MACRO PROCESSOR FOR HP 2100A ASSEMBLER

MACRO PROCESSOR FOR HP 2100A ASSEMBLER

By

KONDAPURAM SESHACHAR SAMPATHKUMARAN, M. Sc.

A

Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the degree

Master of Science

McMaster University

May 1977

MASTER OF SCIENCE (1977)
(Computation)

McMaster University
Hamilton, Ontario

TITLE: MACRO PROCESSOR FOR HP 2100A ASSEMBLER

AUTHOR: Kondapuram Seshachar Sampathkumaran
B.Sc. (Bangalore University)
M.Sc. (Bangalore University)

SUPERVISOR: Professor Nicholas Solnsteff

NUMBER OF PAGES: viii, 104

ABSTRACT

A Macro Processor is implemented in PILOT (Purdue Instructional Language for Writing Operating systems and Translators) for HP 2100A DOS-M Assembler. The Macro Processor has the capability to handle Macro calls within macros, Macro definitions within other Macro definitions, conditional Macro expansion and the string operation of canatenation. A simple set of Macros for Fundamental Structured programming constructs is provided. The project also demonstrates, how an Intermediate-level language like PILOT can be used to implement system software. Experiments with a new programming philosophy for the writing of structured programs are also described.

ACKCONWLEDGEMENT

The author is grateful to Prof. N. Solnsteff for being my project adviser and his constant encouragement throughout the course of this investigation. Grateful thanks are also due to Prof. G. L. Keech and Prof. K. A. Redish for serving on defense committee. Special thanks are also due to Prof. R. A. Rink for constant encouragement throughout my stay at McMaster University. Finally thanks are due to Mr. Chris Bryce for help during software implementation.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

MACRO PROCESSOR

1.0 INTRODUCTION:

The term Macro is derived from Greek makros, meaning long or large. The term Macro is used in scientific literature in various contexts, for example, macroscopic, meaning visible to the naked eye. In the field of computer science the term Macro is used to denote an instruction, the macro-instruction, which generates a long sequence of machine-instructions. The macro-instruction concept has been widely used in assembly systems since as early as the nineteen fifties [GRE 59].

1.1 MACRO INSTRUCTION:

In assembly language programs there are often several occurrences of the same block of assembly language instructions. In this situation the concept of a Macro is useful. An abbreviation can be given to name the repetitive block or sequence of assembly language instructions. In the course of an assembly language program, the occurrence of the abbreviation for the sequence of assembly language instructions is replaced by the entire sequence of instructions.

The computer software which facilitates this type of activity is known as a Macro processor. As an example, consider the repetition of a sequence of assembly language instructions.

```
.
.
.
.
.
.
LDA A
ADA B
STA C
.
.
.
.
.
.
LDA A
ADA B
STA C
.
.
.
.
.
.
```

The sequence of instructions

```
LDA A
ADA B
STA C
```

appears twice in the course of the program. A name can be associated with this sequence of assembly language instructions, and reference to this name in the assembly language program results in substitution of the above sequence of instructions in place of the name. In the above example, we can give a name ADD to the sequence of instructions and then the following input to the macro-processor: will result in the output.

Input to the Macro-processor	Output from the Macro-processor
.	.
.	.
.	.
.	.
.	.
.	.
ADD	LDA A
	ADA B
	STA C
.	.
.	.
.	.
.	.
.	.
.	.
ADD	LDA A
	ADA B
	STA C
.	.
.	.
.	.
.	.
.	.
.	.

In general, a macro expansion provides us with the means to abbreviate repeated sequences of assembly language instructions. The manner in which such abbreviations can be defined is outlined below.

1.2 <u>MACRO DEFINITION FORMAT</u>

Indication of the start of the Macro definition.	MACRO
Name for the sequence of instructions.	
Actual sequence of instructions.	
Indication of the end of the sequence of instructions for which this name stands.	MEND

In an assembly language the statements MACRO and MEND would be called pseudo-operations or pseudo-ops. The pseudo-op MACRO indicates the beginning of a Macro definiton. The line following this pseudo-op is the name to be referred whenever the sequence of instructions for which this name stands is to be inserted in the assembly language program. This name is known as the macro name. The sequence of instructions following the macro name is known as the macro definition body. The pseudo-op MEND indicates the end of the sequence of instructions for this macro name. The macro name, once defined, can be used like an operation code in the assembly language program or, to be explicit, the macro name behaves like an assembly language instruction.

The appearance of a macro name in the assembly language program is known as a macro call. The action of inserting the sequence of instructions wherever the macro call occurs is known as macro expansion. The process of specifying the format for abbreviating the sequence of instructions is referred to as macro definition. The process of insertion of a sequence of instructions using a macro facility, is similar to insertion of open subroutines in many of the higher level language programs. The main difference is that, in case of open subroutine insertion takes place usually at the loading time, whereas in the case of Macro, insertion takes place before or during the translation of assembly language program. If this process of insertion of sequence of instructions takes place during the assembly time, the assembler is termed an Macro-assembler [BRO 74] . The logical action of a Macro-Processor can be viewed as below.

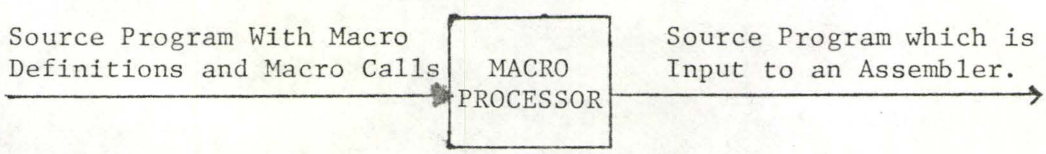


Fig. 1: Logical action of a Macro Processor

The Macro Processor is not restricted to Assembly Language Systems alone. Macro Processors can be designed for any programming language. According to Brown [BRO 74], Macro Processors can be classified as either Special Purpose or General Purpose. A special purpose Macro Processor is designed to process Macros written in a particular base language. A programming language L is referred to as a base language, for it is the base on which Macros are built and where L is a programming language. Historically Macro Processors are associated with a particular Assembly language. A general purpose Macro processor is designed to work on any strings of characters and is thereby suitable for any base language. The Macro Processor implemented in the project falls into the category of special purpose Macro Processors and accepts Macros written in Hewlett Packard Assembly Language.

1.3 MACRO DEFINITION AND MACRO CALL ARGUMENTS:

In the previous examples presented, each occurrence of the abbreviated sequence of instructions involved the same operands. This is an unrealistic situation. A simple modification to situation might appear as follows:

```
 .
 .
 .
LDA PQ
ADA RS
STA PQ
 .
 .
 .
LDA A
ADA B
STA C
 .
 .
 .
```

It can be seen that here two sequences of instructions are identical except for the operand fields. An extension to the solution presented in the previous example will take care of this situation, namely, we give a name to the set of instructions along with general operands. The sequence of instructions in the body of the macro definition will have their operands in terms of the operands specified in the macro name. The operands specified in the macro name are referred to as <u>macro-instruction arguments</u> or <u>dummy arguments</u>. The first character of the macro instruction argument is an ampersand (&). This special character is used to distinguish macro instruction arguments from assembly language symbols. Consider the following example to demonstrate this situation.

Input to the Macro Processor

```
MACRO
ADD   &ARG1, &ARG2
LDA   &ARG1
ADA   &ARG2
STA   &ARG1
MEND
 .
 .
 .
ADD    PQ, RS
 .
 .
 .
ADD    A, B
 .
 .
 .
 .
 .
 .
```

Output from the Macro Processor

```
 .
 .
 .
 .
 .
 .
 .
LDA PQ
ADA RS
STA PQ
 .
 .
 .
 .
LDA A
ADA B
STA A
 .
 .
 .
```

In the above example, the first call to the macro ADD uses PQ, RS as operands and the second call to the same macro ADD uses A, B as operands. The operands used in the macro call are sometimes referred to as macro call arguments.

The arguments in the macro call can be specified in two ways. The strategy wherein the macro call arguments are matched with the macro-instruction or definition arguments according to the order in which they appear is known as positional argument specification. Another strategy is one in which the macro-instruction arguments are referred to both by name and by position. This strategy of specification of the arguments is known as keyword argument specification. The keyword argument specification has the advantage of selective argument specification. The following example illustrates the difference between the two types of specification.

Positional argument specifications ADD PQ, RS

PQ, RS correspond to the first and second macro instruction or definition argument &ARG1 and &ARG2, respectively.

Keyword argument specification ADD &ARG1=A, &ARG2=B

Here it is explicitly specified that A refers to the first macro instrucion or definition argument and B refers to the second macro instruction or definition argument and also their position is explicitly specified. We have taken the approach of positional argument specification in this study.

1.4 CONDITIONAL MACRO EXPANSION PSEUDO-OPS:

The conditional macro expansion pseudo-ops aid in conditional selection of sequences of instructions within the body of macro definitions. The two conditional macro expansion pseudo-ops considered in this study are AIF and AGO. The AIF conditional macro pseudo-op aids in branching to the statement immediately following the label specified depending on the condition of the test performed. The AGO unconditional macro pseudo-op aids in branching to the statement immediately following the label specified within the macro definition body. The macro pseudo-ops AIF and AGO provide flexibility in generating different sequences of instructions from the macro definition body on different conditions. The first character of the label used in the macro pseudo-ops AIF and AGO is a period. Consider the following example which demonstrates the behavior of macro pseudo-ops AIF and AGO.

```
         MACRO                              MACRO CALL
         ADD &ARG1, &ARG2, &ARG3            ADD A, B, 2
         LDA &ARG1                          generates the code
         AIF (&ARG3 EQ 2) .LAB1             LDA A
         SUB &ARG2                          ADA B
         AGO   .LAB2                        STA A
.LAB1    NOP                                MACRO CALL
         ADA &ARG2                          ADD A, B, Ø
.LAB2    NOP                                generates the code
         STA &ARG1                          LDA A
         MEND                               SUB B
                                            STA A
```

1.5 MACRO CALLS WITHIN MACROS:

The macro body of a macro definition is a sequence of assembly language instructions with general operands. The conceptual consideration of this sequence of instructions as another assembly language program leads to consideration of a facility for calling another macro from this conceptual assembly language program which is actually a body of a macro definition. This facility is an extension of the very basic concept of macros, i.e., the abbreviation of a repeated sequence of instructions within macros. This facility is referred to as <u>macro calls within macros</u>. It can be noticed that the macro calls can occur only after the definition of the corresponding macro. This is a fundamental restriction in the macro processor implementation. This restriction applies to macro calls within macros. An example to demonstrate this facility is presented below.

```
  .
  .
  .
  .
  .
MACRO
ADD &ARG1
LDA &ARG1
ADA &ARG1
STA &ARG1
MEND
MACRO
ADDS &ARG1, &ARG2, &ARG3              LDA A
ADD  &ARG1                            ADA A
ADD  &ARG2                            STA A
ADD  &ARG3                            LDA B
MEND                                  ADA B
  .                                   STA B
  .                                   LDA C
  .                                   ADA C
  .                                   STA C
  .                                     .
ADDS  A, B, C,                          .
  .                                     .
  .                                     .
  .                                     .
```

In the above example it should be noted that expansion occurs level by level. A call to macro ADDS results in calls to macro ADD with different arguments.

1.6 MACRO DEFINITION WITHIN MACRO DEFINITION:

The argument for allowing macro calls within macro definition leads us to consider the possibility of allowing macro definitions within another macro definition. The usefulness of this facility is demonstrated if we allow the macro name itself to be a macro instruction argument. This flexibility enables us to define new macros with the help of a single macro instruction.

```
MACRO
TEST    &ARG
MACRO
&ARG    &ARG1, &ARG2
LDA     &ARG1
ADA     &ARG2
STA     &ARG1
MEND
MEND
```

```
TEST          TEST 1
TEST 1        A,B
generates the code
LDA A
ADA B
STA A
```

1.7 MACRO PROCESSOR FUNCTION:

A macro processor should have the capability of recognizing macro definitions and macro calls. It should also have the capability of recognizing conditional macro expansion pseudo-ops and macro definitions within macro definitions. In brief, the function of macro processor can be classified into two phases, the macro definition phase and the macro expansion phase. The third chapter deals with the actual implementation of the macro processor algorithm in PILOT (Purdue Instructional Language for writing operating systems and Translators) on the HP2100A, under the DOS-M operating system.

CHAPTER 2

PILOT - AS A SYSTEMS PROGRAMMING LANGUAGE

2.0 INTRODUCTION:

One of the basic problems faced by anyone who is developing system software is to choose a suitable language among the available higher-level languages (eg. PL/I, PASCAL, BLISS, etc.) and lower-level languages (Assembly Language, PL300). One can find in the literature arguments for and against these two fundamental approaches. It may also be observed that there are inadequacies in a language of either type [FRA 75]. A compromise can be achieved by using a language, which belongs to the class of what we call Intermediate-level Languages, for developing system software. The language PILOT is suggested as a possible candidate for developing system software. PILOT can be considered as an Intermediate-level Language as it has the capability to behave like both a higher-level language and lower-level language to a certain extent.

2.1 FACILITIES AND RESTRICTIONS OF PILOT LANGUAGE:

The compiler of language PILOT is of minimal size consisting of approximately two hundred and fifty source statements [HAL 74]. This size is minimal compared to some existing compilers of higher-level languages. The minimal size of compiler of PILOT does not imply that PILOT excludes some important features common to many of the existing higher-level languages. PILOT supports the concept of modularity. This is one of the important facilities which aids in introducing the concept of hierarchical structure in the software. The term 'concept' is used because of the fact that there is no universally acceptable measure to find out whether a piece of software is structured or how effective is the hierarchical modular structure of the software. The above two qualities of structuredness and hierarchical modular structure in software can be introduced by the implementor by judicious use of some of the available techniques eg. subroutines, block structure, control structure. The concept of modularity has been in extensive use particularly in the area of operating systems. This concept helps to fit one's problem better to a given environment. If the modules of a piece of software are independent, the facility of overlay techniques can be exploited to advantage in the case of minicomputers where resources such as main memory are limited. The availability of subroutines in PILOT is one of the points in favour of PILOT for developing system software. The compiler for PILOT is itself written in PILOT in a modular fashion. It is worth noting that this modular structure of PILOT allows one the flexibility of adding new features to the language. The language PILOT is portable as indicated by its implementation on various machines eg. IBM 1130, IBM 1620, HP2100A.

There are no keywords or reserved words in PILOT and there is no block structure as such, as part of the language. It is possible for a programmer to make a PILOT program resemble block structured code found in any PL/I or ALGOL program. For example consider the program given in fig. 2. The subroutine is divided into two basic parts. The first part is used to declare global and local variables made use in the subroutine. The second part contains the executable code. The difference between the ALGOL type block and the PILOT subroutine is that entry to the subroutine is made at an entry point having as a label the name of the subroutine (i.e., line 10 in fig. 2) rather than through the statement which identifies the beginning of the module (see fig. 2 line no. 1). There are no explicit data types in PILOT. The variables such as ARRAY A, LENGTH OF THE ARRAY, ARRAY INDEX, etc. are declared either as global or local variables used in the subroutine. This also demonstrates the flexible feature of allowing variable names of unlimited length. This assists the programmer in expressing the logical flow of the program and to a great extent helps in the documentation. This is amply demonstrated in the example of fig. 2. This facility may be cumbersome in developing large pieces of software for care has to be taken to avoid collision among variable names, as variable names are truncated at the sixth character. The facility of unlimited variable name length is not used to a great extent in implementing the macro processor this project, for the author wanted to experiment to see how structured programs help in communicating or understanding the logical flow in the program. The assignment statement in PILOT is simple and does not allow parenthetical grouping of expressions. The language PILOT supports two relational operators equality and less than. An interesting observation is made by the author regarding these relational operators later in the chapter. It can be noticed from fig. 2, that the operator (=) is used to assign initial values for some of the variables and in the executable code the same operator is used as a relational operator. The language does not support floating point arithmetic or facilities for bit manipulation. The former does not hinder the work of the systems programmer, whereas the later restriction is irksome. The only type of arithmetic supported by the language PILOT is integer arithmetic. This can be used to advantage to implement common functions which occur during the course of developing system software, packing and unpacking of characters in a word. The facility of integer division can be used to perform the above functions of packing and unpacking. This is an inefficient approach but this technique seems to be simpler and more straight forward than built-in shift operation facilities.

There is another point worth mentioning, the restrictions imposed by the language may look irksome to an application programmer, but for a systems programmer these restrictions give an opportunity to understand the implementation better by forcing him to think at every stage. Another important feature to be exploited in the language PILOT is the use of character variables arithmetically. This facility is not allowed even in higher level languages like PL/I [FRA 75]. This facility is made use for most commonly used operation of searching. It is convenient for the generation of hash codes from the entries to be searched. This is another important facility which supports the claim of PILOT as a systems program writing language. This facility can also be viewed as dealing with characters at the machine level with higher level instructions of PILOT.

The language PILOT allows one to go down one level to a lower level language. This capability is achieved using the facilities provided in the language PILOT known as crutch coding. The term crutch coding can be understood as the ability to mix the code of a lower level language e.g., assembly language instructions, with the code of higher level language. This facility is not unique to the PILOT language. But this facility can be used to achieve machine-dependent features, which cannot be achieved using higher-level language instructions, by mixing the assembly language instructions along with the higher-level language instructions. This facility substantiates the earlier claim that PILOT behaves like a lower-level language. This is another argument in favour of the language PILOT for writing system software. In addition to these facilities, PILOT has the facility for addressing all available core directly, that is, an instruction like K, results in a branch to the location K. There was no opportunity to make use of this facility in this project. The use of this facility can be seen in operating system routines. This is demonstrated in the operating system developed by Halstead [HAL 74], where this facility is made use of in the scheduler and I/O routines.

The restriction of declaring all the global and local variables used in the module is a good software engineering practice, as this aids in debugging any side effects either due to system malfunction or the language itself.

2.2 STRUCTURED PROGRAMMING IN PILOT:

The PILOT language itself does not include facilities for writing structured programs. This lack of structuredness can be seen in the language itself [HAL 74]. A new programming philosophy is advocated to implement the standard structured programming constructs [MCG 75]. We are of the opinion that it is possible to implement structured programming constructs in any higher level language by making use of the philosophy that unconditional jumps within the module should always be to the beginning of a loop or exit from the module, and conditional jumps should branch to the statements below the point from where one intends to jump. The implementer also should think that only two relational operators are provided in the language (=, <). This philosophy was followed in all the modules developed for this project. In one of the studies conducted by Neely [NEE 76], these concepts have been used to write structured programs in FORTRAN. The author does not explicitly specify the above mentioned philosophy. A close look at the examples discussed reveals this fact.

2.3 DISCUSSION OF PILOT AS SYSTEM PROGRAMMING LANGUAGE:

The argument in favour of higher level languages for developing system software is reliability. Reliable software is produced with the help of concepts of modularity and structured programming. Reliable software leads to reduced development and maintenance costs. The PILOT language supports the above mentioned concepts of modularity and structured programming. This shows PILOT can behave to a great extent as a higher level language. The lower level languages like Assembly Language give the implementor more control over the implementation-dependent functions, and produces more highly optimized object code than higher level languages. In the case of the PILOT language the implementor can have control over the machine-dependent functions using the facility of Crutch Coding. In this respect PILOT can be considered to behave like a lower level language. It should be pointed out that PILOT is not the only answer to a suitable language for writing system software. There are several languages like BLISS, PL360, etc. that have appeared in literature for possible candidature to become a system programming language.

In view of the facilities available in PILOT it can be considered as another possible candidate for the writing of system software.

```
       SUBROUTINE, FIND THE MAXIMUM VALUE IN THE ARRAY A
//     EXTERNALS
       ARRAY A,
       LENGTH OF THE ARRAY A,
       ;;
//     LOCALS
       *MAXIMUM VALUE,
       ARRAY INDEX,
       ZERO=0,
       ONE=01,
       ;;
//     PROGRAM
       FIND THE MAXIMUM VALUE IN THE ARRAY A: ?
       ARRAY A[ZERO]←MAXIMUM VALUE,
       ONE←ARRAY INDEX,
  CONTINUE SEARCH:
       ARRAY INDEX=LENGTH OF THE ARRAY A $ RETURN MAXIMUM VALUE. ;
       MAXIMUM VALUE<ARRAY A[ARRAY INDEX] $ INTERCHANGE. ;
       ARRAY INDEX+ONE←ARRAY INDEX,
       CONTINUE SEARCH.
  INTERCHANGE:
       ARRAY A[ARRAY INDEX]←MAXIMUM VALUE,
       ARRAY INDEX+ONE←ARRAY INDEX,
       CONTINUE SEARCH.
  RETURN MAXIMUM VALUE:
       ..
       ,
```

Fig. 2 Structure of a Module in PILOT language indicating certain features.

CHAPTER 3

MACRO PROCESSOR IMPLEMENTATION IN PILOT

3.0 INTRODUCTION:

The two main phases of the Macro Processor mentioned earlier in chapter 1 can be further classified as follows. During the Macro definition phase the Macro Processor should recognize the macro definitions with the help of MACRO and MEND pseudo-ops, and also macro definitions within macros. The Macro Processor should save the body of the macro definition with the pseudo-op MEND, as this information is used during the macro expansion phase. During the Macro expansion phase, the macro processor must recognize macro calls and replace them by the corresponding assembly language instructions after substituting the macro call arguments for the macro definition arguments.

The macro processor implemented in this project accepts source input written in HP2100A Assembly Language together with Macro definitions and Macro calls. The output generated by the Macro Processor is source input to the HP2100A DOS-M Assembler, i.e., the Macro Processor is functionally independent of the Assembler.

3.1 IMPLEMENTATION ASSUMPTIONS:

The major restriction imposed in this study is that macro call substitutes text, not values for macro definition arguments. It is also worth mentioning that the programming style is very much influenced by the programming language used to implement the software. This argument is amply demonstrated in this project. The limitations imposed by the language in some cases will be an advantage as indicated in the previous chapter. The provision of only two relational operators to a great extent aids in the development of structured programs, on the other hand the limitations of the data types available in the language and the lack of facilities for bit handling are great handicaps to the implementor of system software. As discussed later in the chapter, the character strings are stored in arrays with single character per word. String matching is done by comparing individual characters. This leads to inefficiency in both storage as well as execution time. In order to overcome this inefficiency to great extent, packing and unpacking of characters is achieved by the facility of Integer division.

In this project the concept of modularity has been exploited to the maximum extent. This is revealed by the sparseness of the module dependency matrix as shown in Table 2. It should also be mentioned that redundant initialization of flag values and Initial values of counters are made in the program as an aid in debugging any side effects. In short, every effort is made to achieve structuredness, modularity and flexibility within modules to accommodate future modifications, with the limited facilities provided in the language. It is not claimed here that all the available features of the language are exploited.

3.2 DATA BASES OF MACRO PROCESSOR:

Data bases used by the Macro Processor are:

1. Assembly Language source deck with Macro Definitions and Macro Calls.
2. Macro definition Table (AMARDT) for storing the body of the Macro definitions.
3. Macro Name table (AMACNT) for storing the Macro names.
4. Macro Definition Table Counter (MADTC) indicates the next available entry in Macro Definition Table.
5. Macro Name Table Counter (MANTC) indicates the next available entry in the Macro name table.
6. Macro definition table pointer (MDTP) is a pointer to point to the Macro definition Table for the Macro under expansion.
7. Array (S) is used to implement the stack using a Last in First out strategy.
8. Array BUFFER is used to save the Assembly Language input text. The length of the array is forty words, enough to hold one source input of 80 characters with two characters per word.
9. Array SOURCE is used to save the eighty character Assembly Language Statement for processing. The length of this array is eighty words with one character per word.
10. HASHTB an array of thirty words used to store pointer to the macro name table.
11. ALA - argument list array used to save the Macro definition arguments along with their relative position (Index) in the Assembly Language Statement.

12. RESULT - an array of length five words is used to save the extracted fields or substrings from eighty character Assembly Language Statement.
13. IHASHPTR - pointer to save the Hash code generated for the entry to be searched or an entry to be saved in the Macro name table.
14. IFGO - an array of length five words for saving the characters A, I, F, G, O, .
15. OPER - an array of length ten words used for saving the characters A, D, E, G, L, N, O, R, Q, T, .
16. CHAR - an array of length eight words used for saving the characters M, A, C, R, O, E, N, D, .
17. TEMP - a temporary array of length eighty characters to save the Assembly Language source with Macro call arguments substituted for Macro definition argument.
18. CHR - an array of length twenty six words used to save the twenty six special characters presented in the Appendix A.
19. I - is pointer indicates the position in Assembly Language source statements.

3.3 FORMAT OF DATA BASES:

The format of important data bases is presented. The macro name table (AMARNT) is an array of length two hundred words. The macro names associated with their pointers to Macro definition table are saved in the Macro name table. The pointer to the Macro definition table counter is Macro definition table counter (MANTC), which points to the next available location in the macro name table. Fig. 3 illustrates the above described scheme.

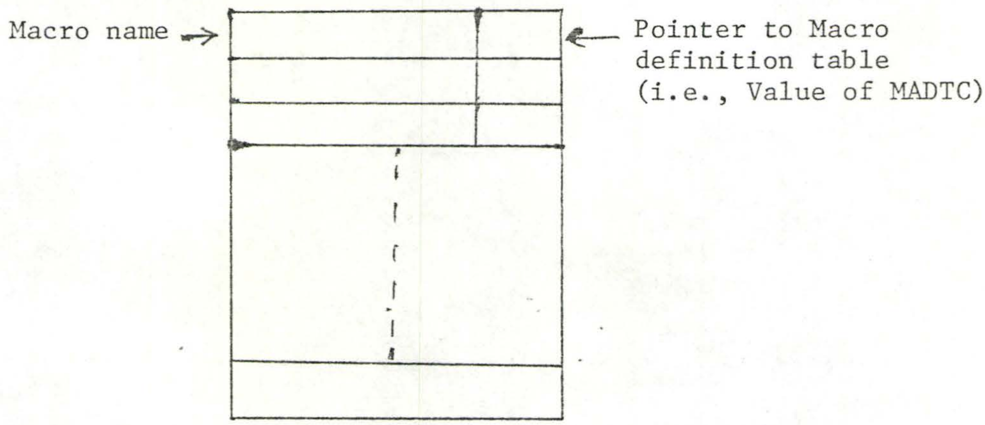


Fig. 3 Structure of Macro Name Table (AMACNT)

The Macro definition table (AMACDT) is an array of length two thousand words. The body of the Macro definition including the MEND pseudo-op, to indicate the physical end of the defined macro, is stored in packed format (i.e., Eighty character source input is packed into a buffer of forty words with two characters per word). Here all the blank characters are also stored. The main reason for this approach was less overhead. This problem can be overcome by minor modifications in the program. The modification is to scan the source input backwards up until a non-black character is encountered. Fig. 4 illustrates the structure of Macro definition table (AMACDT).

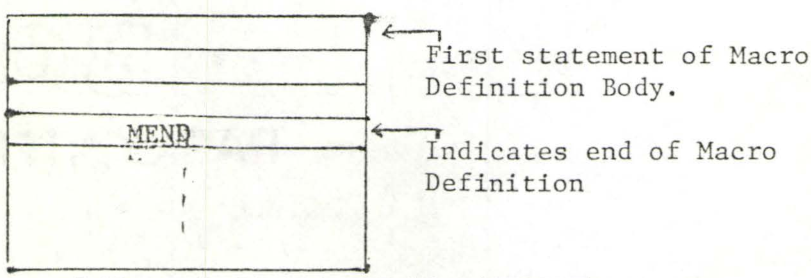


Fig. 4 Structure of Macro Definition Table (AMACDT)

The argument List array (ALA) is an array of eighty words. The macro definition arguments with their associated relative position are saved in the array ALA. This information is used to substitute the index notation for the macro definition arguments appearing in the body of the macro definition. The Structure of Argument List array is illustrated in Fig. 5.

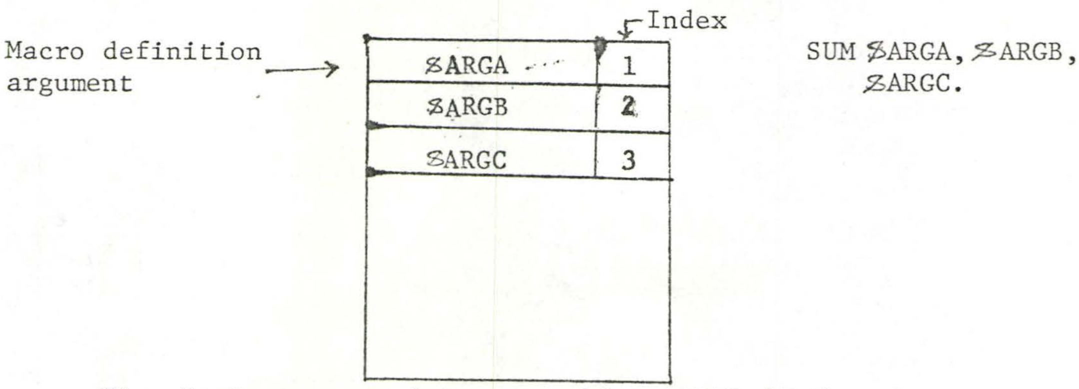
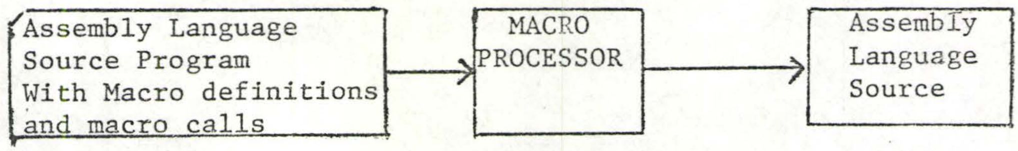


Fig. 5 Structure of Argument List ARRAY (ALA)

3.4 <u>IMPLEMENTATION OF THE ALGORITHM</u>:

The module PROCESSOR is the heart of the Macro Processor, which calls modules AINPUT, COMPARE, EXTRACT, SEARCH, HASH, PRARG, SINOT and STACK. The module PROCESSOR begins by initializing the global arrays and variables.

The outcome of the call to the module AINPUT is a source input either from the card reader, or from the macro definition table where the macro definition body is stored during macro expansion phase. The operation code field of this source input is extracted using the module EXTRACT. A search is made in the macro name table for a match with the operation-code field. A successful match indicates a



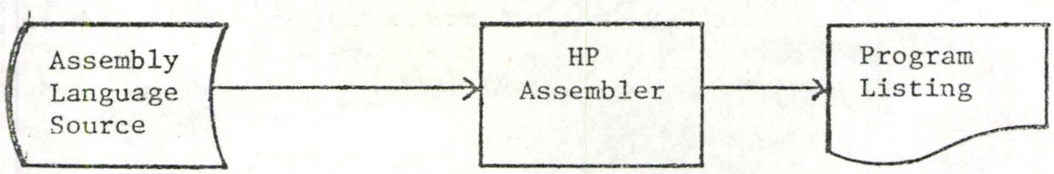


Fig. 6: MACRO PROCESSOR

macro call. A stack will be prepared for macro call arguments using the module STACK. Then the macro expansion phase begins. An unsuccessful match indicates either a macro definition phase or assembly Language source input. In order to determine whether it is the macro definition phase or it is an assembly language source input, the extracted operation-code field is further examined for a match with pseudo-operation code MACRO using module COMPARE. A successful comparison results in the reading of new source input and the extraction of the operation-code field of this new source input. The extracted operation code field which is stored in temporary array RESULT will be the Macro name. This macro name is stored in the Macro name table along with the pointer to the macro definition table, where the macro definition body will be stored. A hash code is generated by finding the sum of the ASCll codes of the first and last characters from the extracted name stored in the array RESULT, and then obtaining the remainder by dividing the sum by the length of the table to be searched. This method of generating a hash code is referred to as the division method [MAU 75]. In this project we have provided storage for handling thirty macros. In order to accommodate more macros, minor changes in the declarations have to be made. The remainder computed for the entry gives the home address in the hash table, where the pointer to the macro name table is stored. The source input is read and stored in the macro definition table until MEND pseudo-operation code is encountered. If there is a macro definition within a macro definition, a macro definition level counter is used to store all the macro definition body. This macro definition level counter is incremented by unity whenever a MACRO pseudo-operation code is encountered and is decremented by unity whenever a MEND pseudo-operation code is encountered. Unsuccessful comparison for the MACRO pseudo-operation code results in the output of these source lines, which will be either assembly language instructions of input text or the source output generated during macro expansion phase by substituting the macro definition arguments with the macro call arguments. This source output will be also an assembly language instruction. The extracted operation code is also compared for match with pseudo-operation code END. A successful match results in the completion of macro processing. The source output text produced by the Macro processor will be passed on to the Assembler for further processing. An unsuccessful match results in a return for further processing of source input text.

It should be pointed out here that source input of forty words stored in the array BUFFER is stored in its entirety in the Macro definition table, i.e., the source input in the array BUFFER does not contain useful information in all the forty words. One solution is to scan the BUFFER backwards until a non-blank character is encountered. Then only the information upto that point can be stored in the macro definition table. This results in savings in storage but an overhead results in scanning each source input to be stored in macro definition table.

The hash code for the macro name is generated in module HASH. The method used as indicated earlier is known as Division method. If the key for which the hash code to be generated is α and the hash table size is n, then home address of the key α in the hash table is given by

$$h(\alpha) = MOD(\alpha,n)$$

where $h(\alpha)$ is the home address of the key α. In this approach we have to face a problem known as collision, which is nothing but the generation of two keys having the same home address. One solution to this problem is to store this key in the next available position.

The module SEARCH is called by the main program PROCESSOR, and this module itself calls the module HASH. The hash code (IHASHPTR) is generated for the entry to be searched in the macro name table AMACNT, using module HASH. If the content of the location in the hash table i.e., HASHTB IHASHPTR is zero then the macro is not defined. The content of the Hash table gives the pointer to the macro name in the macro name table, and a comparison is made for the entry to be searched. This comparison is necessary for a collision might have occurred. A successful comparison results in return of appropriate flag value along with the corresponding pointer for the macro definition table.

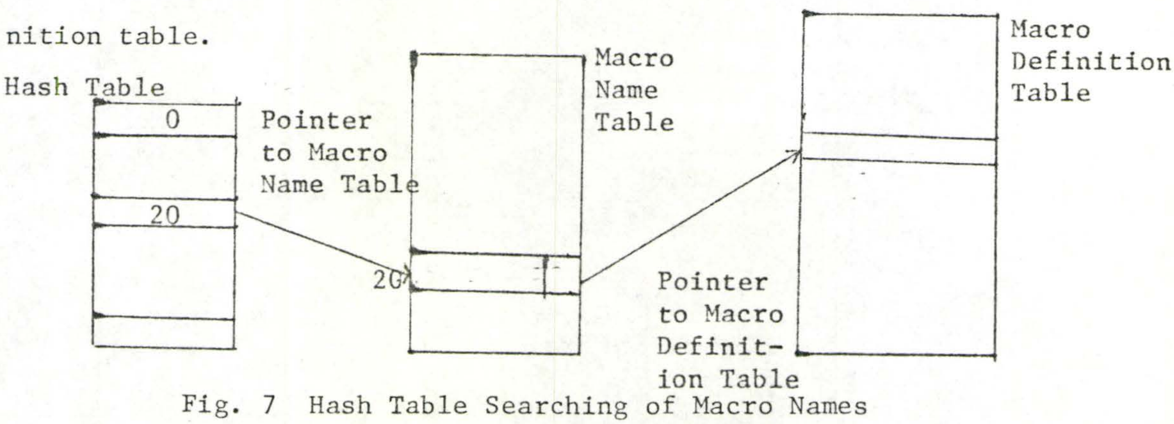


Fig. 7 Hash Table Searching of Macro Names

The module AINPUT is called by the module or main program PROCESSOR. This module calls the modules AGOS, AIFGO, AIFS, EXTRACT, PACK, SUBARG, and UNPAK. The function of this module can be grouped into two phases, Macro expansion phase and source input text phase. These two phases are decided from the value of the stack pointer, STKPT. The function of this module during source input text phase is to read the input text from the card reader into an array buffer of forty words with two characters per word. If the first character of the Input text is an asterisk, this will be a comment and Input text is written to the output file. A new source Input text is read and processing continues. In case of the macro expansion phase, the pointer to the macro definition table is retrieved from the Stack. The operation-code field is extracted from the source input stored in the macro definition table. A comparison is made for a match with the pseudo-operation code MEND. A successful match results in either termination of the macro expansion or expansion of the outer macro after popping back to the previous stack frame, depending upon whether there is a nested macro call or not. An unsuccessful comparison for a match with pseudo-operation code MEND results in substitution of actual arguments i.e., macro call arguments for the macro definition arguments. Further processing of pseudo-operation codes AIF, AGO is performed.

The module UNPAK is called by the modules PROCESSOR, AINPUT. This module itself does not call any other modules. The function of this module is to unpack the contents of the array BUFFER of length forty words with two ASCll characters per word, and to store one ASCll character per word in the array SOURCE of eighty words. In the PILOT language there is no bit handling facility. This forces one to make use of the integer division facility to perform the above function. The word size of the machine under consideration (HP2100A) is 16 bits or two bytes. In order to shift eight bits or one byte the contents of each word is divided by 2^8. Refer to the routine UNPAK for the actual code. The advantages of storing characters in an Unpacked format is that of ease in processing. The module PACK is called by the modules PROCESSOR, AINPUT and ERROR. This module does not call any other modules. The function of this module is exactly opposite to that of module UNPAK. The eighty characters stored in the array SOURCE with one ASCll character per word are packed into the array BUFFER of forty words with two ASCll character per word. Refer to the routine PACK for the actual code. The advantage of storing characters in Packed format is that of storage conservation.

The module EXTRACT is called by most of the other modules except PACK, UNPACK, IFTEST and this module itself calls the module SPCHR. The function of this module is to extract a substring from the string of eighty characters stored in the array SOURCE. The extracted substring is stored in the array RESULT with one character per word. The pointer I indicates the position in the eighty character string stored in the array SOURCE for extracting the substring. A substring is returned as soon as a delimeter such as a blank character, a special character, etc. is encountered.

The module SPCHR is called by the module EXTRACT and this module does not call any other module. The twenty six special characters (see appendix A) are stored in the array CHR. The module using a linear search of the array CHR returns a corresponding flag value depending upon success or failure of the search for the special character. The Linear Search technique is used as other well known techniques like Binary Searching, Hash Table Methods of searching are inefficient for a table with less than thirty entries [MAU 75] and [DON 72].

The module PRARG (Prepare argument list Array) is called by the main program PROCESSOR and this module itself calls the module EXTRACT. The function of this module is to prepare a macro definition argument list array with corresponding relative position of these arguments in the macro definition. The module extracts each macro definition arguments and stores in the array ALA (argument list array) along with index for its relative position in the macro definition.

The module SINOT (Substitute Index Notation) is called by the main program PROCESSOR, and this module itself calls the module EXTRACT. The function of this module is to substitute the index of the macro definition argument stored in the array ALA in the macro definition body. The module extracts the macro definition argument from the macro definition body and substitutes the index of the macro definition argument. The index is obtained by searching for the macro definition argument in the array ALA.

The module SUBARG (Substitute Arguments) is called by the module AINPUT and this module itself calls module EXTRACT. The function of this module is to substitute the macro call arguments stored in the Stack S for the macro definition arguments in source input obtained from the macro body. The source input from the macro definition table is scanned for the macro definition arguments. The formal parameter or macro call argument stored in the stack is obtained using a pointer, which is computed using the stack pointer and the relative position of the macro definition argument in the macro definition.

The module COMPARE does not call any other module. The function of this module is to recognize the pseudo-operation code MACRO, MEND, END, and return the appropriate flag values. The character string MACROEND is stored in an array CHAR. The entry to be compared in array RESULT is input to this module. A comparison is made with the characters stored in the array CHAR, and an appropriate flag value i.e., either 0 or 1, is returned depending upon the success or failure of the comparison.

The module AIFGO is called by the module AINPUT and this module itself calls the module EXTRACT. The function of this module is to recognize calls to pseudo-operations AIF and AGO. The operation code field of the source input String is extracted and is compared for match with characters stored in the array IFGO. An appropriate flag value is returned depending upon the success or failure of the match.

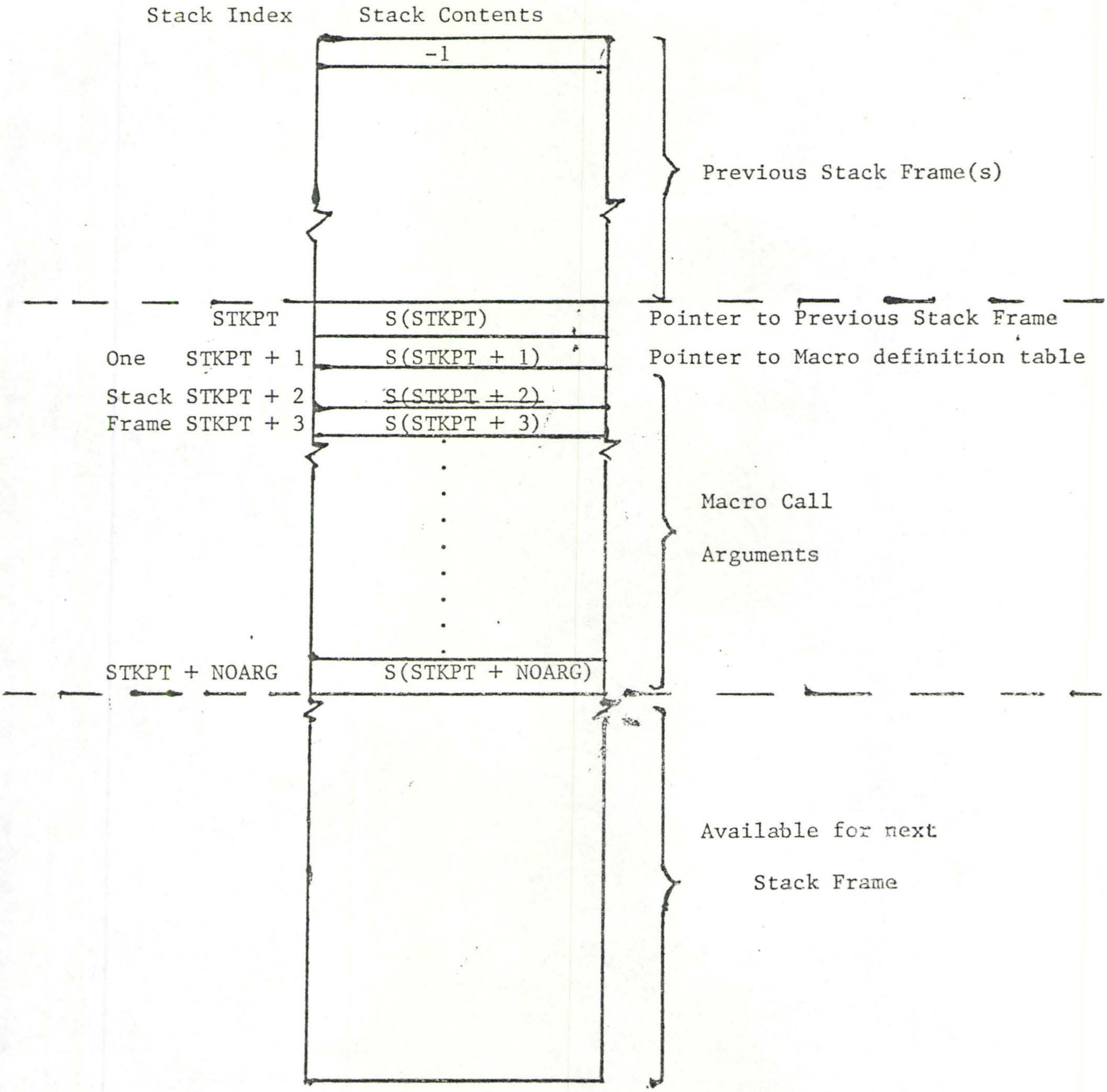
The module AGOS calls the module EXTRACT, and this module is called by the modules AINPUT and AIFS. The functions of this module are to extract the label associated with the pseudo-op AGO, and to update the macro definition table pointer (MDTP). The macro definition body of the macro under expansion is scanned for the label associated with the pseudo-op AGO. A successful match with the label results in updating the pointer to the macro definition body. An unsuccessful match by way of encountering MEND pseudo-op results in writing of an error message to the effect that the label specified with AGO, pseudo-op is undefined. The AGO pseudo-op provides flexibility of conditional expansion of macros.

The module RELATIONAL does not call any other module and is called by the module AIFS. The function of this module is to return an appropriate flag value for the various relational and logical operators. The relational and logical operators provided in this project are greater (GT), greater than or equal (GE), less than (LT), less than or equal (LE), Equality (EQ), Not equal (NE) and Logical AND, OR. The characters stored in the array RESULT are compared with the characters stored in the local array OPER and appropriate flag value is returned through the variable RFLAG.

The module IFTEST does not call any other module and this module is called by AIFS. The function of this module is to return an appropriate flag value through the global variable AIFFLAG depending on whether the operands of the relational test stored in the array OPERAND, satisfy the relation specified by the variable RFLAG.

The module AIFS calls the modules AGOS, IFTEST, EXTRACT and RELATIONAL. This module is called by AINPUT. The function of this module is to extract the operands of the Relational and Logical expressions, and the relational or logical operators. If the Relational and/or logical expression evaluates to true then the module AGOS is called i.e., a conditional jump is made to the label specified, otherwise the processing will continue.

The module STACK is called by the module PROCESSOR during macro expansion phase. This module calls the module EXTRACT. The function of this module is to set up a stack for the macro call arguments. The stack S is implemented using an array. This stack is used by the module SUBARG for substituting macro call arguments for macro definition arguments or dummy arguments of the macro body. The stack can be considered as an array of pointers and character strings. A stack pointer STKPT, indicates the beginning of the current stack frame. A stack pointer value of -1 indicates that the macro processor is not in macro expansion phase. The location S(STKPT) will provide the previous value of the Stack Pointer. The value of the stack pointer for the first frame, which is the top of the stack is zero i.e., STKPT = 0 and the contents of location S(STKPT) is -1. The contents of location S(STKPT+1) give the pointer to the macro definition table for the macro under expansion. The locations S(STKPT+2), S(STKPT+3), ... S(STKPT+NOARG), contain the character strings of the macro call arguments of the macro currently under expansion. A general outline of the stack is presented in fig. 8. The approach of using a stack to save macro call arguments is taken in order to facilitate the handling of macro calls within macros. The inner most macro will be under expansion, when pseudo-op MEND is encountered the expansion of outer Macro will continue. This is achieved by updating the stack pointer i.e., STKPT = S(STKPT) (as indicated earlier S(STKPT) points to the top of the previous stack frame).



S(i): Contents of i^{th} position on the Stack

STKPT: Stack Pointer

NOARG: Number of Arguments

Fig. 8: STACK ORGANIZATION

CHAPTER 4

STRUCTURED PROGRAMMING IN ASSEMBLY LANGUAGE

4.0 INTRODUCTION

The concept of structured programming is in a state of confusion and controversy. In computer science literature one can find a number of definitions for structured programming. The interested reader can refer to the articles in the Infotech state of the art report on structured programming [INF 76] structured programming can be defined as a way of organizing one's thoughts in a way that leads in a reasonable time, to an understandable and correct expression of a computing task. The aim of structured programming is the production of reliable software, which in turn leads to lower cost in overall development and maintenance of a piece of software. The principle of structured programming postulates that a critical factor in producing software, which is understandable, reliable and modifiable is the presence of some quality of structure such as:

(i) Developing software with the use of certain control and data structures.

(ii) Developing software with interconnections of modular units. The number of interconnections depends on the implementor and to a great extent on the problem.

(iii) Developing software in terms of hierarchial levels of abstraction.

With the above aims and principles of structured programming in mind, structured programming can be defined as the practice of writing programs that are well structured according to one or more of the above mentioned qualities.

It is worth presenting a definition of structured programming by Wirth [WIR 74] ,

> Structured programming is the expressing of a conviction that the programmer's knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically, and that a particular technique should be replaced (or augmented) by a method. At its heart lies an attitude rather than a recipe: the admission of the limitations of our minds.

4.1 NEED FOR STRUCTURED PROGRAMMING

As discussed in Chapter 2 of this report, there have been a lot of arguments in favour and against developing software in higher level languages or lower level language (Assembly Language). The one strong argument in favour of higher level language for developing system software is the presence of facilities for developing structured programs with the help of control structures and data structures provided in the language.

The reliability, readability and ease in debugging any piece of software implemented in Assembly Language is enhanced by grouping chunks of code into segments. These segments form a module which, in turn forms a program. In the case of the debugging of Assembly Language programs, there are three considerations: The ability to read and understand the intended function of the code, to follow the flow of control for designated test cases and to ensure data item integrity. The concept of structured programming itself does not help the problem of data integrity. Data integrity means that a portion of code in one segment does not inadvertently modify the contents or logic of other portions of code. With the low level nature of Assembly Language program - i.e., one statement corresponds in general to one machine instruction - a simple function in design may be a few or many instructions. This fluctuation in lines of code has a detrimental effect on readability regardless of organization. However, in adopting structured programming concepts, developing meaningful control function macros, the comprehensibility of structured Assembly Language programs is greatly increased over non-structured Assembly Language programs.

4.2 REVIEW OF WORK ON STRUCTURED PROGRAMMING IN ASSEMBLY LANGUAGE

A practical, productive approach to facilitate structured programming in Assembly Language is to develop macro definitions for each segment of code. A set of structured programming macros were developed by Kessler [KES 72]. These macros have very flexible predicate formats, but their format is rather awkward and does not enhance readability. The advantages of Kessler's predicate formats are that all possible Assembly Language tests and comparisons can be included within the predicates and the length of the predicate is only limited by assembler resources. These macros were developed by Kessler with the following macro and assembly time facilities: Integer arithmetic, character string variables, the arithmetic operations of addition, subtraction, string operation of concatenations, substring extractions, and length determination, and finally conditional expansion pseudo-ops. The macro processor is a part of the Assembler, i.e., macro Assembler. The macro processor developed in this project is functionally independent of the Assembler. The approach taken in this project to provide macros for writing structured programs is a very new development. This approach is aimed at achieving simplicity and readability. The macros developed in this project make use of the facilities of conditional macro expansion psuedo-ops only.

4.3 IMPLEMENTATION OF STRUCTURED PROGRAMMING CONSTRUCTS

The structured programming control structures considered in this study are the fundamental control structures advocated in the literature [MCG 75]. These are IF-THEN-ELSE and DO-WHILE control structures. The approach taken in this study is to develop three Macros in case of IF-THEN-ELSE control structure, a macro to handle the IF-THEN part, a macro to handle the ELSE part and a macro to handle the range of the IF-THEN-ELSE control structure. The macro for the IF-THEN part accepts two operands, a predicate and a label. When the predicate evaluates to a false condition a branch occurs to the specified label, otherwise the processing continues without any branching. The ELSE macro has two arguments. The first argument specifies the range of the IF-THEN-ELSE control structure. The second argument specifies the beginning of the ELSE part of the IF-THEN-ELSE control structure.

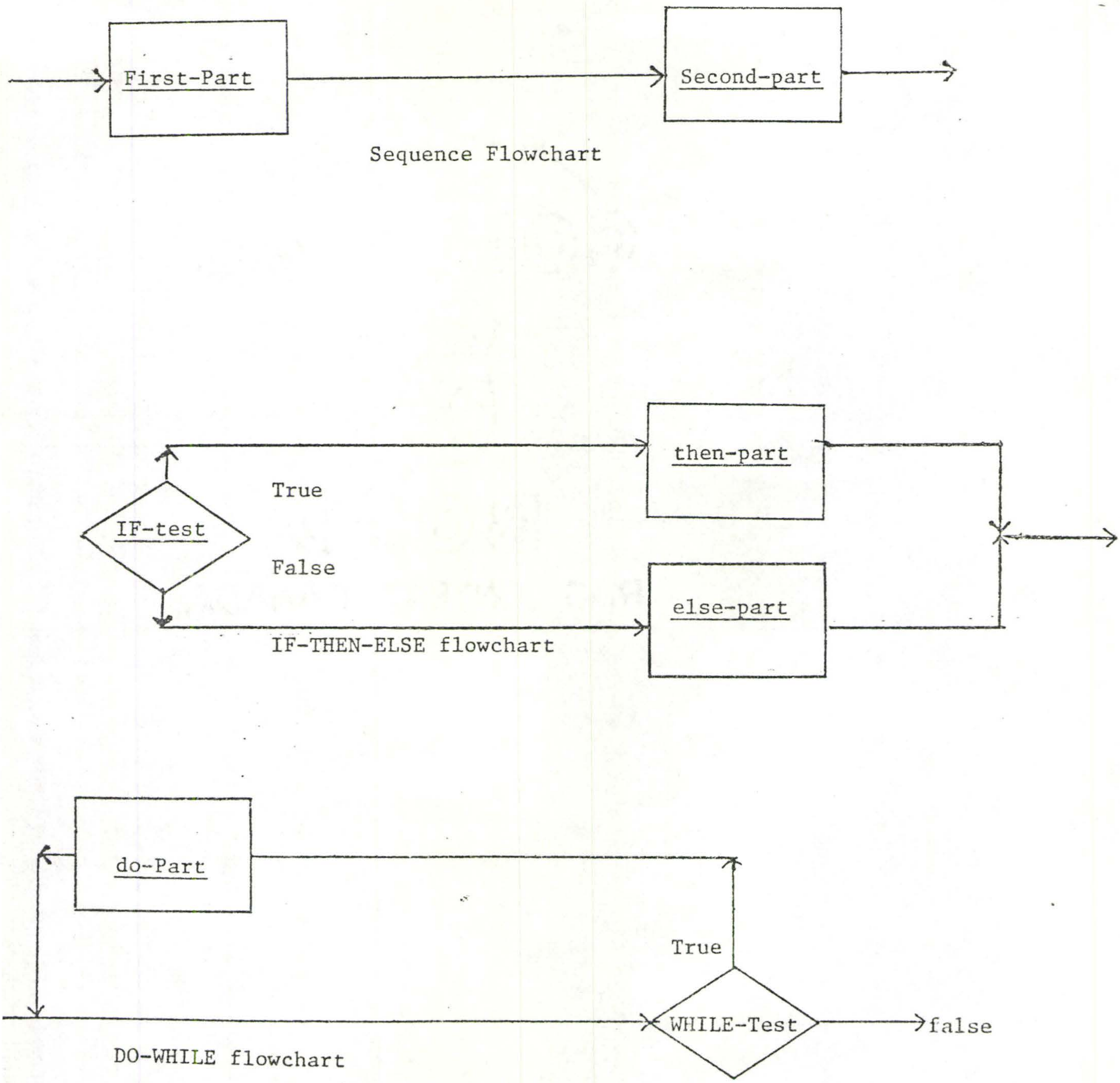


Fig. 9: FUNDAMENTAL STRUCTURED PROGRAMMING CONSTRUCTS

```
        IF Predicate Labell                    IF-THEN

           JMP  Label2
Labell     NOP                                 ELSE


Label2     NOP                                 ENDIF
```

The DO-WHILE control structure is implemented using two macros, one macro to handle the DO-WHILE part and another macro to handle the range of the DO-WHILE control structure.

```
Labell     NOP
           WHILE (Predicate)      Label2       WHILE

           JMP  Labell
Label2     NOP                                 ENDWHILE
```

In implementing the above macros only the psuedo-op's AIF and AGO are used. The above are simple to use and understand. With the help of the above macros, it should be possible to write structured programs in assembly language. This area has potential for further research as there are only three research publications in the literature [RIE 76], [KES 70]. The approach taken in this study in implementing the above macros is unique in the sense that only two psuedo-ops are used and this facility is provided for a macro processor which is functionally independent of the Assembler.

CHAPTER 5

CONCLUSIONS

A Macro Processor with the capability of handling Macro calls within Macros, Macro definitions within a Macro definition, the String operation of concatenation and an ability to branch conditionally and unconditionally within a Macro has been successfully implemented. The PILOT language with its limited facilities has been successfully used to implement the Macro Processor Software. A new programming philosophy of Unconditional branch to beginning of a loop and a conditional branch to a point below that from which the branch is taking place is experimented with to achieve structuredness in the software. The concept of modularity has also been used to the greatest extent possible. The software has sufficient flexibility for modification and extension in order to allow the incorporation of new features. A limited set of Macros for fundamental structured programming constructs is implemented within the frame work of available features of the Macro Processor implemented. Research work in this area has not progressed because of the development of higher level languages for developing system software. The applicability of Macro Processors to aid portability of Assembly language programs is still to be explored.

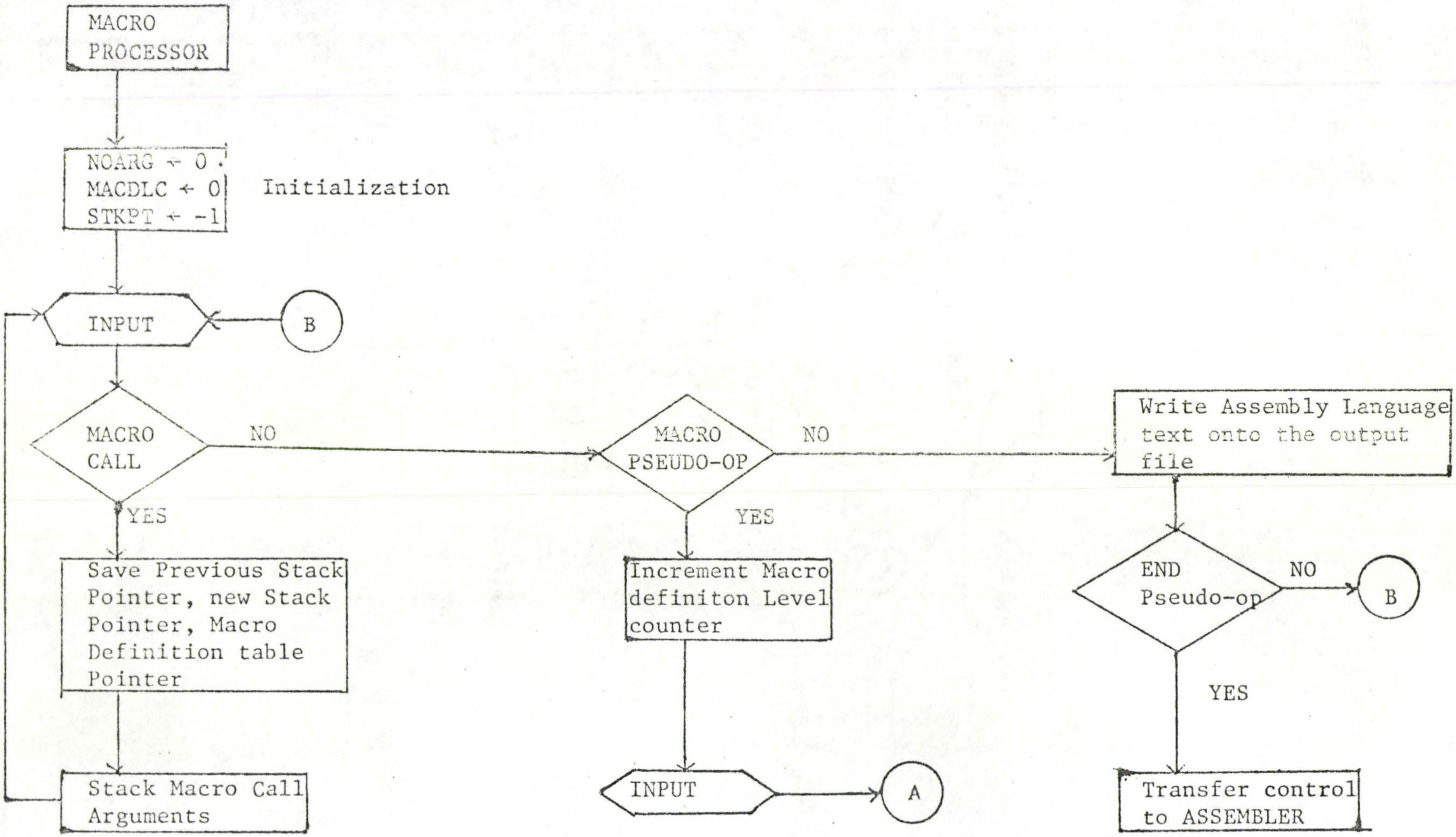
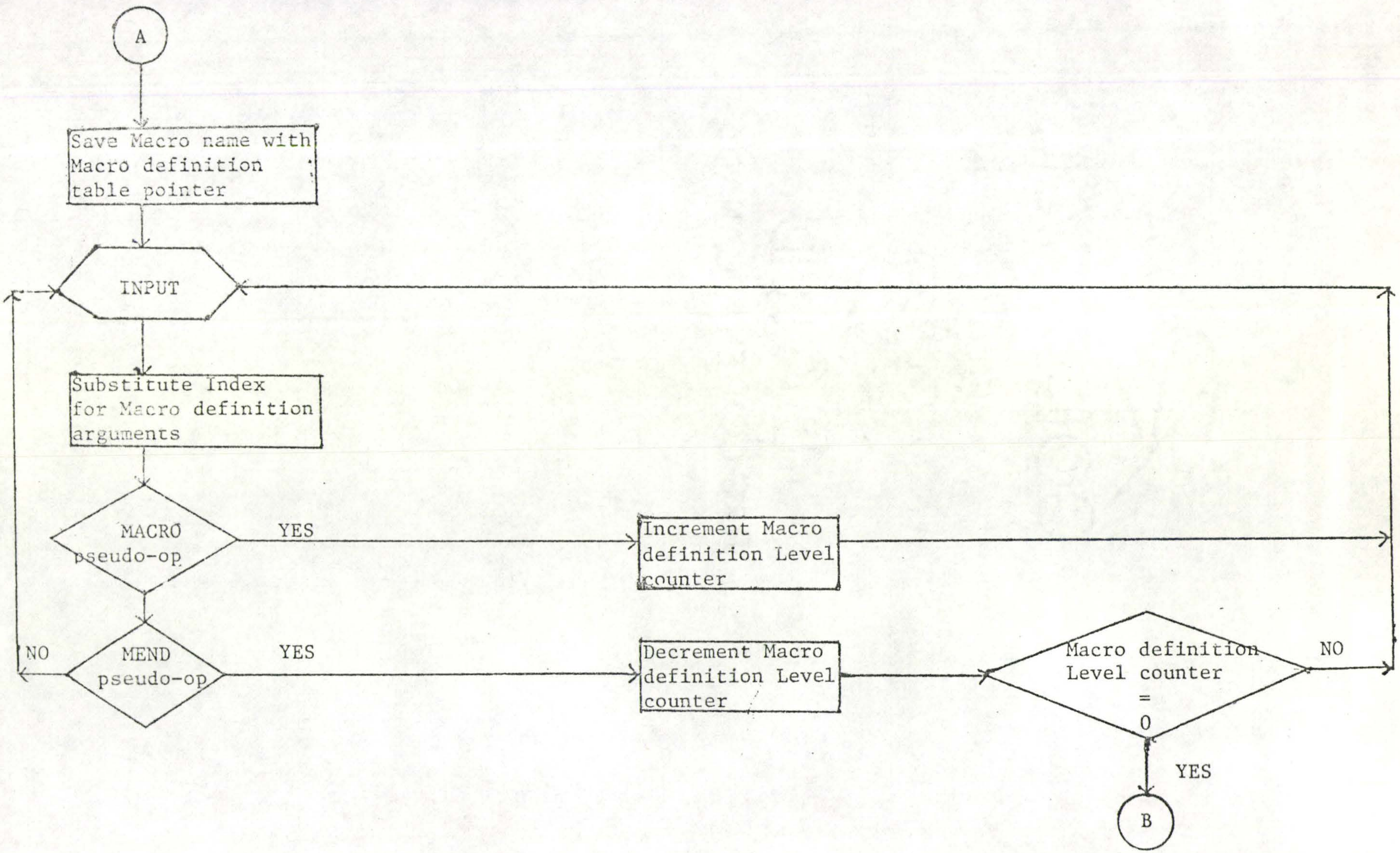
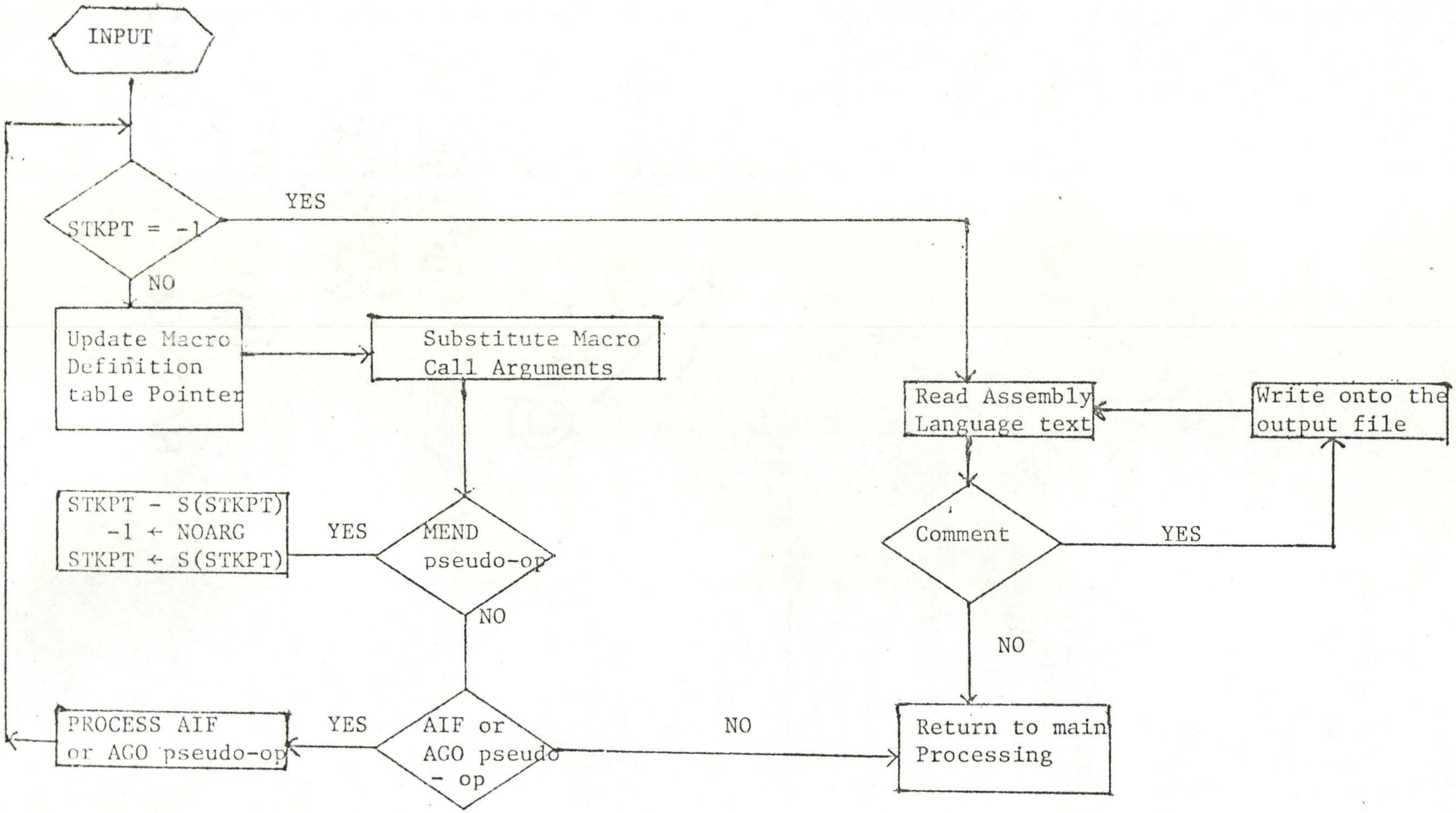


Fig. 10: MACRO PROCESSOR FLOW CHART

Cont.


A
Save Macro name with Macro definition table pointer
INPUT
Substitute Index for Macro definition arguments
MACRO pseudo-op
YES
Increment Macro definition Level counter
NO
MEND pseudo-op
YES
Decrement Macro definition Level counter
Macro definition Level counter = 0
NO
YES
B

Cont.


INPUT
STKPT = -1
YES
NO
Update Macro Definition table Pointer
Substitute Macro Call Arguments
Read Assembly Language text
Write onto the output file
STKPT - S(STKPT)
-1 ← NOARG
STKPT ← S(STKPT)
YES
MEND pseudo-op
NO
Comment
YES
NO
PROCESS AIF or AGO pseudo-op
YES
AIF or AGO pseudo - op
NO
Return to main Processing

	MACRO-PROCESSOR	AINPUT	UNPAK	PACK	EXTRACT	COMPARE	SEARCH	STACK	PRARG	SINOT	SUBARG	AIEGO	AGOS	AIFS	IFTEST	RELATIONAL	SPCHR	HASH	ERROR
ALA									+	+									
AMACDT	+	+											+						
AMACNT	+						+												
BUFFER	+		+	+															
HASHTB	+						+												
OPERAND														+	+				
RESULT		+			+	+	+	+	+	+	+	+	+	+		+		+	
S		+						+			+		+						+
SOURCE	+	+	+	+	+	+		+	+	+	+	+	+	+					
MESSAGE	+																		+

Table 1: Global Arrays Utilization Matrix

	MACRO-PROCESSOR	AINPUT	SUBARG	AIFGO	AGOS	AIFS	IFTEST	RELATIONAL	EXTRACT	SPCHR	UNPAK	PACK	SEARCH	HASH	STACK	COMPARE	PRARG	SINOT	ERROR
MACRO PROCESSOR		+							+			+	+		+	+	+	+	+
AINPUT			+	+	+	+			+		+	+				+			
SUBARG									+										
AIFGO																			
AGOS									+		+					+			+
AIFS					+		+	+	+										
IFTEST																			
RELATIONAL																			
EXTRACT										+									
SPCHR																			
UNPAK																			
PACK																			
SEARCH														+					
HASH																			
STACK									+										
COMPARE																			
PRARG									+										
SINOT									+										
ERROR												+							

Table 2: Module Dependency Matrix

APPENDIX - A

HP CHARACTER SET

Symbol	ASCII (Octal code)
(Space)	4Ø
!	41
"	42
#	43
$	44
%	45
&	46
'	47
(	5Ø
)	51
*	52
+	53
,	54
-	55
.	56
/	57
Ø	6Ø
1	61
2	62
3	63
4	64
5	65
6	66
7	67
8	7Ø
9	71
:	72
;	73
<	74
=	75
>	76
?	77
@	1ØØ

Symbol	ASCII (Octal code)
A	1Ø1
B	1Ø2
C	1Ø3
D	1Ø4
E	1Ø5
F	1Ø6
G	1Ø7
H	11Ø
I	111
J	112
K	113
L	114
M	115
N	116
O	117
P	12Ø
Q	121
R	122
S	123
T	124
U	125
V	126
W	127
X	13Ø
Y	131
Z	132
[	133
]	135
↑	136
←	137

APPENDIX - B

<u>PILOT Language Syntax</u>

```
ROUTINE         RT::=NL;;LA VL..
NOUN LIST       NL::=DC/NL,DC
DECLARATION     DC::=NA/NA=NR/FA/NA/[NR] /NA:NR
FILLED ARRAY    FA::=NA [NR] =NR/FA,NR
VERB LIST       VL::=ST/VL ST
STATEMENT       ST::=AS/CO/SR/JU/CA/RD/WR/CR/LA ST
ASSINGMENT      AS::=OP←OP,/ OP AR OP←OP,
COMPARISON      CO::=OP RE OP : JU ;
SUBROUTINE      SR::=LA: ? VL
JUMP            JU::=OP.
CALL            CA::=OP,
READ            RD::=>NA<
WRITE           WR::=<NA>
CRUTCH          CR::=$NR,NR;/$NR,NA;/$NA,NR;/$NA,NA;
ARITHMETIC      AR::=+/-/*//
RELATIONAL      RE::==/<
OPERAND         OP::=NA/NA SS/SS
SUBSCRIPT       SS::=[IN]
LABEL           LA::=NA
NAME            NA::=LE/NA LE/NA NR
NUMBER          NR::=O O HN/ O ON/DN/O
LETTER          LE::=A/B/C/D/E/F/G/H/IN/O/P/Q/R/S/T/U/V/W/X/Y/Z
INDEX           IN::=I/J/K/L/M/N
HEXANUMBER      HN::=HD/HN HD/HN O
DECIMALNR       DN::=DD/DN DD/DN O
OCTALNR         ON::=OD/ON OD/ON O
OCTALDIG        OD::=1/2/3/4/5/6/7
DECIMALDIG      DD::=8/9/OD
HEXDIG          HD::=A/B/C/D/E/F/DD
```

APPENDIX - C
PROGRAM LISTING AND EXAMPLES

```
//**********************************************************
//*                                                        *
//*   PURPOSE OF THE MODULE                                *
//*   TO PROCESS MACRO CALLS AND MACRO DEFINITIONS IN HP2100 *
//*   ASSEMBLY LANGUAGE SOURCE INPUT TEXT.                 *
//*                                                        *
//*   USAGE                                                *
//*   MODULES REQUIRED                                     *
//*   AINPUT                                               *
//*   EXTRACT                                              *
//*   ERROR                                                 *
//*   HASH                                                 *
//*   PACK                                                 *
//*   PRARG                                                *
//*   SEARCH                                               *
//*   SINOT                                                *
//*   UNPAK                                                *
//*   SYSTEM MODULES REQUIRED                              *
//*   CLSEFILE                                             *
//*   EXEC                                                 *
//*   GET ARGUMENT                                         *
//*   OPENFILE                                             *
//*   JOBFILE                                              *
//*   PUTCODE                                              *
//*                                                        *
//*   INPUT TO THE MODULE.                                 *
//*   ASSEMBLY LANGUAGE INPUT TEXT WITH MACRO DEFINITIONS  *
//*   AND MACRO CALLS.                                     *
//*                                                        *
//*   OUTPUT FROM THE MODULE.                              *
//*   ASSEMBLY LANGUAGE TEXT.                              *
//**********************************************************
   PROGRAM, MACRO PROCESSOR
     AINPUT,
//    BUFFER- AN ARRAY OF FORTY WORDS FOR SAVING ASSEMBLY
//    LANGUAGE TEXT.
     BUFFER,
     COMPARE,

     EXTRACT,
      EFLAG,
      ERROR,
     HASH,
     I,
//    IHASHPTR- SAVES HASH CODE GENERATED FOR ENTRY TO BE
//    SEARCHED AND FOR MACRO NAME TO BE STORED IN MACRO
//    NAME TABLE.
     IHASHPTR,
      MESSAGE,
     PACK,
     PRARG,
     "P.IN",
     "P.OUT",
     "P.HLT",
//    RESULT- AN ARRAY TO SAVE THE EXTRACTED SUB-STRING
//    FROM ASSEMLY LANGUAGE TEXT.
     RESULT,
//    S- AN ARRAY USED FOR IMPLEMENTING THE STACK TO
//    HANDLE MACRO CALLS WITHIN MACROS.
     S,
     SEARCH,
```

```
SINU'
STACK,

TAG,
CLSEFILE,
EXEC,
GET ARGUMENT,
JOBFILE,
OPENFILE,
PUTCODE,
;;

//    AFLAG- FLAG TO INDICATE A MACRO CALL.
     *AFLAG,
//    AMACDT- MACRO DEFINITION TABLE TO STORE THE BODY OF THE
//            MACRO DEFINITION.
     *AMACDT[2000],
//    AMACNT- MACRO NAME TABLE TO STORE MACRO NAME WITH
//            ASSOCIATED MACRO DEFINITION TABLE POINTER.
     *AMACNT[200],
//    ANEG1- REPRESENTS -1.
     *ANEG1=0177777,
//    BLANKBLANK- BLANK CHARACTER WITH ASCII CODE 040
     *BLANKBLANK=040,
//    BUFF LIM- LENGTH OF THE ARRAY 'BUFFER'.
     *BUFF LIM=050,
     *FLAG,
//    HASHTB- HASH TABLE WHICH SAVES POINTER TO MACRO NAME
//            TABLE.
     *HASHTB[30],
//    TEMPORARY VARIABLES.
     ITEMP,
     JTEMP,
     KOUNT,
//   DEFAULT LOGICAL UNIT NUMBERS
     *LUINPUT=5
     *LUOUTPUT=6,
     *LUOBJECT=2,
     *LUPUNCH =0,
     *LPPAGE  =56,
     *LGO     =0,
//   "ASMB  "
     ASMB[3]  =040523,046502,020040,
//    MACDLC- MACRO DEFINITION LEVEL COUNTER WHICH IS USED AS
//            A SWITCH TO CONTROL MACRO CALLS WITHIN MACRO
//            DEFINITION AND MACRO DEFINITION WITHIN MACRO
//            DEFINITIONS.
     *MACDLC,

//    MADTC- POINTER TO MACRO DEFINITION TABLE TO STORE MACRO
//           DEFINITION BODY.
     *MADTC,

//    MANTC- POINTER TO MACRO NAME TABLE TO STORE MACRO NAME.
     *MANTC,

//    MDTP- POINTER TO MACRO NAME IN MACRO NAME TABLE.
     *MDTP,

//    NOARG- NUMBER OF ARGUMENTS IN THE MACRO CALL.
     *NOARG,

//    SAND- SPECIAL CHARACTER '&' TO INDICATE MACRO DEFINITION
//          ARGUMENT.
```

```
      *SAND=046,
 //     STKPT- STACK POINTER.
      *STKPT,

      *0=0,
      *01=01,
      *02=02,
      *03=03,
      *04=04,
      *05=05,
      *06=06,
      *010=010,
      *012=012,
      *036=036,
      *050=050,
      *0110=0110,
      *0144=0144,
      *0200=0310,
      *02000=03720,
      ;;

      MACRO PROCESSOR: ?

 //     GET ARGUMENTS
        EXEC,
$       ,DEF,NEXT,,
$       ,DEF,07,,
   NEXT:
        GET ARGUMENT,
        OPENFILE,
 //   INITIALIZE ERROR FLAG
      0←EFLAG,
//    INITIALIZE TEMPORARY AND LOOP CONTROL VARIABLES.

      0←MDTP,

      0←NOARG,

      0←MACDLC,

      0←FLAG,

      0←KOUNT,

      0←MADTC,

      01←MANTC,
 //     STACK POINTER 'STKPT' POINTS TO THE TOP OF THE STACK.
      ANEG1←STKPT,

      0←ITEMP,

      0←JTEMP,

 //     ZERO FILL HASH TABLE 'HASHTB'.
   MLOOP:
      KOUNT=036 $ NLOOP. ;
      0←HASHTB[KOUNT],
      KOUNT+01←KOUNT,
      MLOOP.
   NLOOP:
      0←KOUNT,
 //     ZERO FILL MACRO DEFINITION TABLE 'AMACDT', MACRO NAME
```

```
//     TABLE 'AMACNT' AND ARRAY 'S'.
  CLOOP:

     KOUNT=0144 $ START. ;

     O←AMACNT[KOUNT],

      O←S[KOUNT],
  ELOOP:

     ITEMP=012 $ DLOOP. ;
     O←AMACDT[JTEMP],

     JTEMP+01←JTEMP,

     ITEMP+01←ITEMP,

     ELOOP.

  DLOOP:

     O←ITEMP,

     KOUNT+01←KOUNT,

     CLOOP.

  START:

//     EXIT IF ERROR FLAG IS SET
     EFLAG=01 $ CALL ASMB. ;
     O←TAG,
     O←I,
//     READ ASSEMBLY LANGUAGE INPUT TEXT.
     O←KOUNT,

     AINPUT,

//     EXTRACT THE OP-CODE FIELD OF THE INPUT TEXT.
     EXTRACT,

//   SEARCH MACRO NAME TABLE FOR MATCH WITH NAME IN OP-CODE FIELD

//     SEARCH MODULE RETURNS FLAG VALUE, AFLAG=01 IF SEARCH
//     IS UNSUCCESSFUL,ELSE AFLAG=0.
     01←AFLAG,
     SEARCH,

     AFLAG=01 $ AOTMACRO. ;
//     * MACRO EXPANSION PHASE *
//     SAVE MACRO CALL ARGUMENTS ON STACK
     STACK,

     START.

  AOTMACRO:

//     CHECK FOR MACRO DEFINITION PHASE.
//   MACRO PSUEDO-OP

     02←FLAG,

     COMPARE,
```

```
    FLAG=02 $ CONT. ;

//  MACRO PSUEDO-OP NO

//   OUTPUT THE ASSEMBLY LANGUAGE TEXT AFTER PACKING INTO
//   THE ARRAY 'BUFFER' OF FORTY WORDS.
    PACK,
    PUTCODE,
    0←FLAG,

    COMPARE,

//   END PSUEDO-OP ENCOUNTERD, PASS CONTROL TO ASSEMBLER
//   FOR FURTHER PROCESSING.
    FLAG=0 $ CALL ASMB. ;

    START.

  CONT:

//   * MACRO DEFINITION PHASE *
//  INCREMENT THE MACRO DEFINITION LEVEL COUNTER

    MACDLC+01←MACDLC,

//   READ ASSEMBLY LANGUAGE INPUT TEXT.
    0←I,
    AINPUT,
    0←I,
//   EXTRACT THE MACRO NAME FROM THE INPUT TEXT.
    EXTRACT,

    0←KOUNT,

//   COMPUTE THE HASH CODE FOR THE ENTRY.
    HASH,
  KLOOP:
//   CHECK FOR COLLISION OF HASH CODE.
    HASHTB[IHASHPTR]=0 $ JLOOP. ;
    IHASHPTR+01←IHASHPTR,
    KLOOP.
  JLOOP:
//   CHECK FOR MACRO NAME TABLE LENGTH
    0200<MANTC $ MAN. ;
//   SAVE THE POINTER TO MACRO NAME TABLE IN HASH TABLE
    MANTC←HASHTB[IHASHPTR],
//   SAVE THE MACRO NAME WITH ASSOCIATED POINTER TO MACRO
//   DEFINITION TABLE IN MACRO NAME TABLE 'AMACNT'.
  ALOOP:

    KOUNT=05 $ FLOOP. ;

    RESULT[KOUNT]←AMACNT[MANTC],

    MANTC+01←MANTC,

    KOUNT+01←KOUNT,

    ALOOP.

  FLOOP:
```

```
 //     SAVE THE MACRO DEFINITION TABLE POINTER IN THE MACRO
 //     NAME TABLE.
       MADTC←AMACNT[MANTC],

       MANTC+01←MANTC,
 //     PREPARE MACRO DEFINITION ARGUMENT LIST ARRAY WITH INDEX
 //     TO THEIR RELATIVE POSITION.
       PRARG,
   BACK:

       0←I,
 //     READ ASSEMBLY LANGUAGE INPUT TEXT, WHICH IS MACRO
 //     DEFINITION BODY.
       AINPUT,

 //     SUBSTITUTE THE INDEX FOR MACRO DEFINITION ARGUMENTS.
       SINOT,
       0←KOUNT,

 //     PACK EIGHTY CHARACTERS IN ARRAY 'SOURCE' INTO ARRAY
 //     'BUFFER' OF FORTY WORDS.
       PACK,
//     MACRO NAME CARD IS ENTERED IN THE MACRO DEFINITION TABLE

//     MACRO PSUEDO-OP

   BLOOP:

 //     CHECK FOR MACRO DEFINITION TABLE LENGTH
       02000<MADTC $ MAD. ;
 //     SAVE THE MACRO DEFINITION BODY IN MACRO DEFINITION
 //     TABLE.
       KOUNT=050 $ GLOOP. ;
       BUFFER[KOUNT]←AMACDT[MADTC],
       KOUNT+01←KOUNT,

 //     UPDATE MACRO DEFINITION TABLE COUNTER- MADTC.
       MADTC+01←MADTC,

       BLOOP.
   GLOOP:
 //     CHECK FOR MACRO DEFINITION WITHIN MACRO.
       0←I,
       EXTRACT,
       02←FLAG,
       COMPARE,
       FLAG=02 $ CHECK. ;
 //     CHECK FOR 'MEND' PSUEDO-OP.
       01←FLAG,

       COMPARE,

       FLAG=0 $ SKIP. ;

       BACK.

//     MEND PSUEDO-OP  YES

 //     DECREMENT MACRO DEFINITION LEVEL COUNTER.
   SKIP:

 //     CHECK FOR SAVING ALL MACRO DEFINITIONS.
       MACDLC-01←MACDLC,
```

```
       MACDLC=0 $ START. ;

 //    MACRO PSUEDO-OP  YES

       BACK.

    CHECK:

 //     INCREMENT MACRO DEFINITION LEVEL COUNTER.
       MACDLC+01←MACDLC,
       BACK.
 //     PROCESS ERROR CONDITION
    MAN:
       02←MESSAGE[010],
       ERROR,
       CALL ASMB.
    MAD:
       03←MESSAGE[010],
       ERROR,

    CALL ASMB:

       CLSEFILE,
       JOBFILE,
       EXEC,
 $     ,DEF,ENDPROG,,
 $     ,DEF,012,,
 $     ,DEF,ASMB,I,
 $     ,DEF,LUOBJECT,,
 $     ,DEF,LUOUTPUT,,
 $     ,DEF,LUPUNCH,,
 $     ,DEF,LPPAGE,,
 $     ,DEF,LGO,,
    ENDPROG:
       ..

       SUBROUTINE, AINPUT

 //*************************************************************
 //*                                                            *
 //*    PURPOSE OF THE MODULE                                   *
 //*                                                            *
 //*    TO SUBSTITUTE MACRO CALL ARGUMENTS FOR MACRO            *
 //*    DEFINITION ARGUMENTS IN CASE OF MACRO EXPANSION PHASE   *
 //*    OR TO READ ASSEMBLY LANGUAGE INPUT TEXT                 *
 //*                                                            *
 //*    USAGE                                                   *
 //*                                                            *
 //*    AINPUT,                                                 *
 //*                                                            *
 //*    MODULES REQUIRED                                        *
 //*                                                            *
 //*    AGO                                                     *
 //*    AIFGO                                                   *
 //*    AIFS                                                    *
 //*    EXTRACT                                                 *
 //*    ERROR                                                   *
 //*    SUBARG                                                  *
 //*    UNPAK                                                   *
 //*                                                            *
 //*    SYSTEM MODULES REQUIRED                                 *
```

```
//*                                                                    *
//*    PUTCODE                                                         *
//*                                                                    *
//*********************************************************************
//      EXTERNALS
     "P.IN",
     "P.OUT",
     "P.HLT",
     AGOS,
     AIFGO,
     AIFS,
     AMACDT,
     ANEG1,
     BLANKBLANK,
     BUFFER,
     BUFF LIM,
     COMPARE,

     EXTRACT,
     EFLAG,
     ERROR,
     FLAG,
     I,
     LUINP,
     LUOUT,
     MACDLC,
     MESSAGE,
     NOARG,
     PACK,
     PUTCODE,
     RESULT,
     S,
     SOURCE,
     STKPT,
     SUBARG,
     TAG,
     UNPAK,
     0,
     01,
     02,
     05,
     010,
     0110,
     0200,
     ;;

//   LOCALS

//      GFLAG- FLAG TO INDICATE A CALL TO PSUEDO-OP'S AIF,AGO
     *GFLAG,
//     TEMPORARY VARIABLES
     INDEX,

     IPTR,
     ITEMP,
     KOUNT,
//     PTR- A TEMPORARY VARIABLE USED TO SAVE THE POINTER TO
//          MACRO DEFINITION BODY.
     *PTR,
//     STAR- SAVES ASCII CODE FOR THE CHARACTER '*'
     STAR=052,

     ;;
```

```
 //     PROGRAM

        AINPUT: ?

 //       INITIALIZE TEMPORARY VARIABLES
        0←ITEMP,

        0←INDEX,

     RECUR:
 //       EXIT IF ERROR FLAG IS SET
        EFLAG=01 $ EXIT INPUT. ;
 //        CHECK FOR STACK LIMIT
        0200<STKPT $ OVERFLOW. ;
 //       CHECK FOR MACRO EXPANSION PHASE. VALUE OF STACK POINTER
 //       STKPT IS -1
        0←KOUNT,
        STKPT=ANEG1 $ EXPAND. ;

        STKPT+01←ITEMP,

 //       RETRIEVE THE MACRO DEFINITION TABLE POINTER.
        S[ITEMP]←IPTR,
        S[ITEMP]+BUFF LIM←S[ITEMP],
 //       TRANSFER INPUT ASSEMBLY LANGUAGE TEXT STORED IN MACRO
 //       DEFINITION TABLE TO ARRAY 'BUFFER' OF FORTY WORDS.
     BLOOP:
        KOUNT=BUFF LIM $ ALOOP. ;
        AMACDT[IPTR]←BUFFER[KOUNT],
        IPTR+01←IPTR.
        KOUNT+01←KOUNT,
        BLOOP.
     ALOOP:
 //       UNPAK THE CONTENTS OF THE ARRAY 'BUFFER'.
        UNPAK,
        0←TAG,
        0←I,
 //       EXTRACT THE OP-CODE FIELD.
        EXTRACT,
 //     CHECK FOR MEND PSUEDO-OP

        01←FLAG,

        COMPARE,

        FLAG=0 $ CHECK. ;
 //       SUBSTITUTE THE ASSOCIATED MACRO CALL ARGUMENTS FOR THE
 //       MACRO DEFINITION ARGUMENTS.
        SUBARG,
        0←I,
        02←GFLAG,
 //       CHECK FOR PSUEDO-OP'S 'AIF','AGO'.
        AIFGO,
        GFLAG=01 $ CALL. ;
        GFLAG=0 $ CALL AIFS. ;
        EXIT INPUT.
 //         PROCESS 'AIF' PSUEDO-OP
     CALL AIFS:
        IPTR←PTR,
        AIFS,
        RECUR.
 //         PROCESS 'AGO' PSUEDO-OP
```

```
CALL:
   IPTR←PTR,
   AGOS,
   RECUR.
CHECK:
      .
//      CHECK FOR MACRO CALL WITHIN MACRO.
   MACDLC=0 $ POP. ;
   EXIT INPUT.
//      POP BACK TO PREVIOUS STACK FRAME.
POP:
   STKPT←INDEX,
   STKPT-S[INDEX]←NOARG,
   NOARG-01←NOARG,
   S[INDEX]←STKPT,
   RECUR.
EXPAND:

//    UNPACK THE SOURCE INPUT OF EIGHTY CHARACTERS

   0←KOUNT,
//      BLANK FILL THE ARRAY 'BUFFER'.
CLOOP:
   KOUNT=BUFF LIM $ DLOOP. ;
   BLANKBLANK←BUFFER[KOUNT],
   KOUNT+01←KOUNT,
   CLOOP.
DLOOP:
//      READ ASSEMBLY LANGUAGE INPUT TEXT.
   >BUFFER<
//      UNPACK THE CONTENTS OF ARRAY 'BUFFER'.
   UNPAK,

   0←INDEX,

//    PROCESS COMMENTS

   SOURCE[INDEX]=STAR $ COMMENT. ;

   EXIT INPUT.

COMMENT:

//      PACK THE CONTENTS OF ARRAY 'SOURCE'.
   PACK,
//      OUTPUT THE COMMENT STATEMENT ON TO THE OUTPUT FILE.
   PUTCODE,
   EXPAND.

OVERFLOW:
//       PROCESS ERROR CONDITION
   05←MESSAGE[010],
   ERROR,
EXIT INPUT:

   0←I,
   ,
```

```
     SUBROUTINE, AIFGO
//**********************************************************
//*                                                        *
//*    PURPOSE OF THE MODULE                               *
//*                                                        *
//*      TO PROCESS PSUEDO-OP 'AIF'.                       *
//*    'AIF','AGO'.                                        *
//*                                                        *
//*    USAGE                                               *
//*                                                        *
//*    AIFGO,                                              *
//*                                                        *
//*    MODULES REQUIRED                                    *
//*                                                        *
//*    ERROR                                               *
//*    EXTRACT                                             *
//*                                                        *
//**********************************************************
//   EXTERNALS
     EFLAG,
     ERROR,
     EXTRACT,
//   GFLAG- FLAG VALUE TO INDICATE CALL TO PSUEDO-OP'S 'AIF',
//   'AGO'.
//         GFLAG=0 INDICATES CALL TO PSUEDO-OP 'AIF'
//         GFLAG=01 INDICATES CALL TO PSUEDO-OP 'AGO'
     GFLAG,
     I,
     LUINP,
     LUOUT,
     MESSAGE,
     RESULT,
     SOURCE,
     0,
     01,
     02,
     03,
     05,
     010,
     ;;
//   LOCALS
     CFLAG,
//   IFGO- ARRAY SAVING CHARACTER 'A I F G O'
     IFGO[5]=0101,0111,0106,0107,0117,
//     TEMPORARY VARIABLES
     ITEMP,
     KOUNT,
     ;;
//   PROGRAM
     AIFGO: ?
     0←CFLAG,
     0←I,
//   INITIALIZE TEMPORARY VARIABLES
     0←ITEMP,
     0←KOUNT,
//   EXTRACT THE OP-CODE FIELD OF SOURCE INPUT.
     EXTRACT,
//   EXIT IF ERROR FLAG IS SET
     EFLAG=01 $ EXIT AIFGO. ;
//   COMPARE FOR MATCH WITH PSUEDO-OP'S 'AIF','AGO'
  ALOOP:
     KOUNT=03 $ SET FLAG. ;
     RESULT[KOUNT]=IFGO[ITEMP] $ BLOOP. ;
```

```
//    NOT PSUEDO-OP'S 'AIF','AGO'
      KOUNT=0 $ EXIT AIFGO. ;
      CFLAG=01 $ EXIT AIFGO. ;
      01←CFLAG,
      03←ITEMP,
      ALOOP.
  BLOOP:
      ITEMP+01←ITEMP,
      KOUNT+01←KOUNT,
      ALOOP.
  SET FLAG:
      ITEMP=05 $ SET GO. ;
//    RETURN FLAG VALUE FOR 'AIF' PSUEDO-OP
      0←GFLAG,
      EXIT AIFGO.
  SET GO:
//    RETURN FLAG VALUE FOR 'AGO' PSUEDO-OP
      01←GFLAG,
  EXIT AIFGO:
      ,
      ..

      SUBROUTINE, AGOS
//***********************************************************
//*                                                         *
//*      PURPOSE OF THE MODULE                              *
//*                                                         *
//*      TO PROCESS UNCONDITIONAL BRANCH PSUEDO-OP 'AGO'    *
//*                                                         *
//*    METHOD                                               *
//*                                                         *
//*    SCAN MACRO DEFINITION BODY FOR MATCH WITH LABEL      *
//*    SPECIFIED IN PSUEDO-OP STATEMENT AGO.                *
//*                                                         *
//*    USAGE                                                *
//*                                                         *
//*    AGOS,                                                *
//*                                                         *
//*    MODULES REQUIRED                                     *
//*                                                         *
//*    ERROR                                                *
//*    EXTRACT                                              *
//*    UNPAK                                                *
//*                                                         *
//***********************************************************
//     EXTENALS
      AMACDT,
      BLANKBLANK,
      BUFF LIM,
      BUFFER,
      COMPARE,
      EFLAG,
      ERROR,
      EXTRACT,
      FLAG,
      I,
      LUINP,
      LUOUT,
      MESSAGE,
      PTR,
      RESULT,
     -S,
      SOURCE,
```

```
     STKPT,
     TAG,
     UNPAK,
     0,
     01,
     05,
     013,
     0110,
     ;;
//    LOCALS
//      TEMPORARY VARIABLES
     ITEMP,
     JTEMP,
     KOUNT,
//      LABEL- ARRAY TO SAVE THE LABEL SPECIFIED IN AGO PSUEDO-OP
//             STATEMENT.
     LABEL[5],
     ;;
//    PROGRAM
     AGOS: ?
     0←TAG,
     0←KOUNT,
//      EXTRACT THE LABEL SPECIFIED IN 'AGO' PSUEDO-OP
     EXTRACT,
//      EXIT IF ERROR FLAG IS SET
//       ZERO FILL ARRAY 'LABEL'
  ALOOP:
     KOUNT=05 $ BLOOP. ;
     0←LABEL[KOUNT],
     KOUNT+01←KOUNT,
     ALOOP.
  BLOOP:
     0←KOUNT,
//      SAVE THE LABEL SPECIFIED IN AGO PSUEDO-OP STATEMENT IN
//       ARRAY 'LABEL'
  CLOOP:
     KOUNT=05 $ DLOOP. ;
     RESULT[KOUNT]←LABEL[KOUNT],
     KOUNT+01←KOUNT,
     CLOOP.
  DLOOP:
     0←KOUNT,
//      SAVE THE POINTER TO MACRO DEFINITION BODY IN THE MACRO
//      DEFINITION TABLE.
     PTR←ITEMP,
//      SCAN FOR THE LABELED STATEMENT.
  LAB:
     EFLAG=01 $ EXIT AGOS. ;
     AMACDT[ITEMP]=BLANKBLANK $ GLOOP. ;
     ITEMP←JTEMP,
  ELOOP:
     KOUNT=BUFF LIM $ FLOOP. ;
     BLANKBLANK←BUFFER[KOUNT],
     AMACDT[JTEMP]←BUFFER[KOUNT],
     JTEMP+01←JTEMP,
     KOUNT+01←KOUNT,
     ELOOP.
  GLOOP:
//      UPDATE POINTER TO POINT TO NEXT ASSEBLY LANGUAGE
//      STATEMENT IN THE MACRO DEFINITION BODY.
     ITEMP+BUFF LIM←ITEMP,
     0←KOUNT,
     LAB.
```

```
 FLOOP:
    UNPAK,
    0←KOUNT,
    0←I,
    EXTRACT,
//        CHECK FOR 'MEND' PSUEDO-OP
    01←FLAG,
    COMPARE,
    FLAG=0 $ NOLABEL. ;
//      COMPARE FOR MATCH WITH THE LABEL SPECIFIED IN AGO
//      PSUEDO-OP STATEMENT.
 COMP:
    KOUNT=05 $ UPDATE. ;
    LABEL[KOUNT]=RESULT[KOUNT] $ CONT. ;
    ITEMP+BUFF LIM←ITEMP,
    0←KOUNT,
    LAB.
 CONT:
    KOUNT+01←KOUNT,
    COMP.
//      UPDATE POINTER TO MACRO DEFINITION TABLE ON STACK.
 UPDATE:
    ITEMP+BUFF LIM←ITEMP,
    STKPT+01←JTEMP,
    ITEMP←S[JTEMP],
    EXIT AGOS.
 NOLABEL:
    01←MESSAGE[010],
    ERROR,
 EXIT AGOS:
    ,

    SUBROUTINE, AIFS
//***************************************************************
//*                                                             *
//*      PURPOSE OF THE MODULE                                  *
//*                                                             *
//*      TO PROCESS PSUEDO-OP 'AIF'                             *
//*                                                             *
//*      METHOD                                                 *
//*                                                             *
//*      EVALUATE THE PREDICATE, A TRUE CONDITION RESULTS IN    *
//*      BRANCH TO THE LABEL SPECIFIED IN THE AIF PSUEDO-OP     *
//*      STATEMENT BY UPDATING THE POINTER TO MACRO DEFINITION*
//*      BODY SAVED IN MACRO DEFINITION TABLE.                  *
//*      A FALSE CONDITION RESULTS IN CONTINUATION OF           *
//*      PROCESSING.                                            *
//*                                                             *
//*      USAGE                                                  *
//*                                                             *
//*      AIFS,                                                  *
//*                                                             *
//*      MODULES REQUIRED                                       *
//*                                                             *
//*      ERROR                                                  *
//*      EXTRACT                                                *
//*      IFTEST                                                 *
//*      RELATIONAL                                             *
//*                                                             *
//***************************************************************
/        EXTERNALS
    EFLAG,
```

```
      ERROR,
      IFTEST,
      MESSAGE,
      SOURCE,
      RESULT,
      RFLAG,
      EXTRACT,
      RELATIONAL,
      AGOS,
      TAG,
      I,
      O,
      O1,
      O2,
      O5,
      O6,
      O10,
      O12,
      LUINP,
      LUOUT,
      ;;
//    LOCALS
//       AIFFLAG- A FLAG TO INDICATE WHETHER EVALUATED PREDICATE
//                IS TRUE OR FALSE
      *AIFFLAG,
//       ASCII CODE FOR THE CHARACTER '('
      LEFT PARANTHESIS=050,
//       OPERAND- AN ARRAY FOR SAVING TWO OPERANDS OF A PREDICATE
      *OPERAND[12],
//       ASCII CODE FOR CHARACTER ')'
      RIGHT PARANTHESIS=051,
//       TEMPORARY VARIABLES
      KOUNT,
      J,
     ITEMP,
      CHECK FLAG,
      LOGICAL,
//       LOCAL OCTAL CONSTANTS
      *O8=010,
      *O9=011,
      ;;
//    PROGRAM
      AIFS: ?
//       INITIALIZE TEMPORARY VARIABLES
      0←J,
      0←ITEMP,
      0←LOGICAL,
      0←RFLAG,
   BEGIN:
//       EXIT IF ERROR FLAG IS SET
      EFLAG=O1 $ EXIT AIFS. ;
      O1←TAG,
      0←KOUNT,
      EXTRACT,
//       PROCEED WITH THE EVALUATION OF THE PREDICATE IF LEFT
//       PARANTHESIS IS ENCOUNTERED.
      RESULT[KOUNT]=LEFT PARANTHESIS $ SCAN. ;
//       BRANCH TO LABEL RETURN TO TAKE APPROPRIATE ACTION
//       ON ENCOUNTERING RIGHT PARANTHESIS.
      RESULT[KOUNT]=RIGHT PARANTHESIS $ RETURN. ;
//       RETRIEVE THE OPERATOR
      RELATIONAL,
//       BRANCH TO LABEL SCAN IF RELATIONAL OPERATOR.
```

```
      RFLAG<08 $ SCAN. ;
//       SAVE THE LOGICAL OPERATOR.
      RFLAG←LOGICAL,
      BEGIN.
   SCAN:
      0←KOUNT,
//       ZERO FILL THE ARRAY OPERAND
   ZERO:
      KOUNT=012 $ START. ;
      0←OPERAND[KOUNT],
      KOUNT+01←KOUNT,
      ZERO.
   START:
//       INITIALIZE TEMPORARY VARIABLES
      0←KOUNT,
      0←J,
      0←ITEMP,
   LOOP0:
      01←TAG,
      0←KOUNT,
//       EXTRACT THE OPERAND OF THE PREDICATE
      EXTRACT,
//       SAVE THE OPERAND IN THE ARRAY OPERAND
   LOOP1:
      KOUNT=05 $ EXIT1. ;
      RESULT[KOUNT]←OPERAND[ITEMP],
      KOUNT+01←KOUNT,
      ITEMP+01←ITEMP,
      LOOP1.
   EXIT1:
      01←TAG,
      J=01 $ EXIT2. ;
      J+01←J,
//       EXTRACT THE RELATIONAL OPERATOR
      EXTRACT,
//       RETRIEVE THE FLAG VALUE OF THE RELATIONAL OPERATOR
      RELATIONAL,
      LOOP0.
   EXIT2:
//       IF LOGICAL OPERATOR 'AND' BRANCH TO LABEL SET AND FLAG
      LOGICAL=08 $ SET AND FLAG. ;
//       IF LOGICAL OPERATOR 'OR' BRANCH TO LABEL SET OR FLAG
      LOGICAL=09 $ SET OR FLAG. ;
//       CHECK FOR SATISFYING THE CONDITION
      IFTEST,
//       SAVE THE CONDITION FLAG VALUE
      AIFFLAG←CHECK FLAG,
      BEGIN.
   SET AND FLAG:
      IFTEST,
      CHECK FLAG=01 $ CONT1. ;
      0←CHECK FLAG,
      BEGIN.
   CONT1:
      AIFFLAG=0 $ CONT3. ;
      BEGIN.
   CONT3:
      0←CHECK FLAG,
      BEGIN.
   SET OR FLAG:
      IFTEST,
      AIFFLAG=01 $ CONT2. ;
      CHECK FLAG=01 $ CONT2. ;
```

```
    BEGIN.
  CONT2:
    O1←CHECK FLAG,
    BEGIN.
  RETURN:
    CHECK FLAG=O1 $ CALL AGOS. ;
//       PREDICATE EVALUATED TO FALSE, RETURN FROM THE MODULE
    EXIT AIFS.
//      PREDICATE EVALUATED TO TRUE, BRANCH TO APPROPRIATE LABEL
  CALL AGOS:
    AGOS,
    EXIT AIFS:
    ,
    ..

    SUBROUTINE, RELATIONAL
//**********************************************************************
//*                                                                   *
//*      PURPOSE OF THE MODULE                                        *
//*                                                                   *
//*      TO RETURN APPROPRIATE FLAG VALUE FOR RELATIONAL AND          *
//*      LOGICAL OPERATORS(NE,EQ,GT,GE,LT,LE,AND,OR)                  *
//*                                                                   *
//*      USAGE                                                        *
//*                                                                   *
//*      RELATIONAL,                                                  *
//*                                                                   *
//*      MODULES REQUIRED                                             *
//*                                                                   *
//*      ERROR                                                         *
//*                                                                   *
//**********************************************************************
//    EXTERNALS
    EFLAG,
    ERROR,
    LUINP,
    LUOUT,
    MESSAGE,
    RESULT,
    O,
    O1,
    O2,
    O3,
    O4,
    O5,
    O6,
    O8,
    O9,
    O10,
    ;;
//    LOCALS
//      RFLAG- SAVES APPROPRIATE FLAG VALUE FOR RELATIONAL AND
//             LOGICAL OPERATORS.
    *RFLAG,
//      TEMPORARY VARIABLES.
    ITEMP,
    KOUNT,
//      'OPER'- AN ARRAY WITH CHARACTERS 'A D E G L N O Q R T'
    OPER[10]=0101,0104,0105,0107,0114,0116,0117,0121,0122,0124,
    ;;
//    PROGRAM
    RELATIONAL: ?
    O←KOUNT,
```

```
//       BRANCH TO APPROPRIATE LABEL.
      RESULT[KOUNT]=OPER[KOUNT] $ SET AND FLAG. ;
      O2←ITEMP,
      RESULT[KOUNT]=OPER[ITEMP] $ SET OR FLAG. ;
      O2←ITEMP,
      RESULT[KOUNT]=OPER[ITEMP] $ LOOP1. ;
      O3←ITEMP,
      RESULT[KOUNT]=OPER[ITEMP] $ LOOP2. ;
      O4←ITEMP,
      RESULT[KOUNT]=OPER[ITEMP] $ LOOP3. ;
//       RELATIONAL OPERATOR 'NE', RETURN RFLAG=01.
      O1←RFLAG,
      EXIT RELATIONAL.
  LOOP3:
      O1←KOUNT,
      O2←ITEMP,
      RESULT[KOUNT]=OPER[ITEMP] $ LOOP5. ;
//       RELATIONAL OPERATOR 'LT', RETURN RFLAG=04.
      O4←RFLAG,
      EXIT RELATIONAL.
  LOOP5:
//       RELATIONAL OPERATOR 'LE', RETURN RFLAG=05.
      O5←RFLAG,
      EXIT RELATIONAL.
  LOOP2:
      O1←KOUNT,
      O2←ITEMP,
      RESULT[KOUNT]=OPER[ITEMP] $ LOOP4. ;
//       RELATIONAL OPERATOR 'GT', RETURN RFLAG=03.
      O3←RFLAG,
      EXIT RELATIONAL.
  LOOP4:
//       RELATIONAL OPERATOR 'GE', RETURN RFLAG=02.
      O2←RFLAG,
      EXIT RELATIONAL.
  LOOP1:
//       RELATIONAL OPERATOR 'EQ', RETURN RFLAG=0.
      O←RFLAG,
      EXIT RELATIONAL.
  SET OR FLAG:
//       LOGICAL OPERATOR 'OR', RETURN RFLAG=09
      O9←RFLAG,
      EXIT RELATIONAL.
  SET AND FLAG:
//     LOGICAL OPERATOR 'AND', RETURN RFLAG=08.
      O8←RFLAG,
  EXIT RELATIONAL:
      ,
      ..

      SUBROUTINE, STACK

//**************************************************************
//*                                                            *
//*     PURPOSE OF THE MODULE                                  *
//*                                                            *
//*     TO SET UP MACRO CALL ARGUMENT LIST AND TO HANDLE MACRO *
//*     CALLS WITHIN MACRO                                     *
//*                                                            *
//*     UASGE                                                  *
//*                                                            *
//*     STACK,                                                 *
//*                                                            *
```

```
//*    METHOD                                                   *
//*                                                             *
//*    STACK IS IMPLEMENTED USING AN ARRAY 'S' WITH LAST IN     *
//*    FIRST OUT STRATEGY.                                      *
//*                                                             *
//*    MODULES REQUIRED                                         *
//*                                                             *
//*    ERROR                                                    *
//*    EXTRACT                                                  *
//*                                                             *
//*************************************************************
//     EXTERNALS

       EFLAG,
       ERROR,
       EXTRACT,
       I,
       LUINP,
       LUOUT,
       MDTP,
       MESSAGE,
       NOARG,
       RESULT,
       STKPT,
       TAG,
       O,
       O1,
       O2,
       O5,
       O110,
       ;;

//     LOCALS

//        TEMPORARY VARIABLES
       ITEMP,

       KOUNT,

//        S- AN ARRAY FOR IMPLEMENTING STACK
       *S[200],
       ;;

//     PROGRAM

       STACK: ?

//       SAVE THE STACK POINTER.
       STKPT+NOARG←ITEMP,

       ITEMP+O1←ITEMP,
       STKPT←S[ITEMP],

//       SAVE THE STACK POINTER FOR THE PREVIOUS STACK FRAME
       ITEMP←STKPT,
       STKPT+O1←ITEMP,

//         SAVE THE MACRO DEFINITION TABLE POINTER IN THE STACK.
       MDTP←S[ITEMP],
    ARGSUB:

       @←TAG,
       I=O110 $ EXIT STACK. ;
```

```
//        EXTRACT THE ACTUAL ARGUMENT FROM THE MACRO CALL STATEMENT.
     EXTRACT,

     0←KOUNT,

//      IF ALL THE MACRO CALL ARGUMENTS ARE SAVED ON THE STACK,
//      EXIT FROM THE MODULE.
     RESULT[KOUNT]=0 $ EXIT STACK. ;

  ALOOP:

     KOUNT=05 $ BLOOP. ;

     ITEMP+01←ITEMP,

     0←S[ITEMP],
//      SAVE THE MACRO CALL ARGUMENT IN THE STACK.
     RESULT[KOUNT]←S[ITEMP],

     KOUNT+01←KOUNT,

     ALOOP.

  BLOOP:

     ARGSUB.

  EXIT STACK:

//      SAVE THE NUMBER OF ARGUMENTS IN THE MACRO CALL.
     ITEMP←NOARG,
     ;



     SUBROUTINE, HASH
//***********************************************************
//*                                                         *
//*  PURPOSE OF THE MODULE                                  *
//*                                                         *
//*  TO GENERATE HASH CODE FOR THE ENTRY TO BE SEARCHED IN  *
//*  IN MACRO NAME TABLE.                                   *
//*                                                         *
//*  METHOD                                                 *
//*                                                         *
//*  DIVISION METHOD                                        *
//*                                                         *
//*  USAGE                                                  *
//*                                                         *
//*  HASH,                                                  *
//*                                                         *
//*  MODULES REQUIRED                                       *
//*                                                         *
//*  ERROR                                                  *
//*                                                         *
//***********************************************************
//   EXTERNALS
     EFLAG,
     ERROR,
     LUINP,
     LUOUT,
```

```
      RESULT.
      0,
      01,
      05,
      036,
      ;;
//    LOCALS
//       IHASHPTR- SAVES HASH CODE GENERATED FOR THE ENTRY
      *IHASHPTR,
//       TEMPORARY VARIABLES
      ITEMP,
      JTEMP,
      KOUNT,
      MESSAGE,
       ;;
//    PROGRAM
      HASH: ?
//     INITIALIZE TEMPORARY VARIABLES.
      0←ITEMP,
      0←KOUNT,
//     OBTAIN THE POINTER TO LAST CHARACTER OF THE ENTRY.
   ILOOP:
      KOUNT=05 $ HLOOP. ;
      RESULT[KOUNT]=0 $ HLOOP. ;
      KOUNT+01←KOUNT,
      ILOOP.
   HLOOP:
      KOUNT-01←KOUNT,
//     COMPUTE THE SUM OF ASCII CODES OF FIRST AND LAST
//     CHARACTERS OF THE ENTRY.
      RESULT[0]+RESULT[KOUNT]←ITEMP,
//     COMPUTE THE REMAINDER BY DIVIDING THE ABOVE SUM
//     BY TABLE LENGTH.
//     (INTEGER DIVISION IS USED)
      ITEMP/036←JTEMP,
      JTEMP*036←JTEMP,
      ITEMP-JTEMP←IHASHPTR,
      ,
      ..

      SUBROUTINE, EXTRACT
//*************************************************************
//*                                                           *
//*    PURPOSE OF THE MODULE                                  *
//*    TO EXTRACT SUBSTRING FROM EIGHTY CHARACTERS SAVED IN   *
//*    ARRAY 'SOURCE'.                                        *
//*                                                           *
//*    METHOD                                                 *
//*                                                           *
//*    SCAN THE EIGHTY CHARACTERS FROM THE POSITION INDICATED *
//*    BY POINTER 'I' UPTILL A DELIMETER IS ENCOUNTERED.      *
//*    USAGE                                                  *
//*                                                           *
//*    EXTRACT,                                               *
//*                                                           *
//*    MODULES REQUIRED                                       *
//*                                                           *
//*    ERROR                                                  *
//*    SPCHR                                                  *
//*                                                           *
//*************************************************************
//    EXTERNALS
```

```
      BLANKBLANK,
      DOLLAR,
      EFLAG,
      ERROR,
      LUINP,
      LUOUT,
      MESSAGE,
      SAND,
      SOURCE,

      SPCHR,
      TAG,
      O,
      O1,
      O5,
      O10,
      O110,
      ;;

//     LOCALS

      *CONCT=042,
//       SAVE ASCII CHARACTER CODE FOR THE CHARACTER ','
      COMMA=054,
//      TEMPORARY FLAG VALUE
      FLAG,
//     I- POINTER TO POSITION OF CHARACTER TO BE EXTRACTED FROM
//        EIGHTY CHARACTER INPUT
     *I,
//     ISC- VARIABLE TO SAVE EACH INPUT CHARACTER IN ORDER TO
//          FIND ANY SPECIAL CHARACTER IN THE INPUT SOURCE
     *ISC,
//       TEMPORARY VARIABLES
      ITEMP,
      KOUNT,
//       RESULT- AN FOR SAVING EXTRACTED SUB-STRING.
     *RESULT[5],
//      SFLAG- FLAG VALUE TO INDICATE A SPECIAL CHARACTER
     *SFLAG,
      ;;

//    PROGRAM

      EXTRACT: ?

      O←KOUNT,

//    INITIALIZE THE ARRAY RESULT

   AINTZ:

      KOUNT=O5 $ START. ;

      O←RESULT[KOUNT],

      KOUNT+O1←KOUNT,

      AINTZ.

   START:

      O←KOUNT,
```

```
    0 ← FLAG,

 ANAMESETUP:

//      EXIT IF ERROR FLAG IS SET
    EFLAG=01 $ EXIT EXTRACT. ;
    I=0113 $ EXIT EXTRACT. ;
    0←SFLAG,
//      BRANCH TO APPROPRIATE LABEL ON ENCOUNTERING DELIMETERS
    KOUNT=05 $ EXIT EXTRACT.;

    SOURCE[I]=BLANKBLANK $ SKIP. ;
    SOURCE[I]=COMMA $ BINCT. ;
//       SAVE THE CHARACTER IN ISC
    SOURCE[I]←ISC,
//       CHECK FOR SPECIAL CHARACTER
    SPCHR,
    SOURCE[I]←RESULT[KOUNT],

    I+01←ITEMP,
//       CHECK FOR THE BEGINING OF MACRO DEFINITION ARGUMENTS
    SOURCE[ITEMP]=SAND $ AINCT. ;
    SOURCE[ITEMP]=DOLLAR $ AINCT. ;
    TAG=0 $ ALOOP. ;
//       BRANCH TO AINCT IF SPECIAL CHARACTER
    SFLAG=01 $ AINCT. ;
 ALOOP:
//       SET FLAG VALUE FOR NOORDINARY CHARACTER
    01← FLAG,

//      INCREMENT POINTER TO THE POSITION OF THE CHARACTER
    I+01 ← I,

    KOUNT+01 ← KOUNT,

    ANAMESETUP.

 BINCT:
    TAG=0 $ SKIP. ;
    SOURCE[I]←RESULT[KOUNT],
 AINCT:
    I+01←I,
//      RETURN ENCOUNTERED SPECIAL CHARACTER
    FLAG=0 $ EXIT EXTRACT. ;
    0←SFLAG,
    0←RESULT[KOUNT],
    I-01←I,
 SKIP:

//      DELIMETER ENCOUNTERED, RETURN SUB-STRING
    FLAG=01 $ EXIT EXTRACT. ;

    I+01←I,
//     CONTINUE SCANNING INPUT SOURCE STRING
    ANAMESETUP.

 EXIT EXTRACT:

    0←TAG,
    ,
```

```
     SUBROUTINE, SPCHR
//*********************************************************
//*                                                       *
//*      PURPOSE OF THE MODULE                            *
//*                                                       *
//*      TO RETURN APPROPRIATE FLAG VALUE FOR SPECIAL     *
//*      CHARACTER.                                       *
//*                                                       *
//*      USAGE                                            *
//*                                                       *
//*      SPCHR,                                           *
//*                                                       *
//*      MODULES REQUIRED                                 *
//*                                                       *
//*      ERROR                                            *
//*                                                       *
//*********************************************************
//   EXTERNALS
     DOLLAR,
     EFLAG,
     ERROR,
     ISC,
     LUINP,
     LUOUT,
     MESSAGE,
     SAND,
     SFLAG,
     O,
     O1,
     ;;
//   LOCALS
//      'CHR'- ARRAY WITH TWENTY SIX ASCII SPECIAL CHARACTERS.
     CHR[26]=040,041,042,043,044,045,046,047,050,051,052,053,054,
     055,056,057,072,073,074,075,076,077,0100,0133,0134,0135,
//      TEMPORARY VARIABLE.
     KOUNT,
//      TEMPORARY CONSTANT.
     O31=031,
     ;;
//   PROGRAM
     SPCHR: ?
     O←SFLAG,
//      IF SPECIAL CHARACTER IS AMPERSAND(&) RETURN.
     ISC=SAND $ EXIT SPCHR. ;
     ISC=DOLLAR $ EXIT SPCHR. ;
     O←KOUNT,
   ALOOP:
//      INPUT CHARACTER IS NOT A SPECIAL CHARACTER, RETURN
     KOUNT=O31 $ EXIT SPCHR. ;
     CHR[KOUNT]=ISC $ SET FLAG. ;
//      INPUT CHARACTER IS A SPECIAL CHARACTER, RETURN
//      FLAG VALUE, SFLAG=01.
     KOUNT+O1←KOUNT,
     ALOOP.
   SET FLAG:
     O1←SFLAG,
   EXIT SPCHR:


     SUBROUTINE, COMPARE
```

```
//*************************************************************
//*                                                          *
//*     PURPOSE OF THE MODULE                                *
//*                                                          *
//*     TO RETURN APPROPRIATE FLAG VALUE FOR PSUEDO-OP'S     *
//*     'MACRO', 'MEND', 'END'.                              *
//*                                                          *
//*     USAGE                                                *
//*                                                          *
//*     COMPARE,                                             *
//*                                                          *
//*     MODULES REQUIRED                                     *
//*                                                          *
//*     ERROR                                                 *
//*                                                          *
//*************************************************************
//    EXTERNALS

      EFLAG,
      ERROR,
      FLAG,
      LUINP,

      LUOUT,

      MESSAGE,
      RESULT,

       O,
       O1,
       O2,
       O3,
       O4,
       O5,
       O6,
      O10,
      ;;

//    LOCALS

//    INITIALIZE ARRAY CHAR WITH CHARACTERS M, A, C, R, O, E, N, D

      CHAR[8]=0115,0101,0103,0122,0117,0105,0116,0104,

//       TEMPORARY VARIABLES.
      CHECK,
      IK,
      KOUNT,
       ;;

//    PROGRAM

      COMPARE: ?

//       INITIALIZE TEMPORARY VARIABLES.
      O←KOUNT,

      O←IK,

//    JUMP TO ELOOP IF COMPARISON IS FOR PSUEDO-OP END

     . FLAG=O $ ELOOP.;
```

```
//      JUMP TO CLOOP IF COMPARISON IS FOR THE PSUEDO-OP MEND

        FLAG=01 $ CLOOP. ;

        04←CHECK,

   ALOOP:

//          CHECK FOR MACRO PSUEDO-OP, RETURN FLAG=0 IF SUCCESSFUL
//                                            FLAG=01 IF UNSUCCESSFUL
        KOUNT=CHECK $ EXIT COMPARE. ;
        RESULT[KOUNT]=CHAR[IK] $ BLOOP. ;

        01←FLAG,

        EXIT COMPARE.

   BLOOP:

        KOUNT+01←KOUNT,

        IK+01←IK,

        ALOOP.

//          CHECK FOR MEND PSUEDO-OP, RETURN FLAG=0 IF SUCCESSFUL
//                                           FLAG=01 IF UNSUCCESSFUL
   CLOOP:

        0←FLAG,

        04←IK,

        03←CHECK,

        RESULT[KOUNT]=CHAR[KOUNT] $ DLOOP. ;

        01←FLAG,

        EXIT COMPARE.

   DLOOP:

        KOUNT+01←KOUNT,

        IK+01←IK,

        ALOOP.

//          CHECK FOR END PSUEDO-OP, RETURN FLAG=0 IF SUCCESSFUL
//                                          FLAG=01 IF UNSUCCESSFUL
   ELOOP:

        0←FLAG,

        05←IK,

        03←CHECK,
   GLOOP:
        KOUNT=CHECK $ EXIT COMPARE. ;
        RESULT[KOUNT]=CHAR[IK] $ FLOOP. ;

        01←FLAG,
```

```
    EXIT COMPARE.

 FLOOP:

    KOUNT+01←KOUNT,

    IK+01←IK,

    GLOOP.
 EXIT COMPARE:

    0←IK,

    ,

    ..

    SUBROUTINE, SINOT
//**********************************************************
//*                                                        *
//*    PURPOSE OF THE MODULE                               *
//*                                                        *
//*    TO SUBSTITUTE THE INDEX FOR THE MACRO DEFINITON     *
//*    ARGUMENTS IN THE MACRO DEFINIYION BODY.             *
//*                                                        *
//*    USAGE                                               *
//*                                                        *
//*    SINOT,                                              *
//*                                                        *
//*    MODULES REQUIRED                                    *
//*                                                        *
//*    ERROR                                               *
//*    EXTRACT                                             *
//*                                                        *
//**********************************************************
//      EXTERNALS
    ALA,
    EFLAG,
    ERROR,
    EXTRACT,
    I,
    LUINP,
    LUOUT,
    MESSAGE,
    RESULT,
    SAND,
    SOURCE,
    0,
    01,
    04,
    05,
    06,
    010,
    0110,
    0144,
    ;;
//  LOCALS
//      TEMPORARY VARIABLES
    IK,
    ITEMP,
    J,
```

```
      JTEMP,
      KOUNT,
      *TAG,
      *DOLLAR=044,
      ;;
//    PROGRAM
      SINOT: ?
//      INITIALIZE TEMPORARY VARIABLES.
      0←I,
      0←ITEMP,
      0←J,
      0←JTEMP,
   BINTZ:
//      EXIT IF ERROR FLAG IS SET
      EFLAG=01 $ EXIT SINOT. ;
      0←ITEMP,
      0←KOUNT,
      01←TAG,
//      EXTRACT THE MACRO DEFINITION ARGUMENT OF ASSEMBLY
      EXTRACT,
      I=0110 $ EXIT SINOT. ;
      RESULT[KOUNT]=0 $ EXIT SINOT. ;
      RESULT[KOUNT]=SAND $ SCAN. ;
      BINTZ.
//      RETRIEVE THE INDEX FOR MACRO DEFINITION ARGUMENT BY
//      SCANNING THE ARGUMENT LIST ARRAY 'ALA'.
   SCAN:
      ITEMP=0110 $ BLOOP. ;
      KOUNT=05 $ EXCHA. ;
      RESULT[KOUNT]=0 $ EXCHA. ;
      RESULT[KOUNT]=ALA[ITEMP] $ ALOOP. ;
      05-KOUNT←JTEMP,
      JTEMP+01←JTEMP,
      JTEMP+ITEMP←ITEMP,
      0←KOUNT,
      SCAN.
   ALOOP:
      KOUNT+01←KOUNT,
      ITEMP+01←ITEMP,
      SCAN.
   EXCHA:
      05-KOUNT←J,
      ITEMP+J←ITEMP,
      I-01←IK,
//      SUBSTITUTE INDEX FOR THE MACRO DEFINITION ARGUMENT.
      ALA[ITEMP]←SOURCE[IK],
      I-KOUNT←IK,
      DOLLAR←SOURCE[IK],
   BLOOP:
      BINTZ.
   EXIT SINOT:
      01←TAG,
      ,
      ...

     SUBROUTINE, PRARG

//***********************************************************
//*    PURPOSE OF THE MODULE                                *
//*    TO PREPARE MACRO DEFINITION ARGUMENT LIST ARRAY WITH *
//*    THEIR ASSOCIATED RELATIVE POSITION IN THE STATEMENT  *
//*                                                         *
//*    USAGE                                                *
```

```
//*
//*    PRARG,                                                   *
//*                                                             *
//*    MODULES REQUIRED                                         *
//*                                                             *
//*    ERROR                                                    *
//*    EXTRACT                                                  *
//*      .                                                      *
//*************************************************************
//    EXTERNALS
     EFLAG,
     ERROR,
     EXTRACT,

     I,
     LUINP,
     LUOUT,
     MESSAGE,
     RESULT,

     SOURCE,

     TAG,
     O,
     O1,
     O4,
     O5,
     O10,
     O110,
     ;;

//   LOCALS

//    'ALA'- ARRAY OF EIGHTY WORDS FOR SAVING MACRO DEFINITION
//    ARGUMENTSS WITH THEIR ASSOCIATED RELATIVE POSITION.
     *ALA[80],
//     TEMPORARY VARIABLES
     ITEMP,
     J,
     KOUNT,
     *O120=0120,
     ;;

//   PROGRAM

     PRARG: ?

//      INITIALIZE TEMPORARY VARIABLES
     O←ITEMP,
     O←J,
  CHECK:

//      CHECK FOR ARGUMENT LIST ARRAY LIMIT
     O120<ITEMP $ ALAFULL. ;
//     EXIT IF ERROR FLAG IS SET
     EFLAG=O1 $ EXIT PRARG. ;
     O←KOUNT,

     O←TAG,
     I=O110 $ EXIT PRARG. ;
//     EXTRACT THE DUMMY ARGUMENTS FROM THE SOURCE INPUT.
     EXTRACT,
```

```
     RESULT[KOUNT]=0 $ EXIT PRARG. ;

//      STORE THE DUMMY ARGUMENT IN THE ARRAY ALA.
  ALOOP:

     KOUNT=05 $ ADJUS. ;

     RESULT[KOUNT]←ALA[ITEMP],

     KOUNT+01←KOUNT,

     ITEMP+01←ITEMP,

     ALOOP.

//      SAVE THE INDEX OF THE DUMMY ARGUMENT IN THE ARRAY ALA.
  ADJUS:

     J←ALA[ITEMP],

     ITEMP+01←ITEMP,
//     INCREMENT THE INDEX FOR MACRO DEFINITION ARGUMENT.
     J+01←J,
     CHECK.

//      PROCESS ERROR CONDITION
  ALAFULL:
     04←MESSAGE[010],
     ERROR,
  EXIT PRARG:

     ,

     ..


     SUBROUTINE, SUBARG
//*************************************************************
//*                                                           *
//*    PURPOSE OF THE MODULE                                  *
//*                                                           *
//*    TO SUBSTITUTE MACRO CALL ARGUMENTS FOR MACRO           *
//*    DEFINITION ARGUMENTS.                                  *
//*                                                           *
//*    USAGE                                                  *
//*                                                           *
//*    SUBARG,                                                *
//*                                                           *
//*    MODULES REQUIRED                                       *
//*                                                           *
//*    ERROR                                                  *
//*    EXTRACT,                                               *
//*                                                           *
//*************************************************************
//     EXTERNALS
     BLANKBLANK,
     DOLLAR,
     CONCT,
     EFLAG,
     ERROR,
     EXTRACT,
     I,
     ISC,
```

```
      LUINP,
      LUOUT,
      MESSAGE,
      RESULT,
      S,
      SAND,
      SFLAG,
      SOURCE,
      STKPT,
      TAG,
      0,
      01,
      02,
      05,
      010,
      0110,
      ;;
//    LOCALS
//      TEMPORARY VARIABLES
      IK,
      IPTR,
      JPTR,
      KOUNT,
      KPTR,
//      TEMP- A TEMPORARY ARRAY FOR SAVING TEXT WITH MACRO
//            CALL ARGUMENTS SUBSTITUTED FOR MACRO
//            DEFINITION ARGUMENTS
      TEMP[80],
      ;;
//    PROGRAM
      SUBARG: ?
//        INITIALIZE TEMPORARY VARIABLES
      0←KOUNT,
      0←IPTR,
      0←JPTR,
      0←I,
      0←TAG,
//      BLANK FILL THE TEMPORARY ARRAY 'TEMP'.
   ZERO:
      KOUNT=0110 $ ALOOP. ;
      BLANKBLANK←TEMP[KOUNT],
      KOUNT+01←KOUNT,
      ZERO.
   ALOOP:
//        EXIT IF ERROR FLAG IS SET
      EFLAG=01 $ EXIT SUBARG. ;
      01←TAG,
      I←JPTR,
      0←KOUNT,
//        EXTRACT THE DUMMY ARGUMENTS OF THE INPUT SOURCE
      EXTRACT,
      I←KPTR,
      RESULT[KOUNT]=0 $ FINAL. ;
//        CHECK FOR MACRO DEFINITION ARGUMENT
      RESULT[KOUNT]=DOLLAR $ START. ;
//        CHECK FOR SPECIAL CHARACTER
      SFLAG=0 $ INTZ1. ;
      ISC=CONCT $ SKIP1. ;
      RESULT[KOUNT]←TEMP[IPTR],
      IPTR+01←IPTR,
   SKIP1:
      ALOOP.
//        TRANSFER INPUT CHARCTER STRING FROM ARRAY SOURCE TO ARRAY
```

```
//        TEMP
  INTZ1:
     JPTR=KPTR $ INTZ2. ;
     SOURCE[JPTR]←TEMP[IPTR],
     JPTR+01←JPTR,
     IPTR+01←IPTR,
     BLANKBLANK←TEMP[IPTR],
     INTZ1.
  INTZ2:
     SOURCE[KPTR]=BLANKBLANK $ SKIP. ;
     ALOOP.
  SKIP:
     SOURCE[KPTR]←TEMP[IPTR],
     IPTR+01←IPTR,
     ALOOP.
  START:
//       COMPUTE THE POINTER TO MACRO CALL ARGUMENT IN STACK
     0←IK,
     0←JPTR,
     I-01←IK,
     SOURCE[IK]*05←JPTR,
     JPTR+02←JPTR,
     JPTR+STKPT←JPTR,
     0←KOUNT,
  BLOOP:
//         SUBSTITUTE THE ACTUAL ARGUMENT OF THE MACRO CALL SAVED IN
//         THE STACK
     S[JPTR]=0 $ CLOOP. ;
     KOUNT=05 $ CLOOP. ;
     S[JPTR]←TEMP[IPTR],
     IPTR+01←IPTR,
     JPTR+01←JPTR,
     KOUNT+01←KOUNT,
     BLOOP.
  CLOOP:
     0←KOUNT,
     SOURCE[I]=BLANKBLANK $ DLOOP. ;
     ALOOP.
  DLOOP:
     IPTR+01←IPTR,
     ALOOP.
  FINAL:
     0←IK,
     0←KOUNT,
//       SAVE THE ASSEMBLY LANGUAGE TEXT WITH MACRO CALL ARGUMENT
//       SUBSTITUTED FOR MACRO DEFINITION ARGUMENT IN ARRAY
//       'SOURCE'.
//         SOURCE.
  SETST:
     KOUNT=0110 $ EXIT SUBARG. ;
     BLANKBLANK←SOURCE[KOUNT],
     TEMP[KOUNT]←SOURCE[KOUNT],
     KOUNT+01←KOUNT,
     SETST.
  EXIT SUBARG:
     ;
     ..

     SUBROUTINE, ERROR
//*************************************************************
//*                                                           *
//*      PURPOSE OF THE MODULE                                *
//*                                                           *
```

```
//*      TO OUTPUT APPROPRIATE ERROR MESSAGE CODE                  *
//*                                                                *
//*      USAGE                                                     *
//*                                                                *
//*      ERROR,                                                    *
//*                                                                *
//*      MODULES REQUIRED                                          *
//*                                                                *
//*      PACK                                                      *
//*                                                                *
//***********************************************************************
//      EXTERNALS
    BLANKBLANK,
    BUFFER,
    LUINP,
    LUOUT,
    PACK,
    "P.IN",
    "P.OUT",
    "P.HLT",
    SOURCE,
    0,
    01,
    012,
    0120,
    ;;
//      EFLAG- FLAG TO INDICATE ERROR CONDITION
    *EFLAG,
//      TEMPORARY VARIABLE
    KOUNT,
//      MESSAGE- AN ARRAY SAVING CHARACTERS '* * E R R O R' AND
//                ERROR CODE
    *MESSAGE[10]=052,052,0105,0122,0122,0117,0122,
//      TEMP- A TEMPORARY ARRAY TO SAVE CONTENTS OF ARRAY SOURCE
    TEMP[80],
    ;;
    ERROR: ?
    0←KOUNT,
//      SAVE THE CONTENTS OF ARRAY SOURCE IN ARRAY TEMP.
 LOOP0:
    KOUNT=0120 $ LOOP1. ;
    BLANKBLANK←TEMP[KOUNT],
    SOURCE[KOUNT]←TEMP[KOUNT],
    BLANKBLANK←SOURCE[KOUNT],
    KOUNT+01←KOUNT,
    LOOP0.
 LOOP1:
    0←KOUNT,
//      TRANSFER ERROR MESSAGE IN ARRAY MESSAGE TO ARRAY SOURCE
 LOOP2:
    KOUNT=012 $ LOOP3. ;
    MESSAGE[KOUNT]←SOURCE[KOUNT],
    KOUNT+01←KOUNT,
    LOOP2.
 LOOP3:
    0←KOUNT,
//      PACK THE CONTENTS OF ARRAY SOURCE
    PACK,
//      OUTPUT THE ERROR MESSAGE
    <BUFFER>
//      TRASFER BACK THE CONTENTS OF ARRAY SOURCE
 LOOP4:
    KOUNT=0120 $ EXIT ERROR. ;
```

```
    BLANKBLANK←SOURCE[KOUNT],
    TEMP[KOUNT]←SOURCE[KOUNT],
    KOUNT+01←KOUNT,
    LOOP4.
  EXIT ERROR:
//      SET ERROR FLAG
    01←EFLAG,
    ;
    ..

    SUBROUTINE, PACK
//*******************************************************
//*                                                     *
//*     PURPOSE OF THE MODULE                           *
//*                                                     *
//*      TO PACK EIGHTY ASCII CHARACTERS INTO A FORTY WORD   *
//*      ARRAY 'BUFFER' WITH TWO ASCII CHARACTERS PER WORD.  *
//*                                                     *
//*     METHOD                                          *
//*                                                     *
//*     INTEGER DIVISION                                *
//*                                                     *
//*     USAGE                                           *
//*                                                     *
//*     PACK,                                           *
//*                                                     *
//*     MODULES REQUIRED                                *
//*                                                     *
//*     ERROR                                           *
//*                                                     *
//*******************************************************
//    EXTERNALS
    BLANKBLANK,
    BUFF LIM,
    EFLAG,
    ERROR,
    LUINP,
    LUOUT,
    MESSAGE,
    SHIFT,
    SOURCE,
    0,
    01,
    010,
    ;;
//    LOCALS
//       LENGTH OF THE ARRAY SOURCE.
    ARRAY LIM=0120,
//      'BUFFER'- ARRAY OF FORTY WORDS TO SAVE EIGHTY CHARACTERS
//                WITH TWO CHARACTERS PER WORD.
    *BUFFER[40],
//      TEMPORARY VARIABLES.
    J,
    K,
    ITEMP,
    ;;
//    PROGRAM
    PACK: ?
//      INITIALIZE TEMPORARY VARIABLES.
    0←J,
    0←K,
    0←ITEMP,
//      BLANK FILL THE ARRAY BUFFER .
```

```
   AINTZ:
      K=BUFF LIM $ BLOOP. ;
      BLANKBLANK+BUFFER[K],
      K+01+K,
      AINTZ.
   BLOOP:
      0+K,
   ALOOP:.
      K=ARRAY LIM $ EXIT PACK. ;
 //      SAVE THE UPPER BYTE IN ITEMP.
      SOURCE[K]*SHIFT+ITEMP,
      K+01+K,
 //      SAVE THE LOWER AND UPPER BYTE.
      SOURCE[K]+ITEMP+BUFFER[J],
      K+01+K,
      J+01+J,
      ALOOP.
      EXIT PACK:
      ,
      ..
LIST END ****
      SUBROUTINE, UNPAK

 //***************************************************************
 //*                                                             *
 //*      PURPOSE OF THE MODULE                                  *
 //*                                                             *
 //*      TO UNPACK THE CONTENTS OF THE ARRAY BUFFER OF FORTY
 //*      WORDS, WHERE EACH WORD IS PACKED WITH TWO ASCII        *
 //*      CHARACTERS.
 //*                                                             *
 //*    METHOD                                                   *
 //*                                                             *
 //*    INTEGER DIVISION                                         *
 //*                                                             *
 //*    USAGE                                                    *
 //*                                                             *
 //*    UNPAK,                                                   *
 //*                                                             *
 //*    MODULES REQUIRED                                         *
 //*                                                             *
 //*    ERROR                                                     *
 //*                                                             *
 //***************************************************************
 //     EXTERNALS

      BLANKBLANK,
      BUFFER,
      BUFF LIM,
      EFLAG,
      ERROR,
      LUINP,

      LUOUT,

      MESSAGE,
       0,
       01,
      010,
      0110,
      ;;

 //     LOCALS
```

```
//    TEMPORARY VARIABLES.
    ITEMP,
    J,
    K,
    *SHIFT=0400,
//    'SOURCE'- ARRAY TO SAVE THE UNPACKED CHARACTERS OF
//              BUFFER WITH ONE ASCII CHARACTER PER WORD.
    *SOURCE[80],
    ;;

//  PROGRAM

    UNPAK: ?

//    INITIALIZE TEMPORARY VARIABLES
    0←J,

    0←K,

    0←ITEMP,
//    BLANK FILL ARRAY SOURCE.
  AINTZ:

    K=0110 $ BLOOP. ;
    BLANKBLANK←SOURCE[K],
    K+01←K,

    AINTZ.

  BLOOP:

    0←K,

    0←J,

    0←ITEMP,
  ALOOP:

    K=BUFF LIM $ EXIT UNPAK. ;

//    SAVE UPPER BYTE OF THE WORD.
    BUFFER[K]/SHIFT←SOURCE[J],

    SOURCE[J]*SHIFT←ITEMP,
    J+01←J,

//    SAVE LOWER BYTE OF THE WORD.
    BUFFER[K]-ITEMP←SOURCE[J],
    J+01←J,

    K+01←K,

    ALOOP.

  EXIT UNPAK:
```

```
SUBROUTINE, SEARCH

//*************************************************************
//*                                                           *
//*  PURPOSE OF THE MODULE.                                   *
//*                                                           *
//*  TO SEARCH MACRO NAME TABLE.                              *
//*                                                           *
//*  METHOD                                                   *
//*                                                           *
//*  HASH TABLE SEARCHING METHOD, HASH CODE GENERATED USING   *
//*  DIVISION METHOD.                                         *
//*                                                           *
//*  USAGE                                                    *
//*                                                           *
//*  SEARCH,                                                  *
//*                                                           *
//*  MODULES REQUIRED                                         *
//*                                                           *
//*  ERROR                                                    *
//*  HASH                                                     *
//*                                                           *
//*************************************************************
//    EXTERNALS

      AFLAG,
      AMACNT,
      EFLAG,
      ERROR,
      HASH,
      HASHTB,
      IHASHPTR,
      LUINP,
      LUOUT,
      MDTP,
      MESSAGE,
      RESULT,
      O,
      O1,
      O2,
      O4,
      O5,
      O10,
      ;;

//    LOCALS

//      TEMPORARY VARIABLES
      ITEMP,

      JTEMP,

      KOUNT,
      ARRY LIM=0140,
      ;;

//    PROGRAM

      SEARCH: ?

//    INITIALIZE TEMPORARY AND LOOP CONTROL VARIABLES

      O1←AFLAG,
```

```
    O←ITEMP,
    O←JTEMP,
    O←KOUNT,

//        OBTAIN THE HASH CODE.
    HASH,
//        EXIT IF ERROR FLAC IS SET
    EFLAG=01 $ EXIT SEARCH. ;
//        CHECK WHETHER ENTRY TO BE SEARCHED IS IN THE MACRO NAME
//        TABLE.
    HASHTB[IHASHPTR]=0 $ EXIT SEARCH. ;
//     OBTAIN FROM HASH TABLE THE POINTER FOR THE ENTRY TO BE
//     SEARCHED IN THE MACRO NAME TABLE.
    HASHTB[IHASHPTR]←ITEMP,
 ALOOP:

    ITEMP=ARRY LIM $ EXIT SEARCH. ;
//     COMPARE FOR ENTRY TO BE SEARCHED IS IN THE MACRO NAME TABLE
    KOUNT=05 $ BLOOP. ;

    RESULT[KOUNT]=AMACNT[ITEMP] $ CLOOP. ;

//      MATCH NOT SUCESSFUL, TRY TO MATCH WITH THE NEXT ENTRY IN THE TABLE
    05-KOUNT←JTEMP,
   JTEMP+01←JTEMP,
    ITEMP+JTEMP←ITEMP,

    AMACNT[ITEMP]=0 $ EXIT SEARCH. ;
    O←KOUNT,

    ALOOP.

 CLOOP:

    KOUNT+01←KOUNT,

    ITEMP+01←ITEMP,

    ALOOP.

//    MATCH IS SUCESSFUL, RETURN THE FLAG VALUE FOR SUCESSFUL MATCH

 BLOOP:

    O←AFLAG,

//        RETRIVE THE MACRO DEFINITION TABLE POINTER .
    AMACNT[ITEMP]←MDTP,
 EXIT SEARCH:
    ,

    ..
```

```
     SUBROUTINE, IFTEST
//*************************************************************
//*                                                           *
//*        PURPOSE OF THE MODULE                              *
//*                                                           *
//*        TO EVALUATE THE PREDICATE OF AIF- PSUEDO-OP        *
//*                                                           *
//*        USAGE                                              *
//*                                                           *
//*        IFTEST,                                            *
//*                                                           *
//*        MODULES REQUIRED                                   *
//*                                                           *
//*        ERROR                                               *
//*                                                           *
//*************************************************************
//   EXTERNALS
     AIFFLAG,
     EFLAG,
     ERROR,
     OPERAND,
     MESSAGE,
     RFLAG,
     0,
     01,
     02,
     03,
     04,
     05,
     06,
     012,
     LUINP,
     LUOUT,
     ;;
//   LOCALS
//      TEMPORARY VARIABLES
     K,
     KOUNT,
     ;;
//   PROGRAM
     IFTEST: ?
//      INITIALIZE TEMPORARY VARIABLES
     0←KOUNT,
     0←K,
//      BRANCH TO APPROPRIATE LABEL DEPENDING UPON RELATIONAL
//      OPERATORS SPECIFIED BY THE FLAG VALUES
     RFLAG=0 $ CHEK EQUAL. ;
     RFLAG=01 $ CHEK NOT EQUAL. ;
     RFLAG=04 $ CHEK LESS. ;
     RFLAG=05 $ CHEK LESS. ;
     05←K,
//      COMPARE THE TWO OPERANDS FOR THE CONDITION GREATER,
//      GREATER THAN OR EQUAL
  LOOP1:
     K=012 $ EXIT1. ;
     OPERAND[K]<OPERAND[KOUNT] $ GREATER. ;
     OPERAND[K]=OPERAND[KOUNT] $ CONT1. ;
//      CONDITION NOT SATISFIED, RETURN FLAG VALUE IN 'AIFFLAG'
     0←AIFFLAG,
     RETURN.
  CONT1:
     K+01←K,
     KOUNT+01←KOUNT,
```

```
   LOOP1.
 GREATER:
//     CONDITION SATISFIED, RETURN FLAG VALUE 01 THROUGH
//     'AIFFLAG'
   01←AIFFLAG,
   RETURN.
 EXIT1:
   RFLAG=02  $ PASS2 . ;
   0←AIFFLAG,
   RETURN.
 PASS2:
   01←AIFFLAG,
   RETURN.
 CHEK LESS:
   0←K,
   05←KOUNT,
//     COMPARE THE TWO OPERANDS FOR THE CONDITION LESS,
//     LESS THAN OR EQUAL
 LOOP2:
   KOUNT=012 $ EXIT2. ;
   OPERAND[K]<OPERAND[KOUNT] $ LESS. ;
   OPERAND[K]=OPERAND[KOUNT] $ CONT2. ;
//     CONDITION NOT SATISFIED RETURN FLAG VALUE
   0←AIFFLAG,
   RETURN.
 CONT2:
   K+01←K,
   KOUNT+01←KOUNT,
   LOOP2.
 LESS:
//     CONDITION SATISFIED RETURN FLAG VALUE THROUGH AIFFLAG
   01←AIFFLAG,
   RETURN.
 EXIT2:
   RFLAG=05 $ PASS1. ;
   0←AIFFLAG,
   RETURN.
 PASS1:
   01←AIFFLAG,
   RETURN.
 CHEK EQUAL:
   0←K,
   05←KOUNT,
//     COMPARE TWO OPERANDS FOR EQUALITY
 LOOP3:
   KOUNT=012 $ EXIT3. ;
   OPERAND[K]=OPERAND[KOUNT] $ EQUAL. ;
//     CONDITION NOT SATISFIED RETURN FLAG VALUE
   0←AIFFLAG,
   RETURN.
 EQUAL:
   K+01←K,
   KOUNT+01←KOUNT,
   LOOP3.
 EXIT3:
//     CONDITION SATISFIED RETURN FLAG VALUE
   01←AIFFLAG,
   RETURN.
 CHEK NOT EQUAL:
   0←K,
   05←KOUNT,
//     COMPARE THE TWO OPERANDS FOR THE CONDITION NOT EQUAL
 LOOP4:
```

```
    KOUNT=O12 $ EXIT4. ;
    OPERAND[K]=OPERAND[KOUNT] $ CONT3. ;
//      CONDITION SATISFIED RETURN FLAG VALUE
    O1←AIFFLAG,
    RETURN.
 CONT3:
    K+O1←K,
    KOUNT+O1←KOUNT,
    LOOP4.
 EXIT4:
//      CONDITION NOT SATISFIED RETURN FLAG VALUE
    O←AIFFLAG,
 RETURN:
    ,
```

INPUT TO THE MACRO PROCESSOR

```
        MACRO
        CHECK &ARG1,&ARG2,&ARG3
        LDA &ARG1     COMPUTE UPPER-INDEX
        CMA,INA
        ADA &ARG2
        SSA           SKIP IF A GE 0
        JMP &ARG3     A IS NEGATIVE IF &ARG2 < &ARG1
        MEND
INDEX   DEC 0
LOWER   DEC 1
UPPER   DEC 100       INDEX OF LAST ELEMENT
BASE    DEF ARRAY     ADDRESS OF FIRST ELEMENT
ARRAY   BSS 100       ARRAY STORAGE IS RESERVED HERE
        CHECK INDEX,UPPER,ERROR
        CHECK LOWER,INDEX,ERROR
        ADA BASE      A REGISTER NOW CONTAINS ADDRESS OF ELEMENT
*                     BASE(INDEX)
        END
```

OUTPUT FROM THE MACRO PROCESSOR

```
INDEX DEC 0
LOWER DEC 1            INDEX OF FIRST ELEMENT
UPPER DEC 100          INDEX OF LAST ELEMENT
          ARRAY        ADDRESS OF FIRST ELEMENT
ARRAY BSS 100          ARRAY STORAGE IS RESERVED HERE
      LDA INDEX       COMPUTE  UPPER-INDEX
      CMA,INA
      ADA UPPER
      SSA               SKIP  IF  A  GE  0
      JMP ERROR         A  IS  NEGATIVE  IF UPPER <INDEX
      LDA LOWER       COMPUTE  UPPER-INDEX
      CMA,INA
      ADA INDEX
      SSA               SKIP  IF  A  GE  0
      JMP ERROR         A  IS  NEGATIVE  IF INDEX <LOWER
      ADA DASE         A REGISTER NOW CONTAINS ADDRESS OF ELEMENT
*                      BASE(INDEX)
      END
```

EXAMPLE NO. 1 SIMPLE MACRO EXPANSION

INPUT TO THE MACRO PROCESSOR

```
MACRO
L      &ARG1,&ARG2
LD&ARG1 &ARG2
MEND
MACRO
ST     &ARG1,&ARG2
ST&ARG1 &ARG2
MEND
L A,X
ST A,X
END
```

OUTPUT FROM THE MACRO PROCESSOR

```
LDA X
STA X
END X
```

EXAMPLE NO. 2 MACRO'S FOR IBM LOAD AND STORE INSTRUCTIONS

INPUT TO THE MACRO PROCESSOR

```
MACRO
LOAD &ARG1
LDA &ARG1
MEND
MACRO
STORE &ARG1
STA &ARG1
MEND
MACRO
ADD &ARG1,&ARG2
LOAD &ARG1
ADA &ARG2
STORE &ARG1
MEND
ADD X,Y
END
```

OUTPUT FROM THE MACRO PROCESSOR

```
LDA X
ADA Y
STA X
END X
```

EXAMPLE NO. 3 MACRO CALL WITHIN MACRO DEFINITION

INPUT TO THE MACRO PROCESSOR

```
MACRO
LOAD &REG,&ARG1
LD&REG &ARG1              LOAD THE REGISTER &REG
MEND
MACRO
STORE &REG,&ARG1
ST&REG &ARG1
MEND
MACRO
ADD &REG,&ARG1,&ARG2
LOAD &REG,&ARG1
AD&REG &ARG2
STORE &REG,&ARG1
MEND
ADD B,X,Y
END
```

OUTPUT FROM THE MACRO PROCESSOR

```
LDS X          LOAD  THE  REGISTER B
ADB Y
STB X
END
```

EXAMPLE NO. 4 MACRO CALL WITHIN MACRO DEFINITION WITH

CONCATENATION

INPUT TO THE MACRO PROCESSOR

```
      MACRO
      STORE &ARG1,&ARG2
      AIF (&ARG2 EQ 1) LAB1
      STA &ARG1
      AGO LAB2
LAB1  NOP
      DST &ARG1,I
LAB2  NOP
      MEND
      STORE A,1
      STORE A,2
      MACRO
      LOAD &ARG1,&ARG2
      AIF (&ARG2 EQ 1) LAB1
      LDA &ARG1
      AGO LAB2
LAB1  NOP
      DLD &ARG1,I
LAB2  NOP
      MEND
      LOAD A,2
      LOAD A,1
      END
```

OUTPUT FROM THE MACRO PROCESSOR

```
      DST A,I
LAB2   NOP
      STA A
      LDA A
      DLD A,I
LAB2   NOP
      END
```

EXAMPLE NO. 5 CONDITIONAL MACRO EXPANSION

INPUT TO THE MACRO PROCESSOR

```
      MACRO
      IF &ARG1 &ARG2 &ARG3 &ARG4
      LDA &ARG1
      CPA &ARG3
      RSS
      JMP &ARG4
      MEND
      MACRO
      ELSE &ARG3 &ARG4
      JMP &ARG4
&ARG3 NOP
      MEND
      MACRO
      IFEND &ARG4
&ARG4 NOP
      MEND
      IF A EQ B LAB1
      LDA AA
      ADA BB
      STA CC
*     ELSE PART
      ELSE LAB1 LAB2
      LDA =D4
      CPA B
      RSS
      JSB MACPR
*     ENDIF
      IFEND LAB2
      END
```

OUTPUT FROM THE MACRO PROCESSOR

```
      LDA A
      CPA B
      RSS
      JMP LAB1
      LDA AA
      ADA BB
      STA CC
*     ELSE PART
      JMP LAB2
LAB1  NOP
      LDA =D4
      CPA B
      RSS
      JSB MACPR
*     ENDIF
LAB2  NOP
      END
```

EXAMPLE NO. 6 SIMPLE IF-THEN-ELSE STRUCTURED PROGRAMMING CONSTRUCT.

INPUT TO THE MACRO PROCESSOR

```
        MACRO
        WHILE &ARG1 &ARG2 &ARG3 &ARG4 &ARG5
&ARG4   NOP
        LDA &ARG1
        CPA &ARG3
        JMP &ARG5
        MEND
        MACRO
        WHEND &ARG4 &ARG5
        JMP &ARG4
&ARG5   NOP
        MEND
        WHILE A EQ B LAB1 LAB2
        LDB XX
        CPB =D12
        JSB SAMP
        INA
        WHEND LAB1 LAB2
        END
```

OUTPUT FROM THE MACRO PROCESSOR

```
LAB1    NOP
        LDA A
        CPA B
        JMP LAB2
        LDB XX
        CPB =D12
        JSB SAMP
        INA
        JMP LAB1
LAB2    NOP
        END
```

EXAMPLE NO. 7 SIMPLE WHILE STRUCTURED PROGRAMMING CONSTRUCT

INPUT TO THE MACRO PROCESSOR

```
       MACRO
       CASE &ARG1
       LDA &ARG1
       MEND
       MACRO
       CASOF &ARG1 &ARG2 &ARG3 &ARG4
       AIF (&ARG4 EQ 0) .LAB1
&ARG2  NOP
       JMP CEND"&ARG4
.LAB1  NOP
       CPA =D"&ARG1
       RSS
       AIF (&ARG3 EQ 2) .LAB2
       JMP &ARG3
       AGO .LAB3
.LAB2  NOP
       JMP CEND"&ARG4
.LAB3  NOP
       MEND
       MACRO
       CAEND &ARG1
CEND"&ARG1 NOP
       MEND
       CASE N
       LDA XX
       CASOF 2 LAB1 LAB2 0
       LDB YY
       ADB =D100
       CASOF 4 LAB2 LAB3 1
       INB
       CPB =D40
       RSS
       JSB CHESS
       CASOF 8 LAB3 2 1
       INA
       CAEND 1
       END
```

OUTPUT FROM THE MACRO PROCESSOR

```
       LDA XX
       CPA =D2
       RSS
       JMP LAB2
       LDB YY
       ADB =D100
LAB2   NOP
       JMP  CEND1

       CPA =D4
       RSS
       JMP LAB3
       INB
       CPB =D40
       RSS
       JSB CHESS
LAB3   NOP
       JMP  CEND1

       CPA =D8
       RSS
       JMP  CEND1

       INA
CEND1  NOP
       END
```

EXAMPLE NO. 8 SIMPLE CASE STRUCTURED PROGRAMMING CONSTRUCT

APPENDIX - D

MACRO PROCESSOR USAGE

The Deck set up for using Macro Processor is similar to that of an HP Assembly Program. The only difference is the file name used in the PROG control card.

:PROG,MACPRO,p1,p2,p3,p4,99

Where

p1 = Logical unit of input device (5 is standard)

p2 = Logical unit of list device (8 is standard)

p3 = Logical unit of punch device

p4 = Lines per page on listing (56 is standard)

99 = Job binary parameter. If present, the object program is stored in the job binary area for later loading.

RESERVED WORDS:

The following are reserved words exclusively used by the Macro Processor Viz: MACRO,MEND,END,AIF,AGO

The major restriction Macro Processor is strictly no recursion is allowed, i.e. A Macro cannot call itself.

The output generated by the Macro Processor is processed by the HP Assembler. As HP Assembler has software to generate extensive error messages, only a limited number of error messages are generated by the Macro processor. The Macro Processor software is flexible enough for extension.

**ERROR 1 : Label specified in AGO psuedo-op is undefined.

**ERROR 2 : Macro Name Table is full.

**ERROR 3 : Macro Definition Table is full.

**ERROR 4 : Argument List Array is full.

**ERROR 5 : Stack over flow.

REFERENCES

[BRO 73] Browning, C.A., "Discussion and Correspondence Using Macros to aid Assembly Language Teaching". Comp. J. Vol. 16, No. 3, (Aug 1973), pp. 281-282.

[BRO 74] Brown, P.J., "Macro Processors", John Wiley & Sons, 1974.

[DON 72] Donovan, J.J., Systems Programming, McGraw Hill Koga Kusha, Ltd., International Student Edition (1972)

[FRA 75] Frailey, D.J., "Should High Level Language be Used to Write System Software?" Proc. ACM Conf. 75, Minneapolis, Minnesota.

[GRE 59] Greenwald, I.D., "A technique for Handling Macro Instructions", Comm. ACM, Vol. 2, No. 11, (Nov. 1959), pp. 21-22.

[HAL 74] Halstead, M.H., "A Laboratory Manual for Compiler and Operating System Implementation", American Elsevier Publishing Co., Inc., 1974.

[HER 75] Herman-Giddens, G.S., Warren, R.B., Barr, R.C., and Spach, M.S., "BIOMAC - Block Structured Programming Using PDP-11 Assembler Language", Software - Practice and Experience, Vol. 5, (1975), pp. 359-374.

[INF 76] Infotech State of the Art Report 1976, "Structured Programming", Infotech International Limited, Berkshire, England.

[KES 70] Kessler, M.M., "*CONCEPT* Report 14, Implementation of Macros to Permit Structured Programming in OS1360", IBM Corp., Gaithersburgh, Maryland 20760 (1970).

[KES 72] Kessler, M.M., "Assembly Language Structured Programming Macros", IBM, Gaithersburgh, Md., (1972).

[MAU 75] Maurer, W.D., and Lewis, T.G., "Hash Table Methods", Computing Surveys, Vol. 7, No. 1, (March 1975).

[MCG 75] McGowan, C.L., and Kelley, J.R., "Top-Down Structured Programming Techniques", Petrocelli Charter, N.Y. 1975.

[NEE 76] Neely, P.M., "The New Programming Dicipline", Software - Practice and Experience, Vol. 6, (1976), pp. 7-27.

[RIE 76] Rieks, G.E., "Structured Programming in Assembler Language", Datamation, (July 76), pp. 79-84.

[SOH 76] Sohrabji, N., "Macro Processor Simplifies Micro-computer", Computer Design, Vol. 15, No. 8, (Aug. 1976), pp. 108-112.

[WIR 68] Wirth, N., "PL360, A Programming Language for the 360 Computers", J. ACM, Vol. 15, No. 1, (Jan. 1968), pp. 37-74.

[WIR 74] Wirth, N., "On the Composition of Well Structured Programs", Computing Surveys, Vol. 6, (Dec. 1974), pp. 247-259.

[WUL 71] Wulf, W., Russel, D.B., and Haberman, A.N., "BLISS: A Language for Systems Programming", Comm. ACM, Vol. 14, No. 2, (Dec. 1971), pp. 780-790.