eight are used for decimal (ASCII) arithmetic and eight are used for binary arithmetic. The decimal pseudo-registers are called DECAs and are numbered 1 through 8. (i.e., DECA 1, DECA 2, ..., DECA 8). Each DECA is 38 bytes in length; this includes an extra digit position for possible rounding.

The binary pseudo-registers are called ACCs and each ACC is 5 bytes long. Of these bytes, 4 bytes hold the binary result; the last byte contains the assumed decimal location.

5.5.2.1 ADD

- id_1 to id_2 ROUNDED

```
LOAD    r,id1
ADD     r,id2
RND     r,id2
STO     r,id2
```

- id_1, id_2 GIVING id_3 id_4 ROUNDED

```
LOAD    r,id1
ADD     r,id2
STO     r,id3
RND     r,id4
STO     r,id4
```

- id_1, id_2 TO GIVING id_3 ROUNDED id_4 ON SIZE ERROR

```
SZRS
LOAD    r,id1
ADD     r,id2
STO     r,r2
RND     r,id3
STO     r,id3
STO     r2,id4
SZJP    next sentence
  ↑
  SIZE ERROR statements
  ↓
  next sentence
```

5.5.2.2 DIVIDE

- id_1 INTO id_2 ROUNDED id_3 ON SIZE ERROR

```
SZRS
LOAD    r,id2
DIV     r,id1
RND     r,id2
STO     r,id2
LOAD    r2,id3
DIV     r2,id1
STO     r2,id3
SZJP    next sentence
  ↑
  SIZE ERROR statements
  ↓
  next sentence
```

- id_1 INTO id_2 GIVING id_3 ROUNDED REMAINDER id_4
  ON SIZE ERROR

```
SZRS
LOAD    r,id2
DIV     r,id1
STO     r,r2
RND     r,id3
STO     r,id3
MULT    r2,id1
LOAD    r3,id2
SUB     r3,r2
STO     r3,id4
SZJP    next sentence
  ↑
  SIZE ERROR statements
  ↓
  next sentence
```

5.5.2.3 MULTIPLY

- id_1 BY id_2 ROUNDED id_3 ROUNDED

```
LOAD    r,id1
MULT    r,id2
RND     r,id2
STO     r,id2
LOAD    r2,id1
MULT    r2,id3
RND     r2,id3
STO     r2,id3
```

5.5.2.4 SUBTRACT

- id_1 id_2 id_3 FROM id_4 id_5

```
LOAD    r,id1
ADD     r,id2
ADD     r,id3
LOAD    r2,id4
SUB     r2,r
STO     r2,id4
LOAD    r3,id5
SUB     r3,r
STO     r3,id5
```

5.5.2.5 COMPUTE

- id_1 ROUNDED id_2 ROUNDED = A + B - C * D ** E + F

```
LOAD    r,A
ADD     r,B
LOAD    r2,D
EXP     r2,E
MULT    r2,C
SUB     r,r2
ADD     r,F
STO     r,r3
RND     r,id1
STO     r,id1
RND     r3,id2
STO     r3,id2
```

- id_1 = A ** B + (-C + D)

```
LOAD    r,A
EXP     r,B
LOAD    r2,C
NEG     r2
ADD     r2,D
ADD     r,r2
STO     r,id1
```

5.5.3 Conditions

5.5.3.1 Class Condition

Class condition test is performed on an alphanumeric, alphanumeric edited, or numeric edited item to determine whether the item is composed entirely of ALPHABETIC (A through Z and space) or NUMERIC (0 through 9) characters. In addition, the NUMERIC test may be performed on a numeric item while the ALPHABETIC test may be performed on an alpahbetic item.

If the PICTURE of the numeric item contains an operational sign, a valid sign must be present. Valid operational signs are A-I and { for positive and J-R and } for negative. In the case of SEPARATE SIGN, valid operational signs are + and -.

- IF id ALPHABETIC

```
        CLSA    id
        BEQ     false
          ↑
        true statement
          ↓
false:  next sentence
```

- If id NOT ALPHABETIC

```
        CLSA    id
        BNE     false
          ↑
          |
        true statement
          |
          ↓
false:  next sentence
```

- If id NUMERIC

```
        CLSN    id
        BEQ     false
          ↑
          |
        true statement
          |
          ↓
false:  next sentence
```

5.5.3.2 SIGN Condition

The sign condition tests are preformed on numeric items or arithmetic expressions.

- If id POSITIVE statement

```
LOAD    r,id
TEST    r
BLE     next sentence
  ↑
statement
  ↓
next sentence
```

- If (A + B - C) NOT NEGATIVE statement

```
LOAD    r,A
ADD     r,B
SUB     r,C
TEST    r
BLT     next sentence
  ↑
statement
  ↓
next sentence
```

The false branches generated for sign conditions are summarized below:

POSITIVE	--BLE
NOT POSITIVE	--BGT
NEGATIVE	--BGE
NOT NEGATIVE	--BLT
ZERO	--BNE
NOT ZERO	--BEQ

5.5.3.3 Relational Condition

Relationals are classified as either alphanumeric or numeric.

5.5.3.3.1 Alphanumeric

The alphanumeric comparison proceeds byte by byte from left to right until an inequality is encountered. When items of unequal length are compared, the excess characters in the longer of the two items are compared to spaces.

- IF id_1 > id_2 AND id_3

```
    CMPAN     id1, id2
    BLE       false
    CMPAN     id1, id3
    BLE       false
      ↑
    True statement
      ↓
false: Next sentence
```

For relational tests involving an ALL 'leteral' where the 'literal' contains more than just a single character, the string of characters comprising the 'literal' is repeatedly compared to successive "string" unit of characters.

```
-   IF ALL 'literal' = id
    COMPFC      flag, literal, id
    BNE         false
    ↑
    True statement
    ↓
false:  Next sentence
```

where flag = 0 first operand is figurative constant

= 1 second operand

5.5.3.3.2 Numeric

Numeric comparisons involving single-word binary data-names are performed in binary mode:

```
-   IF id1 = id2 AND id3
    BLOAD   r, id1
    BCOMP   r, id2
    BNE     false
    BCOMP   r, id3
    BNE     false
    ↑
true:  |
       |
    true statement
       ↓
false:  next sentence
```

Numeric comparisons involving other than single-word binary data-names are performed in decimal mode.

- IF id_1 = id_2 OR id_3

```
    LOAD      r,id1
    COMP      r,id2
    BEQ       true
    COMP      r,id3
    BNE       false
true:  ↑
       |
    true statement
       |
       ↓
false:  next sentence
```

Comparisons involving index-names and/or index data items are performed in binary mode.

- Comparison of an index name with other than an index data item

- IF index-name > id

```
    SETLD     r, index-name
    BCMP      r, id
    BLE       false
      ↑
    true statement
      ↓
false:  next sentence
```

- Comparison involving two index-names

- IF $index\text{-}name_1$ < $index\text{-}name_2$

```
    SETLD     r, index-name1
    SETLD     r2, index-name2
```

```
    BCOMP     r, r2
    BGE       false
          ↑
true:     |
          |
    true statement
          ↓
false:  next sentence
```

. Comparison of an index data item with an index-name or with another index data item

- IF index data item = index-name

```
    BLOAD     r, index data item
    BLOADX    r2, index-name
    BCOMP     r, r2
    BNE       false
          ↑
true:     |
          |
    true statement
          ↓
false:  next sentence
```

5.5.3.4 An example of code generated by IF statement

- IF A ALPHABETIC IF B = C AND (D OR E)
  AND F STOP '1' ELSE STOP '2' ELSE STOP '3'

```
        CLSA    A
        BEQ     false1
        LOAD    r,B
        COMP    r,C
        BNE     false2
        COMP    r,D
        BEQ     true1
        COMP    r,E
        BNE     false2
true1:  COMP    r,F
        BNE     false2
        STOP    '1'
        B       next sentence
false2: STOP    '2'
        B       next sentence
false1: STOP    '3'
        next sentence
```

read

5.5.4 PROCEDURE BRANCHING

5.5.4.1 Jump Exit Table (JET)

To process ALTER and PERFORM EXIT statements and also to handle the compiler-generated GO TO which links the sections with different priority segment numbers, a table called the Jump Exit Table (JET) is produced to cause the desired program counter modification. A JET is produced in the static area.

The following conditions require an entry in the table:

1. Subject procedure-name of ALTER statement
2. Exit procedure-name of PERFORM statement
3. Section-name which is followed by a section with different segment number.

An independent segment (priority number $\geq$ 50) is always considered to be in its initial state each time it is made available to the program, while a fixed segment is always made available in its last used state.

In order to satisfy above requirements, a single JET is generated in the data area of the root module for all segments. Entries for each independent segment are grouped together so that an initialization process of entries can be performed when an independent segment is made available.

read

5.5.4.2 Segment Interface Table (SIT)

All branches into an overlayable segment are always to a single entry point. This is preferrable to having multiple entry points since a branch to another overlayable segment is effected through the MARK and CALL mechanism.

What is being passed to the overlay segment is an unique number assigned to each procedure-name referenced by other segments. The number is used as an index into the Segment Interface Table and the contents of the pointed to entry is placed in the program counter.

At the beginning of root module with segmentation present, a segment interface handler is generated.

```
SEGMENT-INTERFACE-HANDLER:
     LWL     SEGMENT-INTERFACE-HANDLER
     STW     C_SIH
     MARK    0
     LWL     C_SEGBS        external cell
     STW     2,2
     LW      0,C_SEGN
     CALL    4
     EXIT
```

At the segment entry point,

```
SEGMENT-nn-ENTRY:
     LWL     program address 0
     STW     C_SEGLOC
     LWL     SEGMENT-EXIT
     STW     C_SEXT
     LW      6,SIT
     BTOS
SEGMENT-EXIT:
     EXIT
SIT:
     ADDR    procedure-name1
     ADDR    procedure-name2
      .
      .
      .
     ADDR    procedure-namen
```

5.5.4.3 GO TO

- if procedure-name is in the root segment or in the same segment:

```
BRA     procedure-name
```

- if procedure-name is in another segment

```
MARK    "14", C_GOSG
LWL     segment base
LWL     displacement into SIT
CALL    3+2
```

C#GOSG performs following functions:

```
C_SEGBS := segment base
C_SEGN := (displacement into SIT & "7FFF")
C_SEXT := 0
If displacement $(15) = 1, then the branch is
from root to a segment, MARK + 6 := C_SIH
If displacement $(1,0) = 0, then the branch is
from segment to another segment,
    MARK + 6 := C_SEXT
    (MARK ADDRESS + 4) := C_SIH
```

- a simple GO TO (i.e., without procedure-name)

```
MARK    "14",C_GOI
LWL     JET of current paragraph-name
CALL    3 + 1
```

5.5.4.4 GO TO DEPENDING ON

- if the procedure-names referenced in the statement are all defined in the same segment or in the fixed segments

```
MARK        "14",C_GODP
Lr          where r=ACC register number
Ln          where n=number of proc arguments
LWL         proc1
LWL         proc2
 .
 .
 .
LWL         procn
CALL        3 + 2 + n
```

- if any of the procedure-names referenced is in another segment, a pair of words is generated for each procedure name

```
MARK        "14",C_GODPSG
Lr          where r=ACC register number
Ln          where n=number of proc arguments
LWL         segment base of proc1   } if proc1 is in
LWL         SIT Δ of proc1          } another segment
L0                                  } if proc2is in the
LWL         proc2                   } same segment
 .
 .
 .
LWL         segment base of procN   procn
LWL         SIT Δ of procN
CALL        3 + 2 + (2*n)
```

5.5.4.5 ALTER

- in the root segment or in the same segment

```
MARK        "14",C_ALTER
LWL         proc2
LWL         JET of proc1
CALL        3 + 2
```

- in another segment

```
MARK        "14",C_ALTRSG
LWL         segment base of proc2
LWL         SIT Δ of proc2
LWL         JET of proc1
CALL        3 + 3
```

5.5.4.6 PERFORM

The general PERFORM sequence is as follows:

- in the root segment or in the same independent segment

```
MARK        "14",C_PERFM
LWL         proc1
LWL         JET of proc2
CALL        3 + 2
```

- from the root segment to an independent segment

```
MARK        "14",C_PERFMS
LWL         segment base of proc1
LWL         SIT Δ of proc1
LWL         JET of proc2
LWL         segment base of return
LWL         SIT Δ of return
CALL        3 + 5
```

In the following descriptions of PERFORM, 'perform' is documented to mean one of the general formats above.

- PERFORM $proc_1$ thru $proc_2$

```
        perform
        PERFMT          JET of proc2
```

- PERFORM $proc_1$ id_1 times

```
        LOAD            r,id1
        TEST            r
        BLE             label2
        STO             r,temp
label1: perform
        LOAD            r,temp
        SUB             r,=1
        STO             r,temp
        TEST            r
        BGT             label1
label2: PERFMT          JET of proc2
```

- PERFORM p_1 UNTIL A + B - C = D

```
label1: LOAD            r,A
        ADD             r,B
        SUB             r,C
        COMP            r,D
        BEQ             label2
        perform
        B               label1
label2: PERFMT          JET of proc1
```

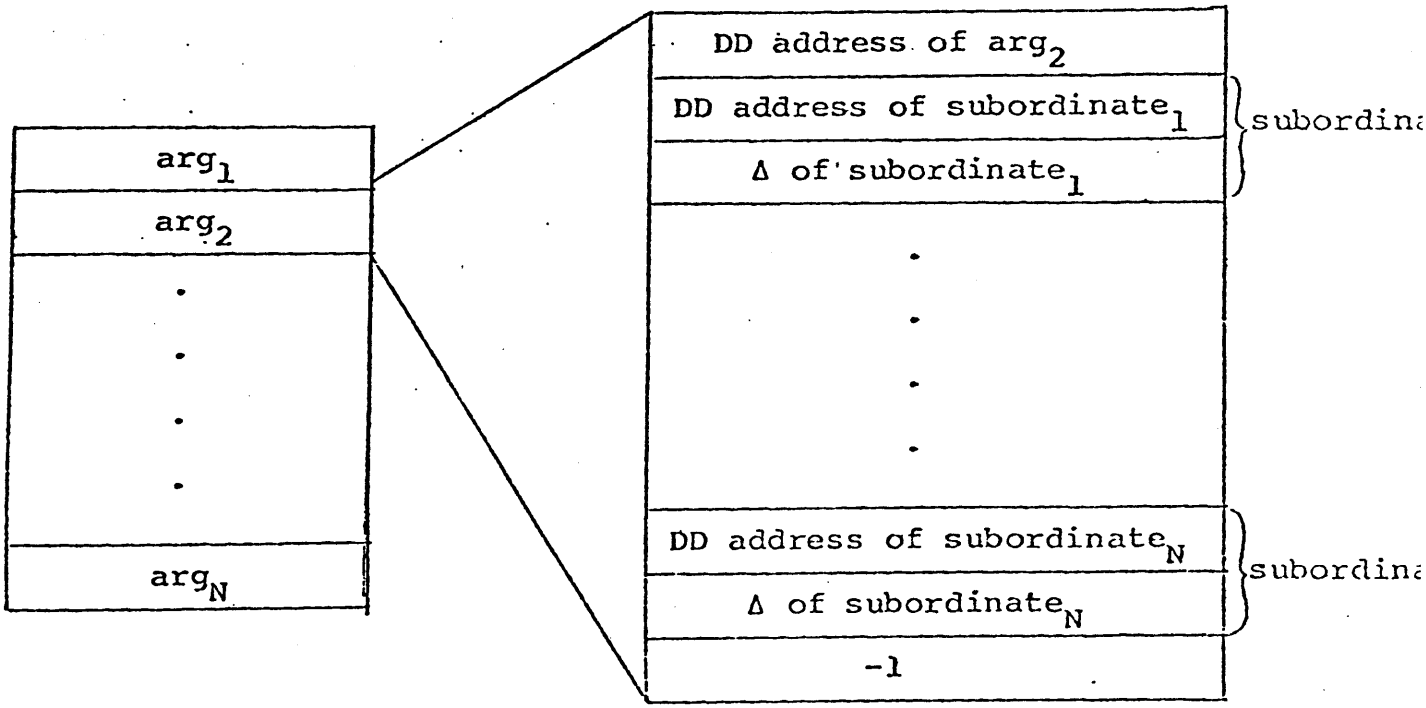
- PERFORM $proc_1$ THRU $proc_2$ VARYING id_1 FROM id_2 BY id_3 UNTIL $cond_1$ AFTER id_4 FROM id_5 BY id_6 UNTIL $cond_2$

```
            LOAD        r,id2
            STO         r,id1
            LOAD        r2,id5
            STO         r2,id4
label1:     cond1
            Bxx         true1
label2:     cond2
            Bxx         true2
            perform
            LOAD        r3,id4
            ADD         r3,id6
            STO         r3,id4
            B           label2
true1:      LOAD        r4,id5
            STO         r4,id4
            LOAD        r5,id1
            ADD         r5,id3
            STO         r5,id1
            B           label1
true2:      PERFMT      JET of proc2
```

5.5.5 Subprogram Linkage

5.5.5.1 PROCEDURE DIVISION USING

A table called the Linkage Control Block, LCB, is generated for the USING parameter list. The table is referenced by C_LINK routine and is used to transfer the absolute addresses to the Linkage Section DDs. The format of LCB is:



5.5.5.2 CALL 'ABC' USING id_1, id_2, ... id_n

```
MARK        "14",ABC
LWL         n     n=number of arguments
LWL         id1
LWL         id2
.
.
.
.
LWL         idn
CALL        3 + 1 + n
```

- at the beginning of subprogram ABC

```
MARK        "14",C_LINK
LWL         n
LWL         LCB     LCB = Linkage Control Block
CALL        3 + 2
```

5.5.5.3 EXIT PROGRAM

- in main program

```
MARK        "14",C_GOI
LWL         JET of current proc
CALL        3 + 1
```

- in subprograms

```
EXIT
```

5.5.5.4 STOP

```
        MARK        C_EXIT
        L0
        CALL        3 + 1
```

5.5.6 DATA MANIPULATION

5.5.6.1 MOVE

The transfer of data to an alphanumeric item is performed with a 'Move' operation while the transfer to a numeric item is done with a 'load/store' sequence. The table below summarizes the permissable moves and the routines that will be generated by the compiler to handle all combination of these moves:

Source / Receiving	G	A	AN	ANE	N	NE	FC& ALL	P	B
G	MVG	MVG	MVG	MVG	MVG	MVG	MVFC	MVG	MVG
A	MVG	MVA	MVA	MVA	--	--	MVFC	--	--
AN	MVG	MVA	MVA	MVA	MVA	MVA	MVFC	CVPN MVA	CVBN MVA
ANE	MVG	MVANE	MVANE	MVANE	MVANE	MVANE	MVFCE	CVPN MVANE	CVBN MVANE
N	MVG	--	LOAD STO	--	LOAD STO	--	LOADFC STO	CVPN STO	CVBN STO
NE	MVG	--	LOAD STE	--	LOAD STE	MVANE	LOADFC STE	CVPN STE	CVBN STE
P	MVG	--	LOAD CVNP	--	LOAD .CVNP	--	LOADFC CVNP	CVPN CVNP	CVBN CVNP
B	MVG	--	LOAD CVNB	--	LOAD CVNB	--	LOADFC CVNB	CVPB BSTO	BLOAD BSTO
JR	MVAJR	MVAJR	MVAJR	MVAJR	LOAD MVAJR	MVAJR	MVFCJR	CVPN MVAJR	CVBN MVAJR

where G = group
A = alphabetic
AN = alphanumeric
ANE = alphanumeric edited
N = numeric (DISPLAY format)
NE = numeric edited
FC = figurative constant
ALL = ALL 'literal'
P = packed (COMP-3 format)
B = binary (COMP & COMP-4 format)
JR = justified right

- group move

When either the source or the receiving field is a group item, a call to C_MVG is generated so that a move is performed without the mode conversion

```
MARK        "14",C_MVG
LWL         physical length of source
LWL         DD of source item
LWL         physical length of target
LWL         DD of target item
CALL        3 + 4
```

- alphanumeric move

```
MARK        "14",C_MVA
LWL         DD of source item
LWL         DD of receiving item
CALL        3 + 2
```

- ALL 'literal' source to an alphanumeric item

```
MARK        "14",C_MVFC
LWL         DD of ALL 'literal'
LWL         DD of receiving item
CALL        3 + 2
```

- ALL 'literal' source to an alphanumeric edited item

```
MARK        "14",C_MVFC
LWL         DD of ALL literal
LWL         DD of receiving item
CALL        3 + 2
```

- alphanumeric edited move

```
MARK        "14",C_MVANE
LWL         DD of source item
LWL         DD of receiving item
CALL        3 + 2
```

- when the receiving item is specified with a JUSTIFIED RIGHT caluse and the source item is not a figurative constant.

```
MARK        "14",C_MVAJR
LWL         DD of source item
LWL         DD of receiving item
CALL        3 + 2
```

- a figurative constant of an ALL literal source to a JUSTIFIED RIGHT item

```
MARK        "14",C_MVFCJR
LWL         DD of source item
LWL         DD of receiving item
CALL        3 + 2
```

- a move of ALL 'literal' source to a numeric or numeric edited item generates a call to C_LOADFC. C_LOADFC loads repetitive 'literal' or figurative constant into pseudo-register r.

```
MARK        "14",C_LOADFC
Lr
LWL         DD of source item
CALL        3 + 2
```

- numeric move

```
MARK        "14",C_LOAD
Lr
LWL         DD of source item
CALL        3 + 2
MARK        "14",C_STO
Lr
LWL         DD of target item
CALL        3 + 2
```

- numeric edited move

```
MARK        "14",C_LOAD
Lr
LWL         DD of source item
CALL        3 + 2
MARK        "14",C_STE
Lr
LWL         DD of target item
CALL        3 + 2
```

5.5.6.2 Conversions

Any of the conversion routines listed below may be thought of as a load, since they can have register receiving argument.

CVPN - packed to numeric (ASCII)
CVBN - binary to numeric (ASCII)
CVPB - packed to binary

'Store' conversion routines (register source) are

CVNP - numeric to packed
CVNB - numeric to binary
CVBN - binary to numeric
CVBP - binary to packed

For instance, a move of packed source to a numeric edited item produces following sequence of code:

```
MARK        "14",C_CVPN
LWL         DD of packed source
Lr
CALL        3 + 2
MARK        "14",C_STE
Lr
LWL         DD of receiving edit item
CALL        3 + 2
```

Another example, a statement MOVE A TO B, C, D, E. where A & B are packed items

C is a numeric item

D is a binary item

E is a numeric edited item

```
CVPN        A, r
CVNP        r, B
STO         r, C
CVNB        r, D
STE         r, E
```

When a numeric item is being compared to either an index data item or an index-name, a conversion to binary mode is required.

```
MARK        "14",C_CVNB
LWL         DD of item
Lr
CALL        3 + 2
```

5.5.6.3 INSPECT

```
LWL         DD of identifier-1
inspect argument1
inspect argument2
      .
      .
      .
inspect argumentN
terminator attribute
MARK        "14",C_INSPCT
CALL        3
```

Where 'inspect argument' is as follows:

- for TALLYING

```
LWL         Attribute
LWL         DD of identifier-3 or 0
LWL         DD of identifier-2 or 0
LWL         DD of identifier-4 or 0
```

- for REPLACING

```
LWL         attribute
LWL         DD of identifier-5 or 0
LWL         DD of identifier-6 or 0
LWL         DD of identifier-7 or 0
```

and 'attribute' is

```
Bit 7 = terminator
Bit 6 = TALLYING
Bit 5 = REPLACING
Bit 4 = AFTER INITIAL
Bit 3 = BEFORE INITIAL
Bits 2-0 = 0 CHARACTERS
         = 1  ALL
         = 2  LEADING
         = 4  FIRST
```

5.5.6.4 STRING

```
        LWL         string attribute
        LWL         DD of identifier-7
       [LWL         DD of identifier-8]   if POINTER
        LWL         delimited attribute1
       [LWL         DD of identifier-3]   if id-3 DELIMITED
        LWL         DD of identifier-1
        .
        .
        .
        .
        LWL         delimited attributeN
        LWL         DD of identifierN
        LWL         -1                    string terminator
       [LWL         label] overflow exit, if OVERFLOW
        MARK        "14",C_STRG
        CALL        3          overflow exit
        BRA         next sentence
label:  ↑
        OVERFLOW statements                if OVERFLOW specified
        ↓
next sentence:
```

string attribute =

Bit 0 = OVERFLOW present
Bit 1 = POINTER present

delimited attribute =

Bits 15-8 = 1 SIZE
= 0 identifier/literal
Bits 7-0 contains the number of 'STRING' identifiers/literals

5.5.6.5 UNSTRING

```
        LWL         unstring attribute
        LWL         DD of identifier-1
       [LWL         DD of identifier-10]   if POINTER
       [LWL         DD of identifier-11]   if TALLYING
        LWL         'delimited' option
        LWL         DD of identifier-2
        .
        .
        .
        .
        LWL         'into' option
        LWL         DD of identifier-4
       [LWL         DD of identifier-5]    if DELIMITER
       [LWL         DD of identifier-6]    if COUNT
        .
        .
        .
        .
        LWL         -1                     unstring terminator
       [LWL         label] overflow exit if OVERFLOW
        MARK        "14",C_UNSTRG
        CALL        3
        BRA         next sentence
label:                                     if OVERFLOW
        OVERFLOW statements
next sentence:
```

unstring attribute is

Bit 0 = OVERFLOW present
Bit 1 = POINTER
Bit 2 = TALLYING

'delimited' option is

Bit 0 = ALL
Bit 15 = 0 for 'delimited' option

'into' option is

Bit 0 = DELIMITER present
Bit 1 = COUNT present
Bit 15 = 1 for 'into' option

5.5.7 SPECIAL INPUT/OUTPUT

5.5.7.1 Accept

If the size of the accepting data item is greater than the maximum of logical device, as many input records as necessary are read.

- FROM CONSOLE

```
MARK        "14",C_ACPTC
LWL         DD of id
CALL        3 + 1
```

- FROM SYSIN

```
MARK        "14",C_ACPTS
LWL         DD of id
CALL        3 + 1
```

ACCEPT DATE/DAY/TIME statements generated two calls: one to load a compiler-generated item with a DATE/DAY/TIME value and another to store the value into the accepting time. The store call follows the MOVE statement rules.

- ACCEPT DATE

```
MARK        "14",C_ACPTDT
LWL         DD of CURRENT-DATE item
CALL        3 + 1
```

The routine loads CURRENT-DATE with a YYMMDD value.

- ACCEPT DAY

```
MARK        "14",C_ACPTDY
LWL         DD of DAY-OF-WEEK item
CALL        3 + 1
```

The routine loads DAY-OF-WEEK with YYDDD value.
Where DDD = Julian day.

- ACCEPT TIME

```
MARK        "14",C_ACPTTM
LWL         DD of TIME-OF-DAY
CALL        3 + 1
```

The routine loads TIME-OF-DAY with HHMMSSHH value.
where H = hour
M = minute
S = second
h = hundredth of second

5.5.7.2 DISPLAY

A maximum logical record size is assumed for each device and as many records as necessary are written to display all the operands specified.

- UPON CONSOLE

```
MARK        "14",C_DSPLC
LWL         display attribute
LWL         DD of operand1
LWL         DD of operand2
  .
  .
  .
  .
LWL         DD of operandn
CALL        3 + 1 + n
```

where display attribute is

Bit 15 = display continue code

Bits 7-0 = number of arguments. The maximum is five per call.

- UPON SYSOUT

A call to C_DSPLS is generated instead.

5.5.7.3 STOP 'literal'

```
MARK        "14",C_STOPLT
LWL         DD of literal
CALL        3 + 1
```

5.5.8 SEQUENTIAL I/O

5.5.8.1 CLOSE

```
MARK        "14",C_CLSSQ
LWL         close attribute
LWL         FIT of file
CALL        3 + 2
```

Where close attribute is

Bit 0 = CLOSE REEL/UNIT

1 = CLOSE WITH LOCK

2 = NO REWIND

5 = REMOVAL

5.5.8.2 OPEN

```
MARK        "14",C_OPNSQ
LWL         open attribute
LWL         FIT of file
[LWL        jet  Declarative JET]   if specified
CALL        jet  3 + 2[+ 1]
```

Where open attribute is

Bit 2 = NO REWIND

3 = REVERSED

4 = EXTEND

5 = I-O

6 = OUTPUT

7 = INPUT

8 = Declarative JET argument present

5.5.8.3 READ

- READ record-name INTO id AT END imperative-statements

```
        MARK        "14",C_REDSQ
        LWL         read attribute
        LWL         FIT of record-name's file
        CALL        3 + 2
        BRA         label1          at end condition exit
          ↑
        move record-name to id
          ↓
        BRA         label2
          ↑
label1:
        AT END imperative-statements
          ↓
label2: next sentence
```

Where read attribute is

Bit 0 = AT END imperative statements present

5.5.8.4 REWRITE

- REWRITE record-name FROM id

```
        Move id to record-name
        MARK        C_RWRSQ
        L0
        LWL         FIT of record-name's file
        CALL        3 + 2
```

NOTE: The sequential I/O REWRITE is meaningful only in a mass storage (disk) file and the file must be in I-O access mode.

5.5.8.5 WRITE

```
        MARK   "14",C_WRTSQ
        LWL    write attribute
        LWL    FIT of record-name's file
        LWL    DD of record-name or 0 if fixed length
        LWL    integer-or-0
        LWL    DD of identifier-2
        CALL   3 + 3 [+1]
[       BRA    label                                          ]
[       BRA    next sentence                                  ]
[label:                                                       ] if EOP is specified
[       EOP    imperative statements                          ]
next sentence:
```

Write attribute is

Bit 0 = EOP present

1 = PAGE

2 = ~~spare~~ POSITIONING

3 = BEFORE ADVANCING

4 = AFTER ADVANCING

5 = ~~spare~~

6 = ~~ADVANCING~~ integer present stet

7 = ADVANCING ~~identifier~~ present stet

5.5.9 RELATIVE I/O

5.5.9.1 CLOSE

```
        MARK        "14",C_CLSRL
        LWL         close attribute
        LWL         FIT of file
        CALL        3 + 2
```

Close attribute is

```
        Bit 1 = CLOSE WITH LOCK
```

5.5.9.2 DELETE

- DELETE file INVALID KEY imperative statement

```
        MARK        "14",C_DLTRL
        LWL         delete attribute
        LWL         FIT of file
        CALL        3 + 2
        BRA         label
        BRA         next sentence
label:  ↑
        INVALID KEY imperative statements
        ↓
next sentence:
```

if INVALID KEY is specified

Delete attribute is

```
        Bit 0 = INVALID KEY present
```

5.5.9.3 OPEN

```
        MARK        "14",C_OPNRL
        LWL         open attribute
        LWL         FIT of file
        [LWL  jet0  Declarative JET]    if specified
        CALL  jet1  3 + 3 [+1]
```

Where open attribute is

Bit 5 = I-O

Bit 6 = OUTPUT

Bit 7 = INPUT

Bit 8 = Declarative JET present

5.5.9.4 READ

- READ file INTO id AT END statements

```
        MARK        "14",C_REDRL
        LWL         read attribute
        LWL         FIT of file
        CALL        3 + 2
        [BRA        label1]          abnormal exit if AT END
          ↑                          present
        Move temp to KEY             if KEY conversion required
        Move record-name to id
          ↓
        BRA         label2
          ↑
label1:
        AT END/INVALID statements
          ↓
label2: next sentence
```

Where read attribute is

Bit 0 = AT END/INVALID KEY present

Bit 2 = NEXT

5.5.9.5 REWRITE

```
                MARK        "14",C_RWRRL
                LWL         Rewrite attribute
                LWL         FIT of file
                CALL        3 + 2
                BRA         label
                BRA         next sentence
                  ↑
label:            |
                  |
                INVALID KEY statements
                  ↓
next sentence:
```

Rewrite attribute is

Bit 0 = INVALID KEY present

5.5.9.6 START

```
                MARK        "14",C_STTRL
                LWL         Start attribute
                LWL         FIT of file
                CALL        3 + 2
                BRA         label
                BRA         next sentence
                  ↑
label:            |
                  |
                INVALID KEY statements
                  ↓
next sentence:
```

Start attribute is

Bit 0 = INVALID KEY present

Bit 2 = EQUAL relational

Bit 3 = GREATER relational

Bit 4 = NOT LESS relational

5.5.9.7 WRITE

```
                MARK        "14",C_WRTRL
                LWL         Write attribute
                LWL         FIT of file
                CALL        3 + 2
                BRA         label
                BRA         next sentence
                ↑
label:          |
                |
                INVALID KEY statements
                ↓
next sentence:
```

Write attribute is

```
                Bit 0 = INVALID KEY present
```

5.5.10 INDEXED I/O

5.5.10.1 CLOSE

```
            MARK        "14", C-CLSIX
            LWL         close attribute
            LWL         FIT of file
            CALL        3 + 2
```

Where close attribute is

Bit 1 = CLOSE WITH LOCK

5.5.10.2 DELETE

```
            MARK        "14", C-DLTIX
            LWL         delete attribute
            LWL         FIT of file
            CALL        3 + 2
            BRA         label           normal return
            BRA         next sentence
label:      ↑
            INVALID KEY statements
            ↓
next sentence:
```

Where delete attribute is

Bit 0 = INVALID KEY present

5.5.10.3 OPEN

```
            MARK        "14", C-OPNIX
            LWL         open attribute
            LWL         FIT of file
            [~~LWL~~    jet. Declarative JET]
            CALL        jet 3 + 2 [+ 1]
```

Where open attribute is

Bit 5 = I-O

Bit 6 = OUTPUT

Bit 7 = INPUT

Bit 8 = Declarative JET present

5.5.10.4 READ

- READ file INTO id INVALID KEY statements

```
        MARK        "14", C-REDIX
        LWL         read attribute
        LWL         FIT of file
[       LWL         DD of data-name]
        CALL        3 + 2
        BRA         label1   AT END/INVALID KEY exit
        |
        |
        move record-name to id
        |
        |
        BRA         label2
label1:  ↑
label1:  |
         |
         |
         INVALID KEY statements
         |
         |
         ↓
label2:next sentence
```

Where read attribute is

Bit 0 = AT END or INVALID KEY present
Bit 1 = data-name argument present
Bit 2 = NEXT

5.5.10.5 REWRITE

```
            MARK        "14", C-WRTIX
            LWL         rewrite attribute
            LWL         FIT of file
            CALL        3 + 2
            BRA         label
            BRA         next sentence
label:      ↑
            |
            INVALID KEY statement
            |
            ↓
next sentence
```

Rewrite attribute is

Bit 0 = INVALID KEY present

5.5.10.6 START

```
            MARK        "14", C-STTIX
            LWL         start attribute
            LWL         FIT of file
           [LWL         DD of data-name]
            CALL        3 + 2 [+ 1]
            BRA         label
            BRA         next sentence
label:      ↑
            |
            INVALID KEY statements
            |
            ↓
next sentence:
```

Where start attribute is

Bit 0 = INVALID KEY present

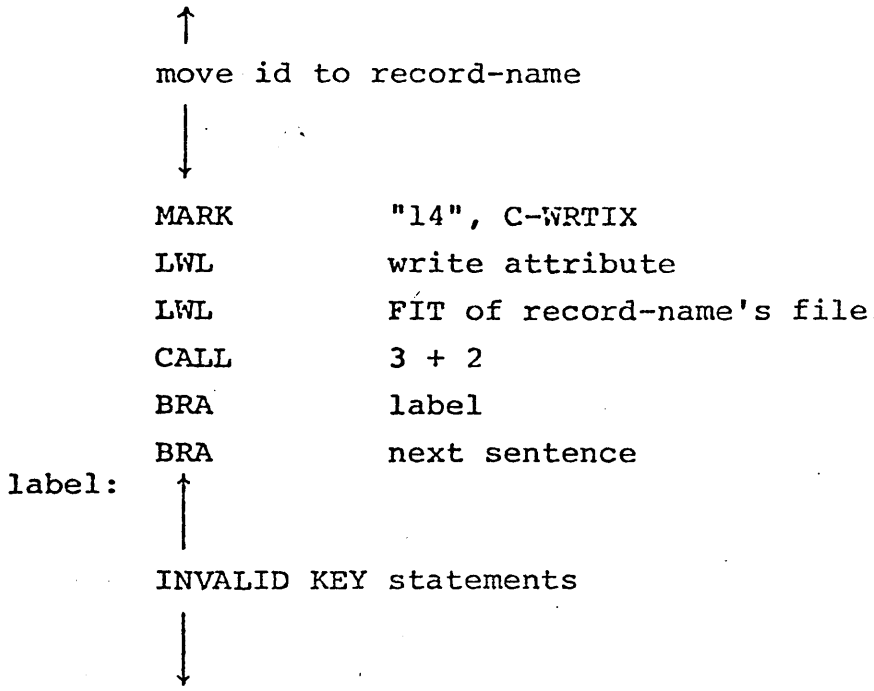
Bit 1 = data-name argument present

Bit 2 = EQUAL relational

Bit 3 = GREATER relational

Bit 4 = NOT LESS relational

5.5.10.7 WRITE



Where write attribute is

Bit 0 = INVALID KEY present

5.5.11 SUBSCRIPTING/INDEXING

5.5.11.1 Subscript

```
MARK        "14", C-SCRIPT
LWL         DD of table item
LWL         Sum of constant subscripts
LWL         DD of subscript1 or 0
LWL         DD of subscript2 or 0
LWL         DD of subscript3 or 0
LWL         DD of dummy
CALL        3 + 6
```

C-SCRIPT places the computed address of a table item into the address word of dummy DD.

5.5.11.2 Index

```
MARK        "14", C-INDEX
LWL         DD of table item
LWL         Sum of constant subscripts
LWL         DD of subscript1 or 0
LWL         DD of subscript2 or 0
LWL         DD of subscript3 or 0
LWL         DD of dummy
CALL        3 + 6
```

5.5.12 Table Handling

5.5.12.1 SEARCH

```
          MARK          "14", C-SRCI
          LWL           Search attribute
          LWL           Maximum occurrence #
          LWL           DD of OCCURS DEPENDING item
          LWL           XD of varying index-name
          CALL          3 + 3
label1:   MARK          "14", C-SRC
          LWL           XD of varying index-name
          LWL           label2
          CALL          3 + 2
            ↑
          AT END statements
            |
            ↓
          BRA           next sentence
label2:     ↑
            |
          WHEN1 condition          } WHEN
            |                      }
            ↓                      }
          BRA           next sentence
label3:     .                      }
            .                      }
            .                      }
            .                      } WHEN2 - WHENn
            .                      }
            .                      }
            .                      }
```

```
label_n:  MARK      "14",C_BLOAD
          Lr
          LWL       DD of binary literal 1
          CALL      3 + 2
          MARK      "14",C_SETAX
          Lr
          LWL       XD of varying index-name
          CALL      3 + 2
        ┌ MARK      "14",C_SETAX                                  ┐
        | Lr                                                      | if specified
        | LWL       XD of second varying index-name               |
        └ CALL      3 + 2                                         ┘
          BRA       label_1
next sentence:
```

where search attribute is

Bit 0 = 0 maximum occurrence argument is literal

= 1 maximum occurrence argument is data-name

5.5.12.2 SEARCH ALL

```
                MARK            "14",C_SRCI
                LWL             Search attribute
                LWL             Maximum occurrence #
                LWL             DD of OCCURS DEPENDING item
                LWL             XD of varying index-name
                CALL            3 + 3
label1:         MARK            "14",C_SRCA
                LWL             XD of varying index-name
                LWL             label2
                CALL            3 + 2
                ↑
                AT END statements
                ↓
                BRA             next sentence
label2:         MARK            "14",C_SRAD
                LWL             Search direction attribute
                CALL            3 + 1
                ↑
                WHEN condition
                ↓
                MARK            "14",C_BNE
                LWL             label3
                CALL            3 + 1
                ↑
                WHEN imperative statements
                ↓
                BRA             next sentence
label3:         BRA             label1
next sentence:
```

Where search attribute is

Bit 0 = 0 Maximum occurrence argument is literal

= 1 Maximum occurrence argument is data-name

and search direction attribute is

Bit 15 = 0 DESCENDING

= 1 ASCENDING

5.5.12.3 SET

The compiler generates calls to convert from and to occurrence number and array offset values when either the sending or the receiving field is an index-name.

Permissable SET fields are:

Case	Source	Receiving	Action
1	integer	index-name	occur # ⟶ offset
2	index data item	index-name	no conversion
3	index-name	index-name	offset⟶occur#⟶offset
4	index data item	index data item	no conversion
5	index-name	index data item	no comversion
6	index-name	integer	offset⟶ occur#

Case 1 integer ⟶ index-name

```
MARK        "14",C_CVNB
Lr
LWL         DD of integer item
CALL        3 + 2
MARK        "14",C_SETST
Lr
LWL         XD of index-name
CALL        3 + 2
```

Case 2 index data item ⟶ index-name

```
MARK        "14",C_BLOAD
Lr
LWL         DD of index data item
CALL        3 + 2
MARK        "14",C_BSTX
Lr
LWL         XD of index-name
CALL        3 + 2
```

Case 3 index-name$_1$ ⟶ index-name$_2$

```
MARK        "14",C_SETLD
Lr
LWL         XD of index-name1
CALL        3 + 2
MARK        "14",C_SETST
Lr
LWL         XD of index-name2
CALL        3 + 2
```

Case 4 index data item$_1$ ⟶ index data item$_2$

```
MARK        "14",C_BLOAD
Lr
LWL         DD of index data item1
CALL        3 + 2
MARK        "14",C_BSTO
Lr
LWL         DD od index data item2
CALL        3 + 2
```

Case 5 index-name ⟶ index data item

```
MARK        "14",C_BLOADX
Lr
LWL         XD of index-name
CALL        3 + 2
MARK        "14",C_BSTO
Lr
LWL         DD of index data item
CALL        3 + 2
```

Case 6 index-name ⟶ integer

```
MARK        "14",C_SETLD
Lr
LWL         XD of index-name
CALL        3 + 2
MARK        "14",C_BSTO
Lr
LWL         DD of integer item
CALL        3 + 2
```

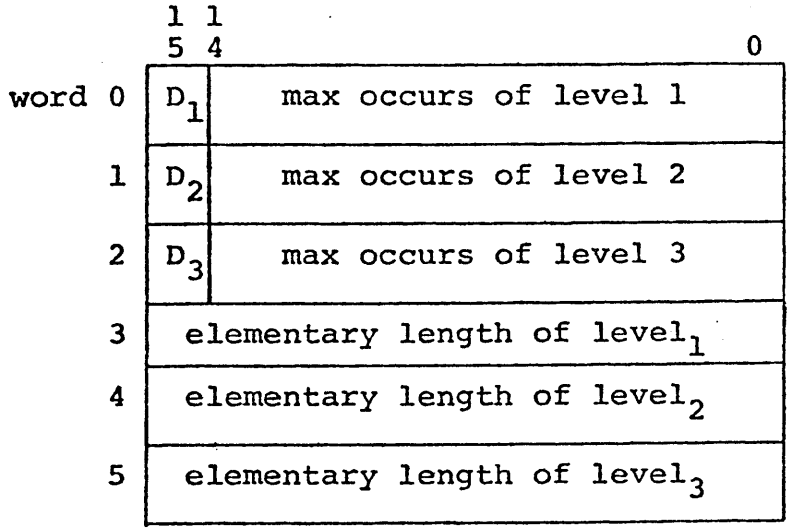
5.5.12.4 SET UP/DOWN

```
MARK        "14",C_BLOAD
Lr
LWL         DD of integer
CALL        3 + 2
MARK        "14",C_BNEG        if DOWN BY
Lr
CALL        3 + 1
MARK        "14",C_SETAX
Lr
LWL         XD of index-name
CALL        3 + 2
```

5.5.12.5 OCCURS DEPENDING

A table called the Variable Array Table, VAT, is generated for each array that contains an OCCURS DEPENDING item.

The format of VAT is:



where

D_n = occurs depending level indicator

max occurs of level n = maximum occurrence number specified for that level. For example, if OCCURS FROM 1 TO 28 TIMES then the maximum occurrence is 28.

elementary length of $level_n$ = the sum of physical byte length of every elementary items for that level.

- Data-name in variable array

  A call to C_VAR is generated for each reference to variable data-name.

```
MARK        "14",C_VAR
LWL         DD of variable item
LWL         DD of occurs depending in binary
LWL         VAT
LWL         dummy DD
CALL        3 + 4
```

- Index-name in variable array

  A call C_VARX is generated for each reference to an index-name that refers to variable array.

```
MARK        "14",C_VARX
LWL         DD of variable item
LWL         DD occurs depending in binary
LWL         VAT
LWL         XD of index-name
CALL        3 + 4
```

For instance, if an array is described as:

```
01  A.
  02  B  PIC X(3).
  02  C.
    03  D  PIC X(5)
    03  E  OCCURS 5.
      04  F  PIC X(2).
      04  G  OCCURS 3  PIC X(2).
      04  H  OCCURS 4.
        05  I  PIC X(5).
        05  J  OCCURS 2  PIC X(2).
        05  K.
          06  L  OCCURS 2 TO 8 DEPENDING ON Z.
            07  M  PIC X(8).
            07  N.
              08  O  PIC X(2).
              08  P  PIC X(3).
```

then all data items except B are considered variable since the length or/and the definition address is/are altered according to the current value of Z.

The array A's VAT is:

word 0	0	5	from E
1	0	4	from H
2	1	8	from L
3	2 + (2*3) = 8		from F & G
4	5 + (2*2) = 9		from I & J
5	8 + 2 + 3 = 13		from M, O & P

and a reference to J produces a call to C_VAR as:

```
MARK        "14",C_VAR
LWL         8000          DD of variable J
LWL         8700          DD of variable Z in binary
LWL         B200          VAT
LWL         9C00          dummy DD
CALL        4
```

where

```
(8000) = 0002  }
(8001) = 7403  }  DD of J

(8002) =    3  }                  # of dimensions
(8003) =  460  }  ASD of J        length of level1
(8004) =  113  }                  length of level2
(8005) =    2  }                  length of level3

(8700) = 8004  }
(8701) = 7A50  }  DD of Z
(7A50) = 0003     contents of Z
```

(B200) - as described for array A.

C_VAR performs following calculations:

```
(9C00) = 0002    from the first word of J's DD
(9C01) = 7403
(9C02) =    3
(9C03) =  148
(9C04) =   35
(9C05) =    2
```

5.5.13 ANS Debugging

An object-time switch is used to activate the debug declaratives. The switch is tested by each of debugging routines and if not set, following calls are treated as a non-functional.

5.5.13.1 Procedure-name

Case 1 PERFORM - immediately before the 'perform' of procedure-name

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
CALL        3 + 3
```

debug attribute =

Bits 7-4 = 1 PERFORM LOOP

Bits 3-0 = 0

Case 2 ALTER - immediately after the execution of ALTER

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
LWL         symbolic string of ALTERED procedure1
LWL         symbolic string of ALTERED procedure2
CALL        3 + 5
```

↑
perform debugging declarative
↓

debug attribute is

Bits 7-4 = 0 second symbolic string to DEBUG-CONTENTS

Bits 3-0 = 2 number of symbolic string arguments

Case 3 USE procedure

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
LWL         symbolic string of USE procedure
CALL        3 + 4
```

↑

perform debugging declarative

↓

debug attribute is

Bits 7-4 = 2 USE PROCEDURE "21"

Bits 3-0 = 1 number of symbolic arguments

Case 4 At the program entry - immediately before the first non-declarative procedure-name definition.

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
CALL        3 + 3
```

debug attribute is

Bits 7-4 = 3 START PROGRAM "30"

Bits 3-0 = 0

Case 5 GO TO - immediately before GO TO

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
CALL        3 + 3
```

debug attribute is

Bits 7-4 = 0 "0?"

Bits 3-0 = 0

Case 6 implicit transfer of control to procedure-name

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
CALL        3 + 3
```

debug attribute =

Bits 7-4 = 4 FALL THROUGH "4?"

Bits 3-0 = 0

Case 7 immediately after the procedure-name definition

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
LWL         symbolic string of procedure-name
CALL        3 + 4
  ↑
perform debug declarative
  ↓
```

debug attribute =

Bits 7-4 = 8 do not clear the DEBUG-ITEM item

Bits 3-0 = 1 place symbolic string to DEBUG-NAME "81"

Case 8 a reference to procedure-name in the INPUT or OUTPUT phrase of a SORT or MERGE statement

```
MARK        "14",C_DBGSU
LWL         DD of DEBUG-ITEM item
LWL         debug attribute
LWL         Line number
LWL         symbolic string of procedure-name
CALL        3 + 4
  ↑
perform debug declarative
  ↓
debug attribute =
                Bits 7-4 = 5    SORT INPUT
                         = 6    SORT OUTPUT
                         = 7    MERGE OUTPUT
                Bits 3-0 = 1    symbolic string to
                                  DEBUG-NAME
```

5.5.13.2 Identifier (data-name)

```
MARK            "14",C_DBGSU
LWL             DD of DEBUG-ITEM item
LWL             debug attribute
LWL             line number
LWL             symbolic string of identifier
CALL            3 + 4
  ↑
move subscript1 to DEBUG-SUB-1
move subscript2 to DEBUG-SUB-2
move subscript3 to DEBUG-SUB-3
move identifier to DEBUG-CONTENTS
  ↓
  ↑
perform debug declarative
  ↓
where debug attribute is
                Bits 7-4 = 9 identifier, move
                             spaces to DEBUG-SUB-1
                             thru -3.
                Bits 3-0 = 1 symbolic string to
                             DEBUG-NAME.
```

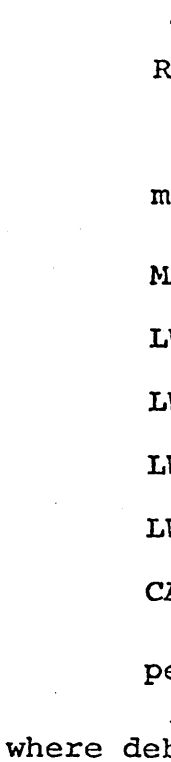
"9.1"

5.5.13.3 File-name

- other than READ statement

```
MARK            "14",C_DBGSU
LWL             DD of DEBUG-ITEM item
LWL             debug attribute
LWL             line number
LWL             symbolic string of file-name
CALL            3 + 4
  ↑
perform debug declarative
  ↓
where debug attribute is:
                Bits 7-4 = 10   file-name, move spaces
                                to DEBUG-CONTENTS
                Bits 3-0 = 1    symbolic string to
                                DEBUG-NAME
```

- READ statement



where debug attribute is

Bits 7-4 = 11 file-name, DEBUG-CONTENTS IS already initialized with the record just read.

Bits 3-0 = 1 symbolic string to DEBUG-NAME

Note: The symbolic string arguments in calls to C_DBGSU are in the form of:

WORD byte count of string (=n)

$WORD_0$ ⎫
⎪ ⎬ symbolic string
$WORD_{(n+1)/2}$ ⎭

5.5.14 IBM Extensions

5.5.14.1 EXAMINE

The EXAMINE statement produces a call to C_INSPCT with appropriate attributes. See code generation for INSPECT statements for more information.

5.5.14.2 EXHIBIT

```
        MARK        "14", C_EXHBT
        LWL         exhibit attribute
       [LWL         changed save area]  if CHANGED
        LWL         exhibit operand attribute1
       {LWL         DD of id1          }
       {LWL         DD of literal1     }
       [LWL         symbolic string of id1]  if NAMED
        .
        .
        .
        .
        .
        LWL         exhibit operand attributen
       {LWL         DD of idn          }
       {LWL         DD of literaln     }
       [LWL         symbolic string of idn]  if NAMED
        LWL         exhibit operand attributen+1
        MARK        "14",C_EXHBT
        CALL        3
```

where exhibit attribute is

Bits 1-0 = 0 NAMED

= 1 CHANGED

= 2 CHANGED NAMED

and exhibit operand attribute is

Bit 0 = 0 identifier argument

= 1 literal argument

Bits 3-0 = 15 end of argument

5.5.14.3 TRACE

The IBM debugging switch, C_~~IBDBG~~ TRACEFLAG, is reserved by the compiler and is referenced by C_TRON, C_TROFF and C_TRACE routines.

- READY TRACE

```
MARK        "14",C_TRON
CALL        3
```

- RESET TRACE

```
MARK        "14",C_TROFF
CALL        3
```

- The trace calls are generated at each section or paragraph definition point as follows:

```
MARK        "14",C_TRACE
LWL         symbolic string of procedure-name
CALL        3 + 1
```

where symbolic string is in the form of

```
DATA        byte count of string
DATA
  .
  .         symbolic string of procedure-name
  .
DATA
```

5.5.14.4 TRANSFORM

```
MARK        "14",C_TRSFRM
LWL         DD of id3
LWL         DD of id1
LWL         DD of id2
CALL        3 + 4
```

5.5.15 Sort/Merge

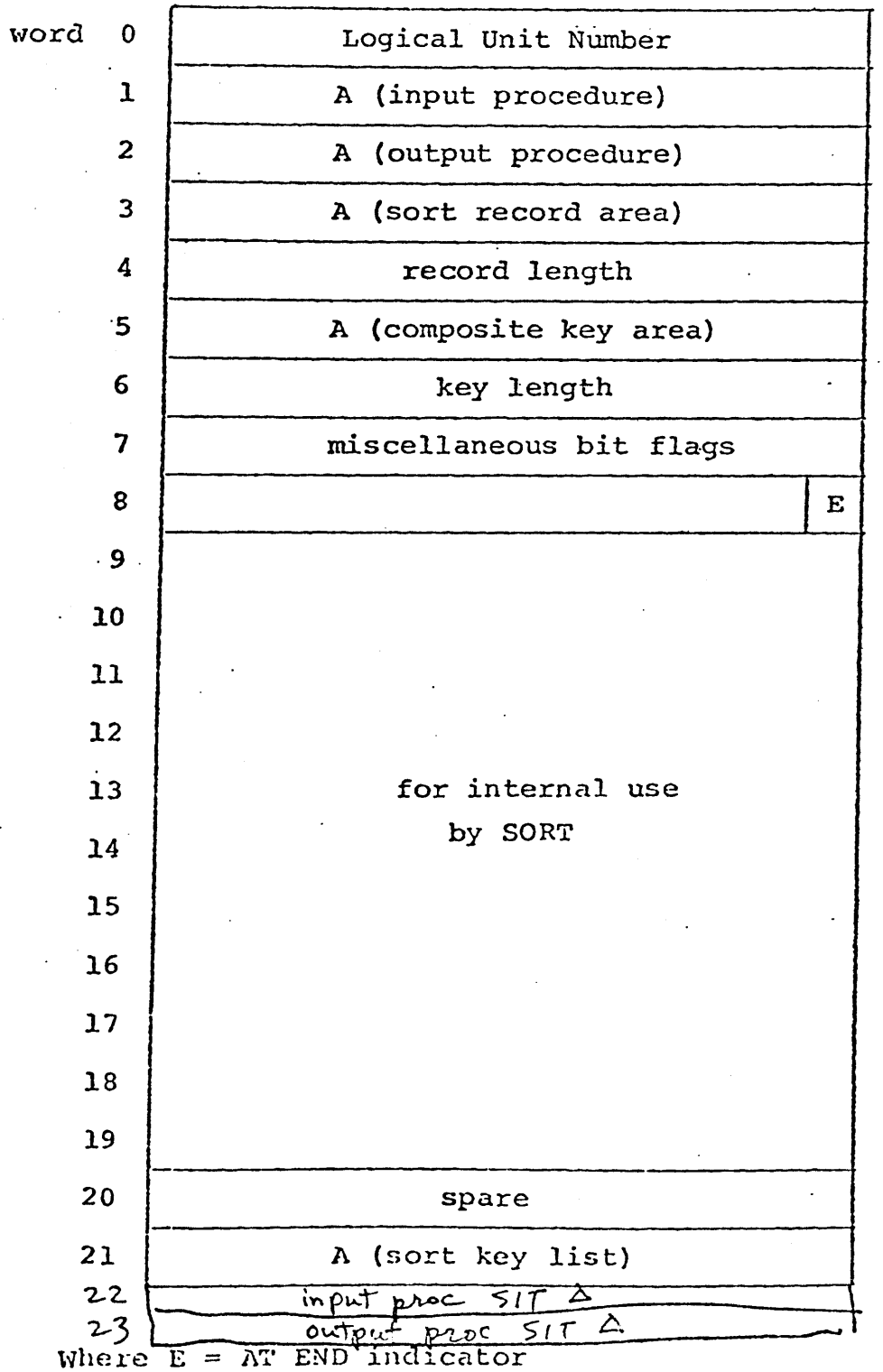
The compiler generates an implicit input procedure if the SORT statement includes a USING clause. Following sequence of code is produced for each of file-name specified.

```
            OPEN INPUT file-name_n
LOOP.       READ record-name_n AT END GO TO CLOSE-FILE.
            RELEASE sort-record FROM record-name.
            GO TO LOOP.
CLOSE-FILE.   CLOSE file-name_n
```

Likewise, if the SORT statement includes a GIVING clause instead of OUTPUT PROCEDURE, the compiler produces equivalent text.

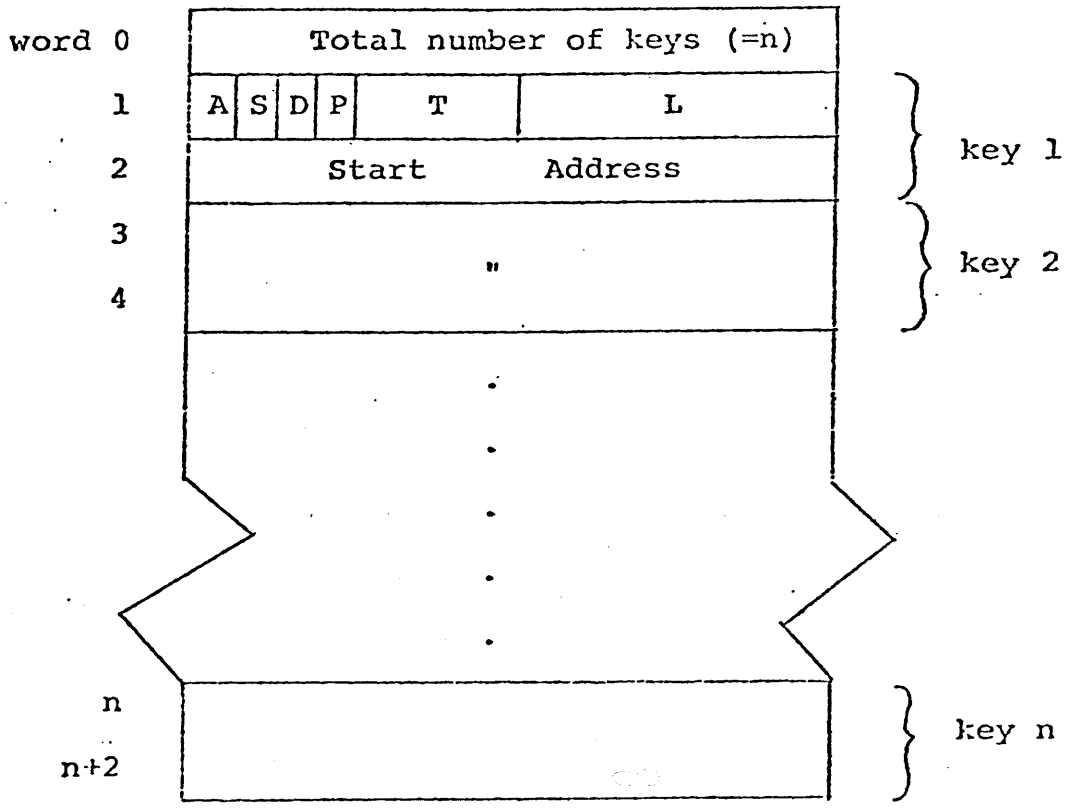
```
            OPEN OUTPUT file-name_n
LOOP.       RETURN file-name_n AT END GO TO CLOSE-FILE.
            WRITE sort-record FROM record-name.
            GO TO LOOP.
CLOSE-FILE.     CLOSE file-name
```

5.5.15.1 Sort Control Block, SCB



Where E = AT END indicator

5.5.15.2 Sort Key List



Bit 15 A = 0 ASCENDING
= 1 DESCENDING

Bits 11-8 T = type
= 0 alphanumeric
= 1 numeric
=2 numeric packed (COMP-3)
=3 numeric ASCII (DISPLAY)
Bit 14 S = 1 signed
Bit 13 D = 1 leading sign
Bit 12 P = 1 sign to separate
Bits 7-0 L = Byte length of key (≤255)

5.5.15.3 RELEASE

```
MARK        "14",C_RELSE
LWL         FIT of file to be sorted
LWL         SCB
CALL        3 + 2
MARK        "14",C_SRTRTN
LWL         JET of input procedure
CALL        3 + 1
EXIT
```

label:

C_RELSE constructs a single composit key in the area pointed to by word 5 of SCB. The composite key is in an ascending logical order. Following transformations are required on each key:

- alphanumeric - none

  numeric - convert to binary and add bias.

  Bias = "8000———0"

- if descending keys, do 1'1 complement.

C_SRTRTN places the address of label (immediately after the EXIT instruction) into the address portion of pointed to JET.

5.5.15.4 RETURN

```
        MARK            "14",C_SRTRTN
        LWL             JET of output procedure
        CALL            3 + 1
        EXIT
        MARK            "14",C_RETRN
        LWL             return attribute
        LWL             FIT of sorted output file
        LWL             SCB
        CALL            3 + 3
        BRA             label
                          ↑
        move record-area to INTO id
                          ↓
        BRA             next sentence
                          ↑
label:                    |
        AT END          imperative statements
                          ↓
next sentence:
```

where return attribute is

Bit 0 = AT END statements present

5.5.15.5 SORT/MERGE

```
MARK        "14",SCBSU
LWL         SCB
LWL         FIT of sort file
LWL         input procedure
LWL         output procedure
LWL         sort key length
LWL         sort key list
CALL        3 + 6
MARK        "14",C_SORT
LWL         SCB
CALL        3 + 1
```

5.5.15.6 Input Procedure of Sort

The input procedure of a sort has following sequence of codes,

```
entry:  GOI         sort proc JET
          .
          .
          .
          .
          .
        RELSE       FIT,SCB
        SRTRTN      sort proc JET
        EXIT
          .
          .
          .
          .
        SRTEOF      SCB
        EXIT
```

5.5.15.7 Output Procedure of Sort

The output procedure of a sort has following sequence of codes,

```
entry:  GOI          sort proc JET
          .
          .
          .
          .
          .
        SRTRTN       sort proc JET
        RETRN        attribute, FIT, SCB
        BRA          label
        move record-area to INTO id
        BRA          next sentence
          .
label:    .
          .
        AT END statements
          .
          .
          .
next sentence:
        SRTRTN       sort proc JET
        EXIT
```

5.5.16 Report Writer

Report Writer processing takes place in phases 1 through 5. Phase 1 merely processes report file FDs like any other FDs, except that report-names specified on REPORTS ARE clauses are saved on the Report stack for later processing. The Report Writer phase is called when a REPORT SECTION is recognized by the scan mechanism. This phase is composed of two parts: report writer syntax analysis and the report writer encode. Each part is essentially a pass over the report writer source. The first pass places pertinent information out to the DT-Text and/or RW-Text after each line is analyzed syntactially. When the end of DATA DIVISION is detected, the encoding of RW-Text takes place by making a pass over the text and producing EP-Text for report writer procedures. Phases 4 and 5, in turn, translates Report Writer DT-Text into report writer data blocks and report writer EP-Text into report writer OP-Text, respectively.

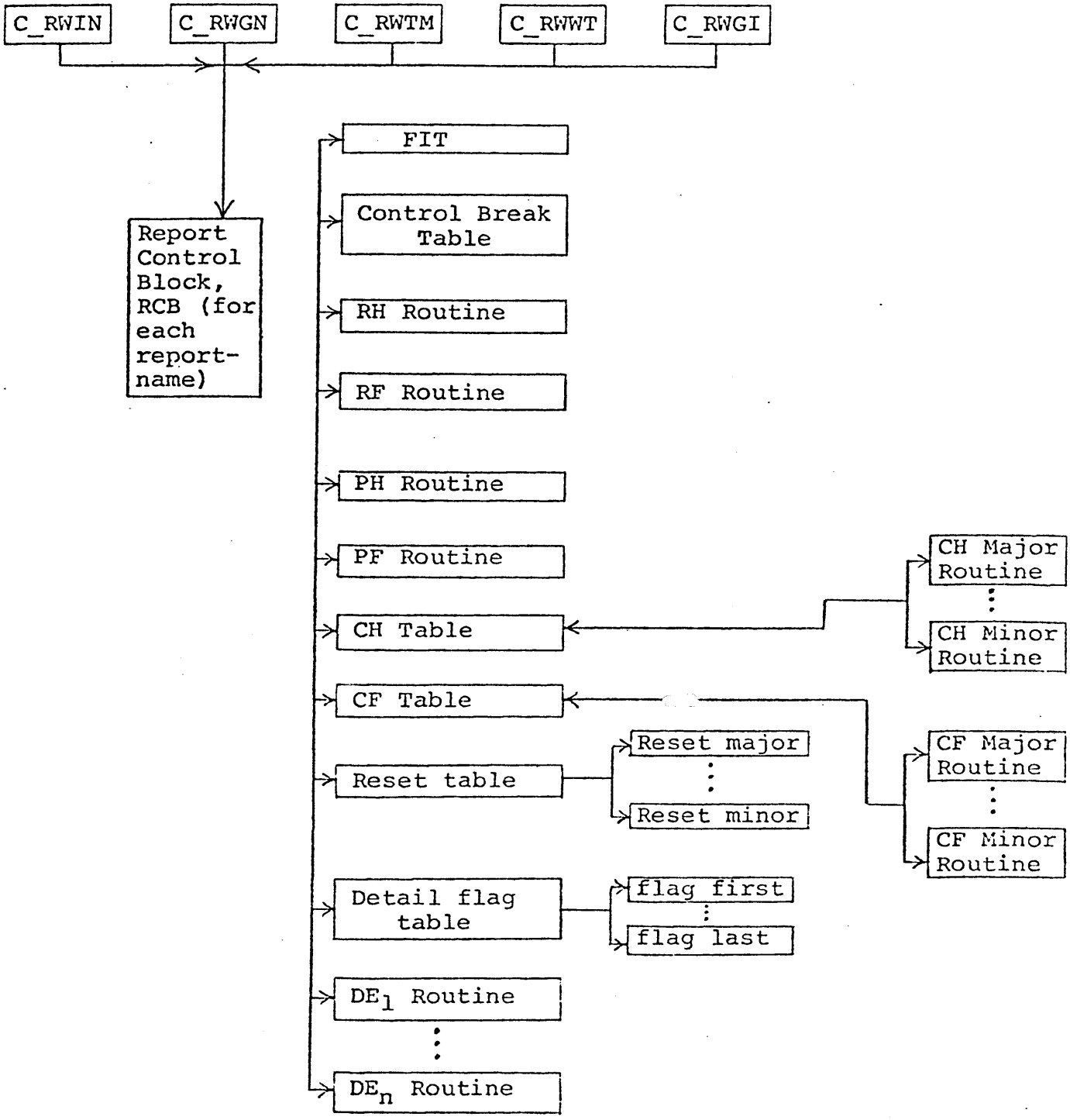
5.5.16.1 Operations

Following table shows the Report Writer operations that occur in response to the various types of source statements:

Source Statements	Summary of Compiler Activities
FD...REPORT IS	Store report-names in Report Stack for diagnostic purpose.
RD report-name	. Make an entry in File Stack for each report-name . Generate Report Control Blocks
CONTROLS ARE	. Assign control level numbers to data names
PAGE, HEADING, etc.	. Save line numbers for printer carriage spacing on File Stack . Significant line numbers are placed in RCB
TYPE	identifies Report Group
COLUMN	defines the column position in print buffer
SOURCE	generate 'MOVE report item to print buffer'
VALUE	generate 'MOVE value to print buffer'
SUM	generate 'ADD operand to sum-counter' in a summing routine for the group the sum-counter references.
02(-49)...LINE	generate a call to C_RWWT with RCB address and line spacing information as arguments.
USE BEFORE REPORTING	generate a 'perform' of declarative section at the entry of report group routine
INITIATE report-name	generate a call to C_RWIN with RCB address argument
GENERATE report-name	generate a call to C_RWGN with RCB address and a zero DE numbers arguments.
GENERATE detail-name	generate a call to C_RWGN with RCB address and DE number arguments.
TERMINATE report-name	generate a call to C_RWTM with RCB address argument.

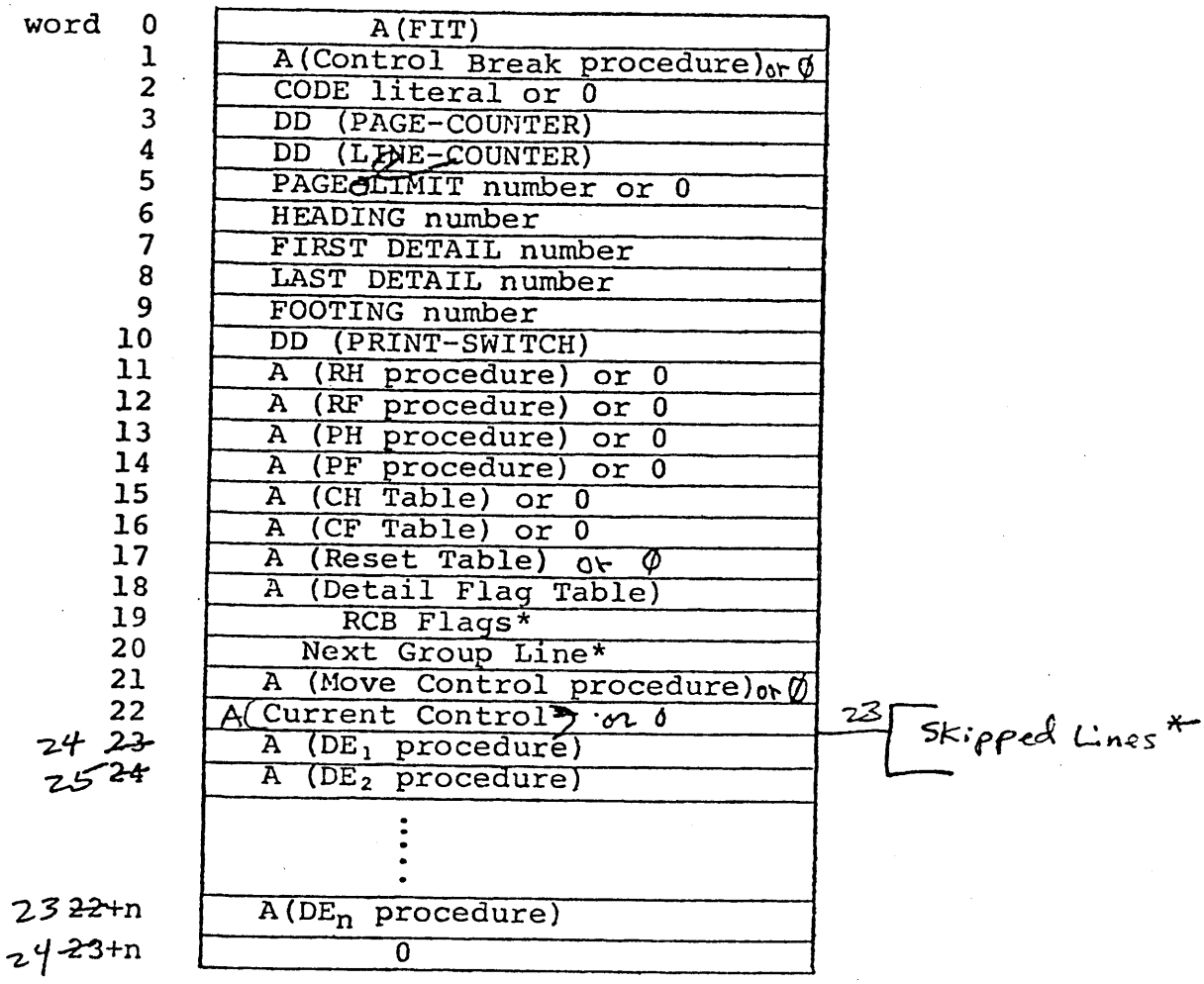
Source Statements	Summary of Compiler Activities
RESET	generate 'MOVE 0 to sum-counter' in RESET SUM routine for that control footing.
Group Indicate	generate a call to C_RWGI with RCB address, DE number, and address of location to skip over move code.
Suppress Printing	generate 'MOVE 1 to print-switch'
Next Group	generate a call to C_RWWT with RCB address and next group information as arguments.
UPON data-name-1	generate 'ADD operand to sum-counter' in the summing routine named by data-name-1.

5.5.16.2 Report Writer System Overview



5.5.16.3 Report Control Block, RCB

A Report Control Block is generated by the compiler for each report name and is used to facilitate the report writer functions.

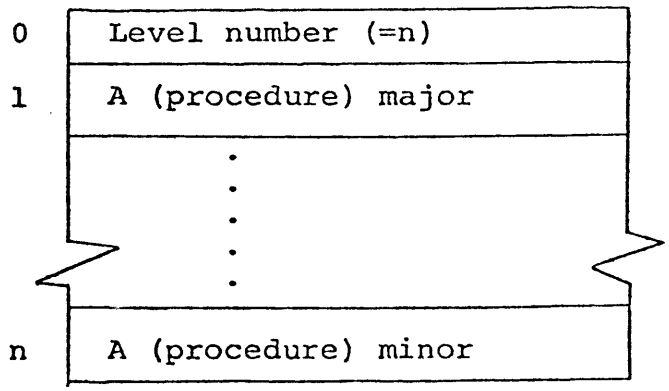


where A( ) = address of
DD( ) = DD address of

*are initialized to zero and are modified by report writer runtime.

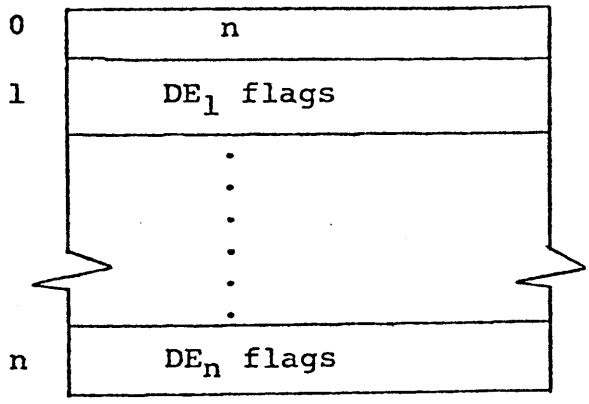
5.5.16.4 CH, CF and Reset Tables

The tables are pointed to by the Report Control Block and contains the addresses of the procedures.



5.5.16.5 Detail Flag Table

The Detail flag table is pointed to by the Report Control Block and contains the flags for all of the detail report groups.



5.5.16.6 Code Generation

- INITIATE

MARK	"14,C_RWIN
LWL	RCB of report-name
CALL	3 + 1

- GENERATE
  - report-name

MARK	"14",C_RWGN
LWL	RCB of report-name
LØ	
CALL	3 + 2

  - detail-name

MARK	"14",C_RWGN
LWL	RCB of report-name
LWL	DE number
CALL	3 + 2

- TERMINATE

MARK	"14",C_RWTM
LWL	RCB of report-name
CALL	3 + 1

- SUPPRESS PRINTING

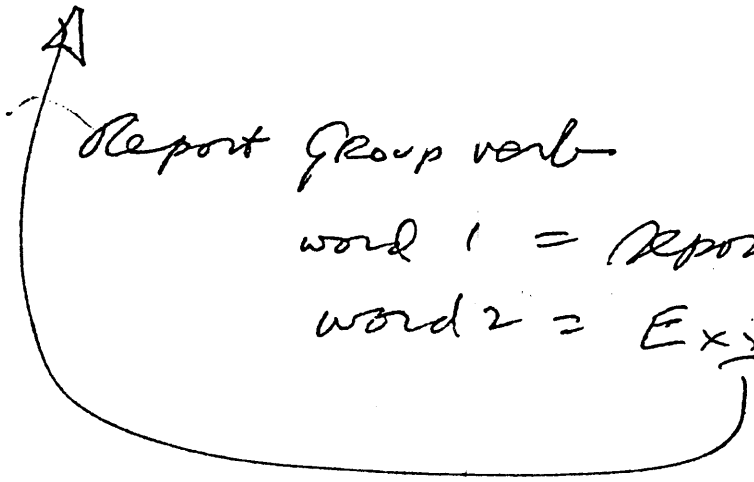
  This statement generates MOVE 1 TO PRINT-SWITCH.

Report Group Header

X'0nn' Reset sum subroutine. where nn = reset control level (base 1).

X'400' Control Break subroutine

X'800' Control save subroutine



Report Group verb

word 1 = Report file pointer

word 2 = Exxx

- At the beginning of report group with relative line body.

MARK	"14,C_RWGR
LWL	RCB of report group
LWL	Sum of relative lines
call	3 + 2

- GROUP INDICATE

MARK	"14",C_RWGI
LWL	RCB of report group
LWL	DD of GROUP INDICATE
CALL	3 + 2

- LINE a call to RWAD is generated prior to ~~before moves for~~ SOURCE and ~~that to a~~ LINE-COUNTER and PAGE-COUNTER are updated ~~before~~ first. A call to RWWT is generated at the LINE clause to perform actual print output.

MARK	"14",C_~~RWWT~~ RWAD
LWL	RCB of report group
LWL	attribute
~~LWL~~	~~integer~~
[LWL	advancing line #] if integer specified
CALL	3 + 3 [+1]

attribute is

Bit 0 = integer specified
1 = NEXT ~~GROUP~~ PAGE
2 = PLUS
7 = NEXT GROUP

MARK	"14", C_RWWT
LWL	RCB of report group
CALL	4