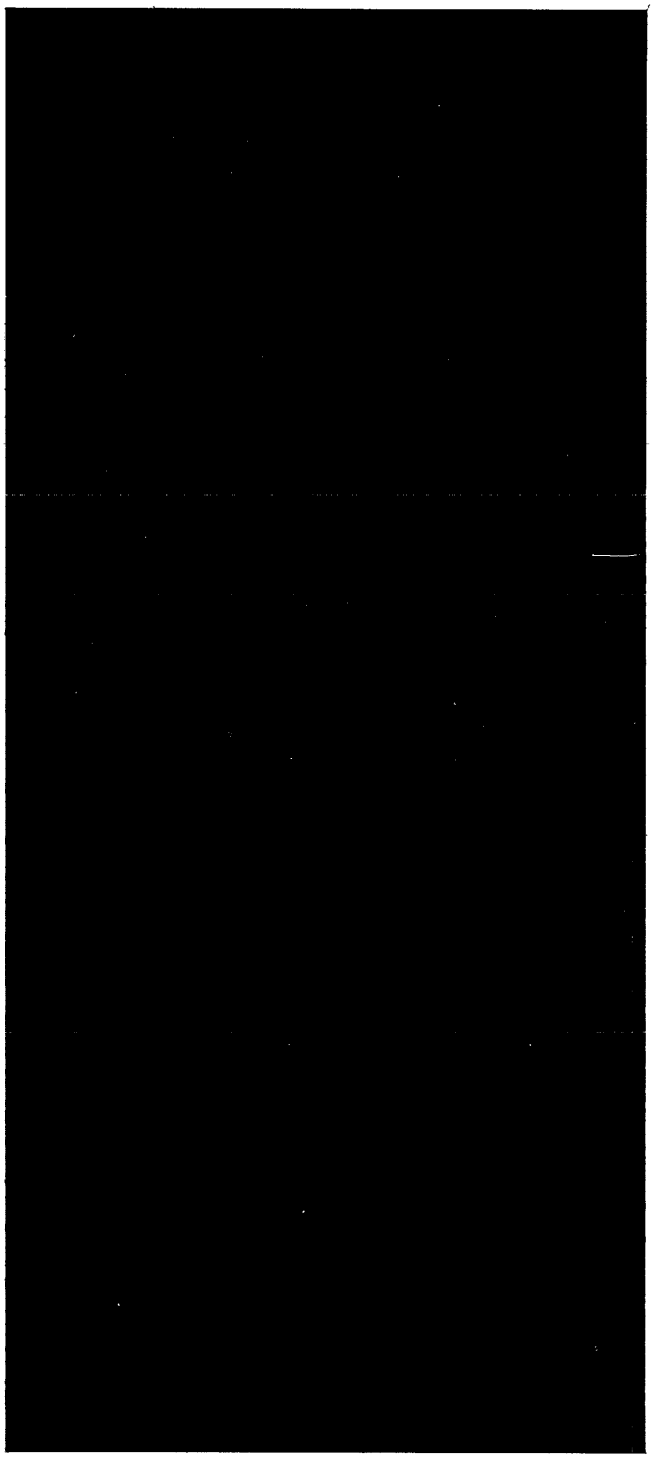
Honeywell

# ALM ASSEMBLER PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68) MULTICS

SOFTWARE



RESTRICTED DISTRIBUTION

Honeywell

# ALM ASSEMBLER PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68) MULTICS

**RESTRICTED DISTRIBUTION**

SUBJECT:

This Document is a Description of the ALM Assembler.

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the Multics Programmers' Manual, Commands and Active Functions (Order No. AG92), Subroutines (Order No. AG93), and Subsystem Writers' Guide (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set which, when complete, will supersede the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96).

THE INFORMATION CONTAINED IN THIS COPYRIGHTED DOCUMENT IS THE EXCLUSIVE PROPERTY OF HONEYWELL INFORMATION SYSTEMS. DISTRIBUTION IS LIMITED TO HONEYWELL EMPLOYEES AND CERTAIN USERS AUTHORIZED TO RECEIVE COPIES. THIS DOCUMENT SHALL NOT BE REPRODUCED OR ITS CONTENTS DISCLOSED TO OTHERS IN WHOLE OR IN PART.

DATE:

February 1975

ORDER NUMBER:

AN63, Rev. 0

# PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005

© 1975, Honeywell Information Systems Inc.
File No.: 2L23

AN63

This PLM assumes that the reader is familiar with the description of the ALM language in the Multics Programmers' Manual, Subsystem Writers' Guide (Order No. AK92).

Throughout this manual, references are frequently made to two of the four manuals that are collectively referred to as the Multics Programmers' Manual (MPM). For convenience, these references will be as follows:

| Document | Referred To In Text As |
|---|---|
| Subroutines (Order No. AG93) | MPM Subroutines |
| Subsystem Writers' Guide (Order No. AK92) | MPM Subsystem Writers' Guide |

CONTENTS

Page

ILLUSTRATIONS

# SECTION I

# OVERVIEW

The ALM (assembly language for Multics) assembler translates a stream of ASCII characters, which represents the source code for a Multics program written in the ALM language, into a Multics standard binary object segment. It optionally produces a listing of the text of the program followed by linkage data, symbol definitions, and a cross-reference table. The assembler is accessed by invoking the alm command with appropriate arguments. (See MPM Subsystem Writers' Guide.)

## FEATURES AVAILABLE

The following is a partial list of the features that are available to the ALM programmer.

1. The entire machine instruction repertoire can be used.
2. Pointer register names are known to the assembler so that "epp4" and "epplp" can be used interchangeably.
3. Data generation and storage allocation pseudo-operations can be used.
4. Variable field literals can be used.
5. Literals with du and dl modification can be used.
6. Complete address field modification can be used.
7. User-defined location counter controls are available.

## FEATURES NOT AVAILABLE

The following is a list of features that are not available to the ALM programmer.

1. Macro and macro-related operations

2. Most listing control pseudo-operations.

NOTE: Throughout the rest of this manual, pseudo-operations will be identified as pseudo-ops.

# SECTION II

# OVERALL OPERATION

The ALM assembler is a two-pass translator. It also includes a post assembly processor that produces Multics intersegment linkage data and symbol table data for the object program. (The linkage format and symbol table data are described in the MPM Subsystem Writers' Guide.) The first and second passes and the post assembly processing are handled by the pass1_, pass2_, and postp2_, procedures. These procedures are essentially administrative procedures that call common subroutines to perform the required functions. A list of these subroutines can be found in Section IV of this manual.

## THE FIRST PASS: pass1_

The primary function of the first pass is to define all symbols internal to the program being assembled. An internal table of symbols and their values are generated. The values are used by pass2_ to generate the variable address fields of the instructions. The predefined system location counters used by pass1_ are initialized as though they were internal symbols within the object program.

Symbols are defined by pass1_ as follows. A program counter is updated as each source instruction is processed. In the case of certain pseudo-ops, the counter is incremented by 1 for a group of instructions. Each value of the program counter represents one binary word in the object segment. The values are used by pass2_ to assign locations to the binary words. Each time pass1_ encounters a new symbol in the label field of a statement, it defines the symbol with the current value of the program counter; the symbol is assigned to the internal symbol table by the subroutine table_.

Symbols that have been defined by pseudo-ops are handled differently depending on the pseudo-op being processed.

1. Internal symbols are defined as described above.

2. External symbols, possibly including a trap pointer, are assembled as a result of the basref and segref pseudo-ops.

3. External symbols defined relative to the stack pointer base (pr6) result from the temp, tempd, and temp8 pseudo-ops.

4. Symbols resulting from the use pseudo-op are defined as internal location counter references and are collected at their corresponding join pseudo-ops.

5. The bool, equ, and link pseudo-ops cause symbols to be defined in terms of expressions given in the arguments of the pseudo-ops.

For more details on pseudo-ops, see the alm command description in the MPM Subsystem Writers' Guide. If a symbol cannot be defined in pass1_, pass2_ will attempt to define it.

Literals are also processed by pass1_. They are evaluated and assigned to the pool of literals (literal table). They are not assigned a specific location until pass2_ because the length of the object program up to its end statement is not known during pass1_. No literal appears twice. The pool is maintained in order by pointers associated with each literal. (See Section III of this manual for more detail.)

The first pass does not produce an intermediate or collation segment; the second pass rereads the input stream from the beginning. A list, ordered by the value of the program counter after each source statement has been processed, is generated by pass1_. This list is checked for correspondence by pass2_. If it does not correspond, a phase error is signalled. The format of each block in the list is given in Figure 2-1 below.

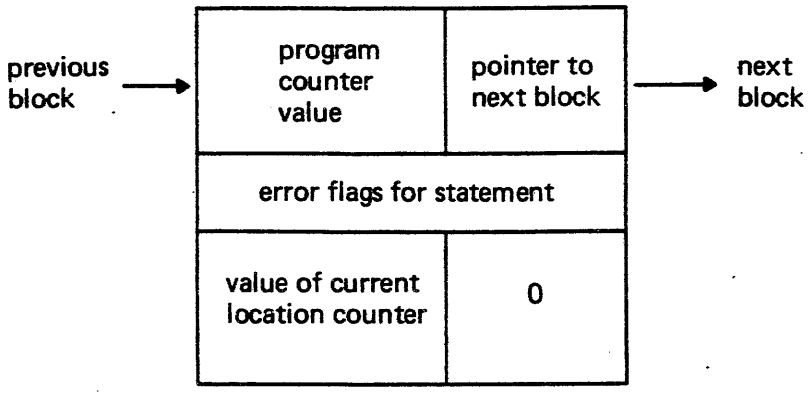



Figure 2-1. Assembler List Maintenance Block

THE SECOND PASS: pass2_

The second pass of the ALM assembler generates the binary output associated with each input statement. It also generates the assembly listing and completes information (literals, segdef names, etc.) to be generated by the post assembly processor postp2_.

The binary code for a normal instruction is generated as follows. The oplook_ subroutine is called to find the binary equivalent of the symbolic operation code in a table (alphabetically ordered) that is associated with the oplook_ procedure. The binary value of the variable address field is determined by the varevl_ subroutine. The operation code and address field are assembled in a binary word equivalent to the symbolic instruction. When a newline character (ASCII code 012) is encountered, prwrd_ is called to generate the listing.

The binary code for pseudo-ops is generated as follows. First the symbols are evaluated to make sure their values are the same as those determined by pass1_. System pseudo-ops (call, push, return, entry, etc.) are expanded to generate the special code they imply. Single word pseudo-ops (zero, setlp, vfd, etc.) are evaluated individually and their binary values generated.

Since many pseudo-ops generate more than one word of equivalent printed output, the listing for a given source line is maintained in a buffer by the prwrd_ subroutine until a newline character indicating the end of a source statement is encountered.

The pass2_ procedure checks the input stream for syntax errors and monitors the assembler itself for possible malfunctions. The errors that were detected by pass1_ are transmitted to pass2_, which may add to or duplicate the errors signalled for a given statement. If a phase error occurs (pass1_ and pass2_ program and/or current location counters do not match), the assembly is aborted.

THE POST ASSEMBLY PROCESSOR: postp2_

The post assembly processor for the ALM assembler serves two major functions:

1. It processes and generates all the definition information (i.e., the contents of the definition section of the object segment).

2. It generates the linkage block and the symbol table header for the object segment.

For the text portion, postp2_ generates binary output for:

1. literals

2. entry points

For the definition section, postp2_ generates binary output for:

1. segdef definitions

2. external names

3. trap-pointer words

4. type-pair words

5. internal expression words

The order of output of this information is important because the previous items are referenced by the later ones and thus the binary locations must be known.

For the linkage block, the post assembly processor writes out the linkage header and the linkage pairs.

The post assembly processor has the overall task of defining the locations for all the information it puts out. Relative pointers are the only connection within the assembler among the linkage pairs, expression words, type-pair words, trap-pointer words, external names, and segment names.

# SECTION III

# DETAILS OF INTERNAL OPERATION

## ACCESS TO THE ASSEMBLER

The assembler is entered from the alm command by a call to the alm_6180_ procedure. This central procedure calls the main programs of the assembler and reports assembly on the user_output switch.

## STRUCTURE OF THE DATA BASE, glpl_

Every item of information maintained by the assembler is kept in a list structure. The total list structure is accessed via entry points of glpl_ (general list processing language).

## ASSIGNMENT TABLE STRUCTURE, table_

The table of symbols (assignment table) is maintained as a list of all symbols that have been defined within the program. The list and structure is managed by the table_ procedure.

The assigned symbols fall into a number of classes (eight at present), which include internal, external, and stack. The class is indicated in the flag field of the table entry (see Figure 3-1 below). A given symbol can be assigned to more than one class with no conflict since the class of symbol is recognized by its contextual use.

The symbols are constructed from a character string (up to 31 ASCII characters) and the count of the string.

The assignment table is not one long list of symbols, but 211 (a convenient prime) lists. The symbols are distributed randomly among these lists according to the following procedure.

The first word of the symbol is taken; a constant is added to it; the resultant value mod 211 is extracted. This value specifies which of the 211 lists is to be searched for the symbol. The list entries for internal symbols and multiple location counter symbols are, respectively, the three-word or five-word blocks pictured in Figures 3-1 and 3-2 below.

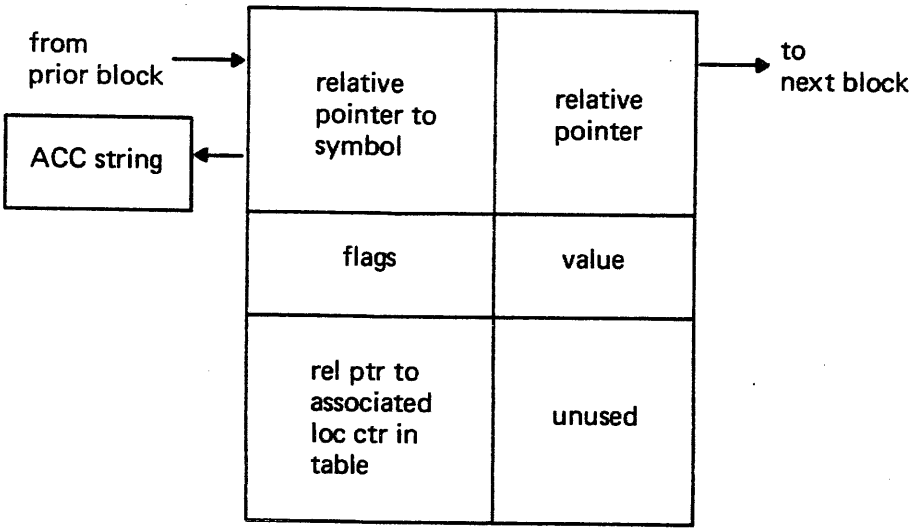



Figure 3-1. Internal Symbol Block

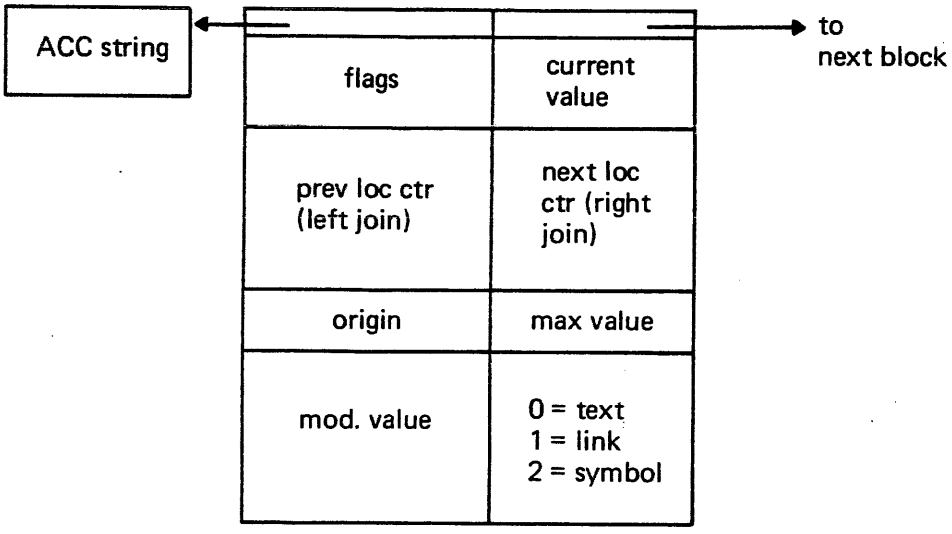



Figure 3-2. Multiple Location Symbol Block

The following list explains the terms used in Figures 3-1 and 3-2.

value — is an 18-bit number that is the value of the program counter when the symbol was encountered.

flags — is a 3-bit class number and a 15-bit indicator field.

Class number:

| | | |
|---|---|---|
| clunk | 0 | undefined |
| clint | 1 | internal |
| clept | 2 | external |
| clbas | 3 | pointer register (unused) |
| clstk | 4 | stack |
| clndx | 5 | index register (unused) |
| clmlc | 6 | multiple location counter |
| | 7 | unused |

Note: If a symbol belongs to more than one class, a different block for each class will appear in the list.

Indicator bits:

| | | |
|---|---|---|
| fdef | 00001 | symbol defined |
| fmal | 00002 | multiple definition |
| fpks | 00004 | in phase error |
| fset | 00010 | symbol resettable |
| frel | 00020 | relocatable |
| fabs | 00040 | absolute |
| fbol | 00100 | Boolean |

| | | |
|---|---|---|
| fcom | 00200 | in common |
| find | 00400 | value is indirect reference |

Note: Only nine of the 15 are used; they may be ORed together.

## EVALUATION OF THE VARIABLE (OPERAND/ADDRESS) FIELD

The variable field is evaluated by two subroutines, varevl_ and expevl_. Instructions and pseudo-ops are treated differently. The variable field of instructions may contain an external reference followed by an internal expression or simply an internal expression, either of which may be followed by a comma and modifier. Some pseudo-ops are constructed like normal instructions while others have specific requirements, e.g., the evaluation of internal or Boolean expressions. The varevl_ subroutine handles the evaluation and formatting of external references. Arithmetic or Boolean expressions are evaluated by expevl_, which may be called either by varevl_ or the main passes.

The values of symbols and expressions may be either absolute or relative to some location counter (lc). The operands of the arithmetic operators are restricted to the combinations in the following list:

| operand 1 | operator | operand 2 | = | result |
|---|---|---|---|---|
| absolute | + | absolute | = | absolute |
| relative to lc | + | absolute | = | relative to lc |
| absolute | + | relative to lc | = | relative to lc |
| absolute | - | absolute | = | absolute |
| relative to lc | - | absolute | = | relative to lc |
| relative to lc | - | relative to lc | = | absolute |
| absolute | * | absolute | = | absolute |
| absolute | / | absolute | = | absolute |
| -none- | (unary)- | absolute | = | absolute |

Expressions evaluated by pass1_, such as those appearing in equ and org pseudo-ops, must be absolute.

Procedure varevl_ may be called in three cases:

1. to evaluate a full address field (possibly external, possibly literal).

2. to evaluate a complete internal expression with no modifier.

3. to evaluate a pure Boolean expression with no modifier.

If the address field is an external reference, varevl_ checks for an address in one of the following formats:

EXTERNAL

1. <seg>|[xname]±inexp,mod

2. <seg>|inexp,mod

3. pr|[xname]±inexp,mod

4. pr|inexp,mod

5. segref_name±inexp,mod (segref_name or basref_name)

6. stackname±inexp,mod

7. inexp,mod

8. =literal,mod

The first six examples above are references external to the segment being assembled and cause varevl_ to turn on bit 29 of the instruction. Examples 1, 2, 3, and 5 cause entries to be made in the link, type block, and external name lists and force the instructions to be referenced through the linkage segment (e.g., use pr4 = lp with bit 29 set on). Conversely, examples 4 and 6 cause reference to be made directly to the segment without making entries in any of the assembler's tables or lists. The internal expressions, the literals, and the modifiers are evaluated by expevl_, litevl_, and modevl_, respectively.

Subroutine expevl_ is responsible for evaluating arithmetic and Boolean expressions consisting entirely of symbols, numbers, operators (+, -, *, and /), parentheses, and expression terminators (e.g., blank, comma, semicolon, etc.). Parentheses bracket subexpressions and they may be nested to any level up to 100 pairs. Expressions are evaluated by a stack technique in

which the operators and delimiters are examined in order of precedence as follows:

| Name | Order | Meaning |
|---|---|---|
| lndt | 1 | left end terminator |
| rndt | 2 | right end terminator (all others) |
| ( | 3 | left parenthesis |
| ) | 4 | right parenthesis |
| + | 5 | binary plus or Boolean OR |
| - | 5 | binary minus or exclusive OR |
| * | 6 | binary multiply or AND |
| / | 6 | binary divide or AND NOT |
| neg | 7 | Unary minus |
| not | 7 | Unary NOT |

Any unknown operator is given a precedence of 2, which is synonymous with the right end terminator. An excessive right parenthesis is treated as a terminator and if the field ends with an unbalanced left parenthesis, an error is reported and the field is set to zero.

A literal is recognized by varevl_ by the presence of the equal sign (=) in the first position of the variable field and causes varevl_ to invoke subroutine litevl_. Subroutine litevl_ determines the specific type of literal (e.g., its, itp, vfd, etc.) and invokes a particular data field evaluator to process the field. The literals thus evaluated are then placed in a list of literals (the literal pool) in such a way that no literal appears twice.

The structure of a block in the literal list is shown in Figure 3-3 below.

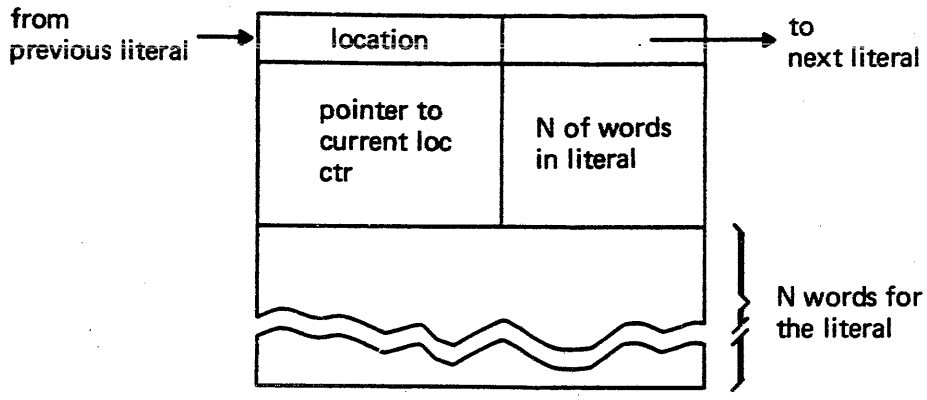



Figure 3-3. Literal List Block Structure

The location of the literal is not assigned until pass2_ when the literal is actually used in the object code. All multiword literals are located at an even location. Single word literals may be located at an odd or even location and are placed in the table so as to fill in any "holes" between multiword entries first. Subroutine litevl_ also checks for the du and dl modifiers and returns the proper address and modifier in such uses. No entry is made in the literal list if du or dl is specified.

The modifier field is evaluated by subroutine modevl_. All types of symbolic (named) modifiers are allowed including the numeric modifiers.

## LINKAGE GENERATION AND LIST MAINTENANCE

The information required for intersegment communication is generated and maintained by the following eight entries.

lstman_$blkasn

lstman_$calser (obsolete)

lstman_$eptasn

lstman_$lnkasn

lstman_$namasn

lstman_$outasn (obsolete)

lstman_$sdfasn

lstman_$trpasn

The namasn entry of lstman_ is responsible for assigning external (segment and location) names to the namlst list and making sure that those names are entered only once. The structure of a namlst block is as shown in Figure 3-4 below.

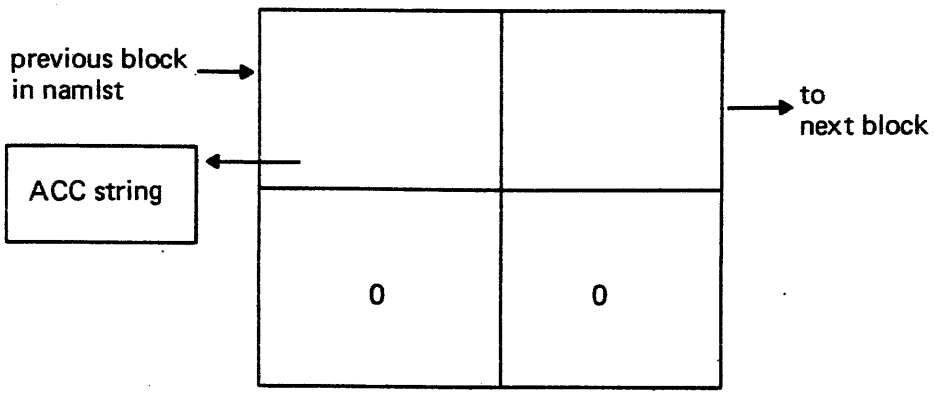



Figure 3-4. Structure of a namlst Block

The blkasn entry of lstman_ is responsible for maintaining the list of type-pair blocks, blklst. A given unique type-pair block is entered only once in the list. The format of a type-pair block is as shown in Figure 3-5 below.

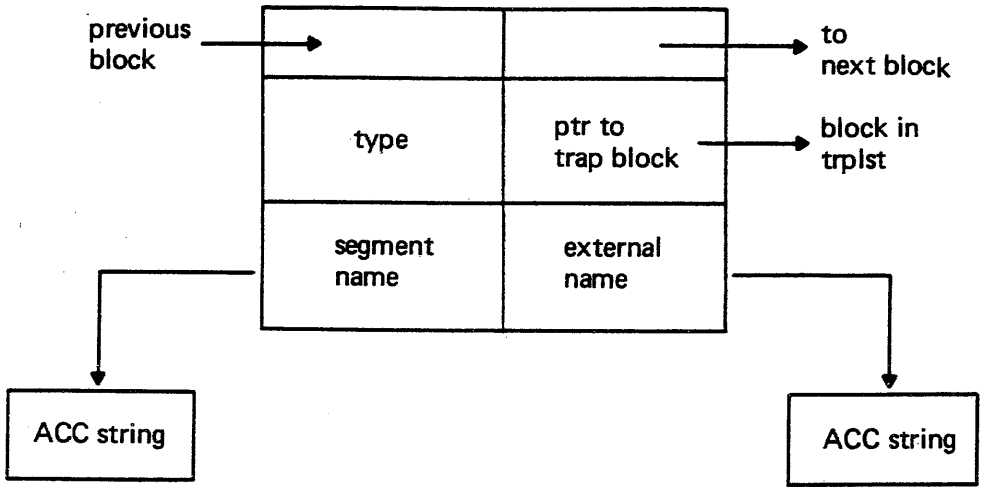



Figure 3-5. Structure of a Type-Pair Block

The trpasn entry of lstman_ is responsible for maintaining the list of trap-pointer words, trplst. No trap-pointer word is entered more than once in the list. A block in this list is constructed as shown in Figure 3-6 below.

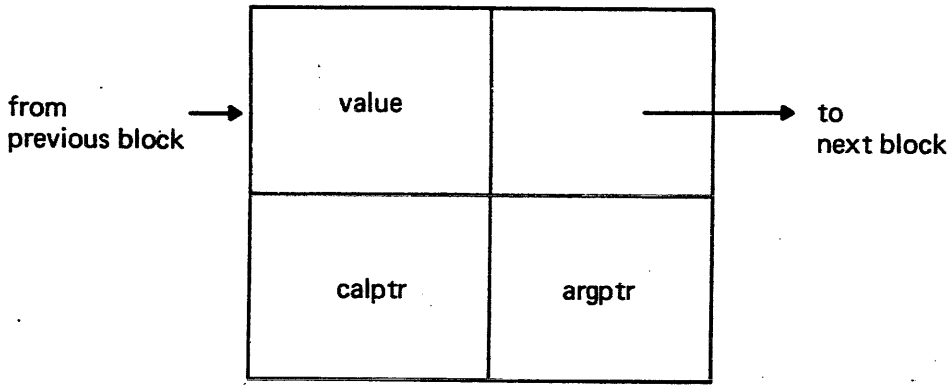



Figure 3-6. Structure of a Trap-Pointer List Entry

The following list explains the terms used in Figure 3-6.

calptr — is the link number of the call to the trap routine

argptr — is the link number of the associated argument list

value — is assigned by postp2_ as the relative location of the trap-pointer word when generated

The lnkasn entry to lstman_ generates and maintains lists of linkage data. The explst is a list of internal expressions. An entry in the explst is a block of the format shown in Figure 3-7 below.

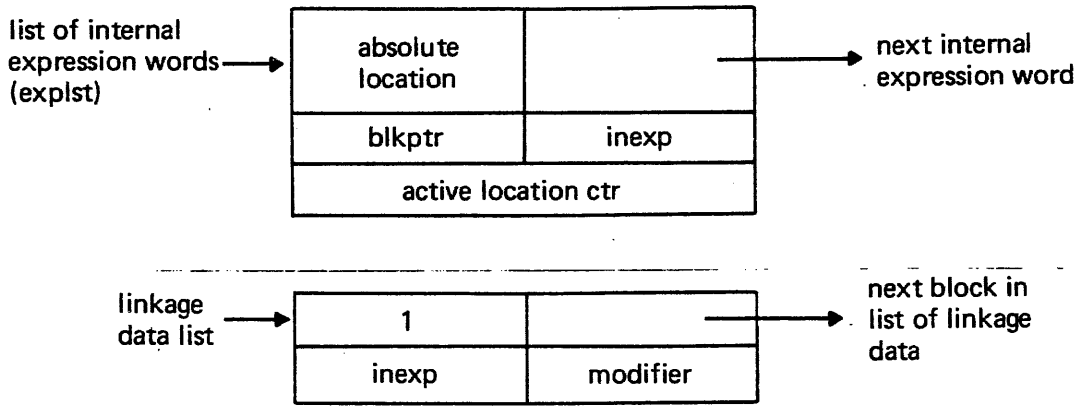



Figure 3-7. Structure of a Normal explst Block

The following list explains the terms used in Figure 3-7.

blkptr — points to the associated type-pair data in the blklst entry

modifier — is the address modifier in the original instruction

The eptasn entry to lstman_ assigns entry points to the link structure list. A block in the list has the format shown in Figure 3-8 below.

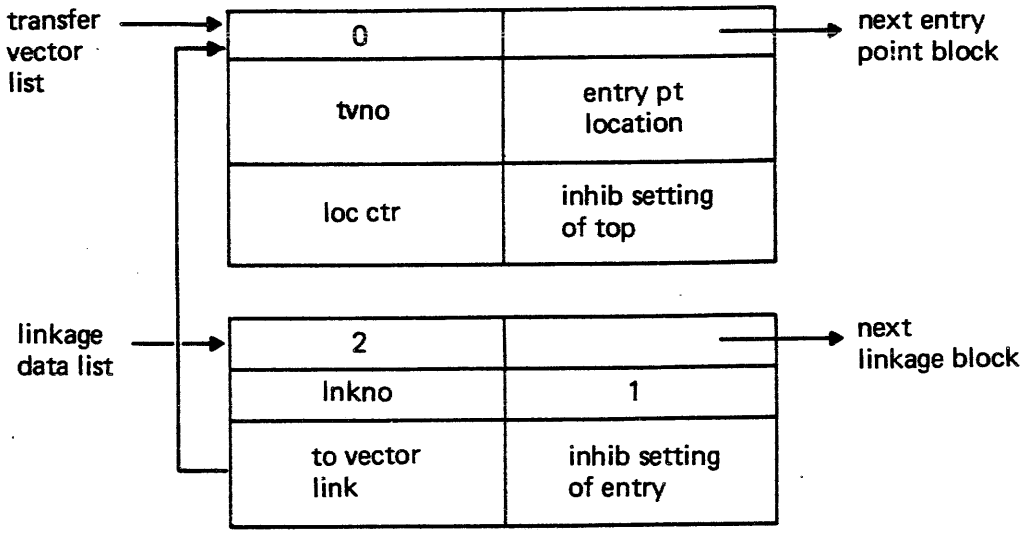



Figure 3-8. Structure of an Entry Point Interlude Block

Information about external names (segdefs) is entered by the sdfasn entry to lstman_ into the definition list, sdflst. An entry in sdflst is formatted as shown in Figure 3-9 below.

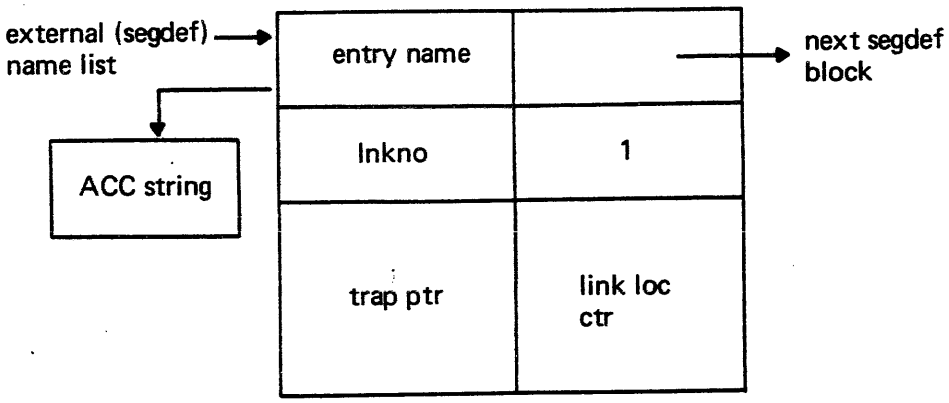



Figure 3-9. Structure of a segdef Block

The overall structure of the linkage information in the above lists implies that the order of definition of final values in the linkage section of the object segment must be 1) names, 2) trap-pointer words, 3) type-pair words, and 4) the internal expression words.

The lists just mentioned (namlst, blklst, trplst, explst, lnklst, sdflst, outlst) are interconnected with pointers much like words are linked in the text and linkage portions of the object segment. For example, if the instruction:

```
lda <sega>|[namea ]+inexp,mod
```

were assembled, the resulting object code in terms of relative pointers would be as shown in Figure 3-10 below.

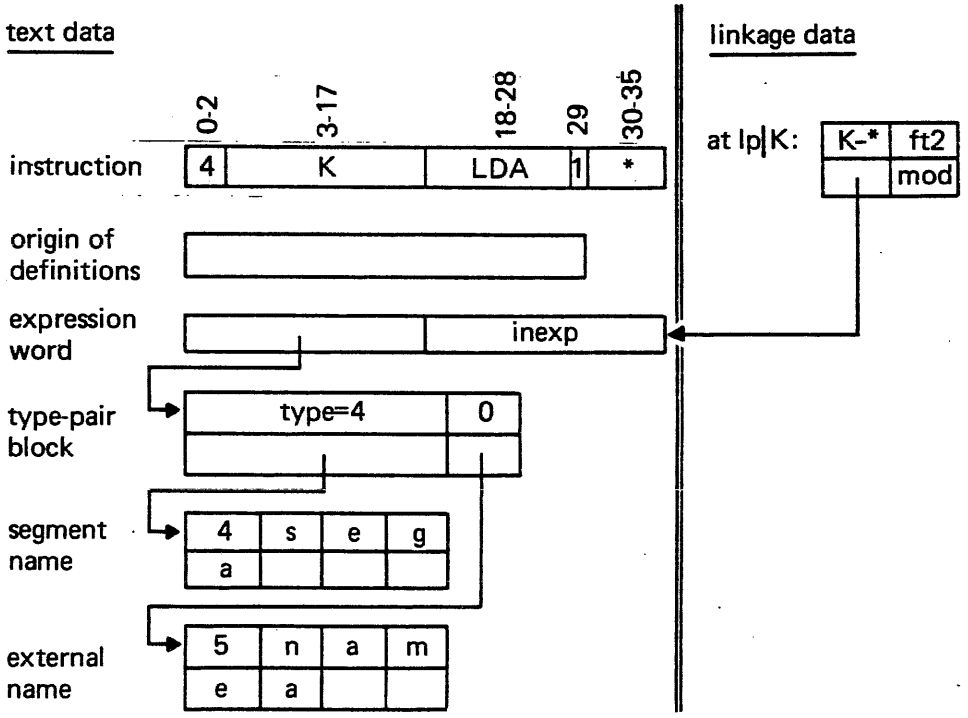



Figure 3-10. Schematic Object Code for "lda" Instruction

## EVALUATING DATA FIELDS (CONSTANTS)

Literals and pseudo-ops that produce constants are evaluated by calls to subroutines especially designed for that purpose. Five basic types of constants are allowed: ASCII, BCD, decimal, octal, and vfd. The first two types of fields are evaluated by the subroutine ascevl_. The last three are evaluated by decevl_, octevl_, and vfdevl_, respectively.

Subroutine ascevl_ is used to evaluate the variable field of the acc, aci, and bci pseudo-ops. The variable (character) field is bounded by a nonblank character. No more than 40 words (159 characters for the acc pseudo-op, 160 characters for the aci pseudo-op, and 240 words for the bci pseudo-op may be generated. Literal fields containing ASCII characters are evaluated by either litevl_ or decevl_ depending on their format.

Subroutine decevl_ is used to evaluate the operand field of a dec pseudo-op and to evaluate a decimal field. The field may be integer, fixed, floating, or double precision with the usual conventions for the (decimal) point, and the letters B, D, and E. ASCII literals of the form "=naxxx" (e.g., "=3aSYM") are also evaluated by decevl_. Decimal words are manipulated by the various entry points in decsub_, which handles the decimal values in "triple" precision, one word for the exponent and two words for the mantissa.

Subroutine octevl_ evaluates the subfields in the operand field of the oct pseudo-op and evaluates octal literals of the form "=oxxx (e.g., "=o675432"). No signs are allowed, and a check is made for more than 12 characters or a nonoctal digit in the data field.

Subroutine vfdevl_ evaluates the entire field of a vfd pseudo-op and also evaluates vfd literals of the form "=vxxx". Three types of vfd subfields are allowed: arithmetic, Boolean (octal), and ASCII. The arithmetic and Boolean subfields are processed by expevl_, while the ASCII subfields are evaluated by vfdevl_.

INPUT

All input to the assembler is processed by inputs_$next and inputs_$nxtnb. These entry points read the next character from the input segment and store the ASCII character with its corresponding code type for the user. The inputs_$next entry handles any legitimate Multics character. The inputs_$nxtnb entry reads any character except a space or horizontal tab. If the previous character was a statement terminator (newline, ASCII code 012; semicolon, 073; or carriage return, 015), inputs_$nxtnb simply returns to the calling program.

The entry getid collects the next complete symbol identifier (up to 31 alphanumeric characters, the first of which must be a letter, period, or underline) and associated break character (tab, blank, comma, etc.). The entry getnam collects the next symbol, which may contain any legitimate character. The subroutine setid_, like getid_, collects the identifier but also assigns it to free storage for subsequent use by the assembler. The procedure getid_ is a transfer vector to access these three entries (getid_, getnam, and setid_).

OUTPUT

Output operations consist of the generation of an object segment and an optional listing segment. Output for the object segment and the processing of the relocation bits are done as follows:

1. The text is written directly into the final object segment, as it is being assembled.

2. The relocation bits for the text, linkage, and symbol as well as the object linkage and symbol words are temporarily stacked in the assembler's scratch segment. All the entries in the subroutine object_ manipulate this scratch segment.

3. After postp2_ has completed its processing, subroutine pakbit_ is called to process all the relocation bits.

4. Subroutine merge_ then appends all the object linkage and symbol words, including the packed words of relocation bits from pakbit_, to complete the object segment being assembled.

## OUTPUT OF THE LISTING

The major part of processing the optional listing segment is done by the prwrd_ subroutine. During pass2_, each input character that is read is placed in a one character per word buffer. When a newline character (ASCII code 012) is encountered, prwrd_$source_only is called to combine the input statement from the source segment with the printed equivalent of the binary word. Whenever a word of binary output is generated, a call is made to prwrd_ to convert that word and its associated location and error flags to printable characters and place the result into the output segment. Subroutines prlst_ and prnam_ are also used to generate printable output for the listing segment. The prlst_ program is used to insert a line of noninput characters (e.g., headings, etc.) into the listing segment. The prnam_ program generates the characters for ASCII names (e.g., segment names, entry points, etc.) that are inserted in the object segment.

Error comments are transmitted to the user's error_output I/O switch (e.g., terminal) and placed in the (optional) listing segment by subroutine prnter_. This program calls ioa_ and prlst_ to write out the actual line of text. (For a description of ioa_, see the MPM Subroutines.)

## UTILITY PROGRAMS

The assembler uses a number of smaller subroutines to perform frequently executed tasks. These programs are the following:

1. glpl_ is a set of routines for performing fast list processing.

2. utils_ is a set of routines for performing miscellaneous tasks that were required to support the FORTRAN in which the assembler was originally written.

All programs of the assembler use a common data segment named eb_data_. This data segment contains pure information in its text portion and impure information in its linkage portion. For details, consult the segment and/or the calling procedures.

# SECTION IV

## SUBROUTINE SUMMARY

The following is a list of the various subroutines of the assembler, ordered by function:

1. Main control programs

| | |
|---|---|
| alm | command interface |
| alm_6180_ | drives the major programs of the assembler |
| pass1_ | first pass of ALM |
| postp1_ | post processor for joining multiple location counter |
| pass2_ | second pass of ALM |
| postp2_ | post processor for linkage and symbol data |
| pakbit_ | packs relocation bits |
| merge_ | appends linkage and symbol data to the object segment |
| alm_eis_parse_ | handles EIS multiword instruction pseudo-ops |

2. Assignment table maintenance

| | |
|---|---|
| table_ | assigns or searches the internal symbol table |

3. Variable field evaluation

| | |
|---|---|
| varevl_ | evaluates an operand |
| expevl_ | evaluates a complete expression |
| modevl_ | determines the address modifier |
| litevl_ | evaluates literals |

4. Processing list of linkage data

| | |
|---|---|
| lstman_$namasn | assigns a symbol to the external name list |
| lstman_$blkasn | assigns a type-pair block |
| lstman_$lnkasn | assigns a linkage-pair block |
| lstman_$trpasn | assigns a trap-pointer block |
| lstman_$outasn | assigns a mastermode/executeonly call-out node (obsolete) |
| lstman_$calser | searches for a mastermode/executeonly call-out node (obsolete) |
| lstman_$eptasn | assigns an entry point node |
| lstman_$sdfasn | assigns a segdef node |
| alm_definitions_ | puts out the symbolic definition region |

5. Data Generating Subroutines

| | |
|---|---|
| ascevl_ | evaluates acc, aci, and bci variable fields |
| decevl_ | evaluates decimal fields |
| octevl_ | evaluates octal fields |
| vfdevl_ | evaluates vfd fields |

6. General Utility Programs

| | |
|---|---|
| getid_ | collects the characters of an identifier |
| setid_ | same as getid_ |
| oplook_ | searches for an op-code or pseudo-op symbol |
| utils_ | performs high-speed logical operations |
| glpl_ | manages the "free storage" segment |
| inputs_ | reads the source segment |

7. Printing-associated routines

| | |
|---|---|
| prwrd_ | converts a binary word to printable characters |
| prlst_ | inserts a line of noninput into the listing |
| prnter_ | reports an error message on the error_output I/O switch and in the listing |
| prnam_ | converts and deposits printable characters into the listing segment from a binary word containing ASCII |

8. Manage binary words for the output segment

| | |
|---|---|
| putout_ | determines the portion of the object segment and calls the appropriate subroutine to write out a list or single word into the object segment handler |
| putxt_ | writes a binary word and the associated relocation bits for the text portion of the object segment |
| pulnk_ | writes a binary word and the associated relocation bits for the linkage portion of the object segment |

| | |
|---|---|
| pusmb_ | writes a binary word and the associated relocation bits for the symbol portion of the object segment |
| object_ | manages a rigidly formatted scratch segment of binary data |
| alm_source_map_ | creates the source map for the object segment |
| alm_cross_reference_ | formats and prints the cross-reference table |
| alm_include_file_ | manages include files |

9. Relocation bit processor

| | |
|---|---|
| getbit_ | determines the relocation bits from the components of an assembled binary word |
| pakbit_ | collects and packs the relocation bits for the object segment |

10. Symbol table management

| | |
|---|---|
| sthedr_ (obsolete) | template for symbol table header |
| new_sthedr_ | new version of sthedr |

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68) MULTICS ALM ASSEMBLER PROGRAM LOGIC MANUAL

ORDER NO. AN63, REV. 0

DATED FEBRUARY 1975

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell