AN ON-LINE MACRO PROCESSOR FOR THE MOTOROLA
6800 MICROPROCESSOR

THESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Chang-Boe Hsieh, B. S.

Denton, Texas

May, 1980

The first chapter discusses the concept of macros: its definition, structure, usage, design goals, and the related prior work. This thesis principally concerns my work on OLMP (an On-Line Macro Processor for the Motorola 6800 Microprocessor), which is a macro processor which interacts with the user. It takes Motorola assembler source code and macro definitions as its input; after the appropriate editing and expansions, it outputs the expanded assembler source statements.

The functional objectives, the design for implementation of OLMP, the basic macro format, and the macro definition construction are specified in Chapter Two.

The software and the hardware environment of OLMP are discussed in the third chapter.

The six modules of OLMP are the main spine of the fourth chapter.

The comments on future improvement and how to link OLMP with the Motorola 6800 assembler are the major concern of the final chapter.
TABLE OF CONTENTS

LIST OF TABLES ..................... v
LIST OF ILLUSTRATIONS ................. vi

Chapter

I. INTRODUCTION ................. 1

   The Macro Definition
   The Macro Structure
   The Usage of Macros
   The Design Goals
   The Prior Work and Examples
   OLMP (On-Line Macro Processor)

II. RULES FOR MACRO DEFINITIONS AND CALLS OF OLMP ................. 21

   Introduction
   The Comment Record
   The Prototype Record
   The Model Record
   The Macro Trailer Record
   The Macro Definition within a Macro Definition
   The Concatenation of Formal Arguments

III. THE HARDWARE AND THE SOFTWARE ENVIRONMENT
     OF OLMP ................................ 38

   Introduction
   The Hardware Environment of OLMP
   The Software Environment of OLMP

IV. SIX MODULES OF OLMP ................. 54

   Introduction
   The Module of INPUTR
   The Module of COMPSR
   The Module of MACPRO
   The Module of PASONE
   The Module of PASTWO
   The Module of CORECT
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V. CONCLUSION.</td>
<td>91</td>
</tr>
<tr>
<td>The Future Improvement of OLMP</td>
<td></td>
</tr>
<tr>
<td>How to Connect OLMP with The Cross Assembler</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>100</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>104</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>122</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table                                             Page
1.1 Contrasts between OLMP and TRAC               18
3.1 Major Features of HP 2000                     42


LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Flow Chart of TRAC.</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Logical Blocks of a Source Program.</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Logical Block Structure of Macro Nesting Levels.</td>
<td>32</td>
</tr>
<tr>
<td>3.1</td>
<td>Standard HP 2000 Access System Configuration</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>HP 2000 Access Level.</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>File Structure of Disk Files.</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>Logical Blocks of OLMP.</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Flow Chart of INPUTR.</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Actual Organization of File &quot;SOURCE&quot;.</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Flow of Rearranging of Source String.</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Actual Organization of File &quot;COMPSD&quot;.</td>
<td>66</td>
</tr>
<tr>
<td>4.6</td>
<td>Flow Chart of MACPRO.</td>
<td>67</td>
</tr>
<tr>
<td>4.7</td>
<td>Flow Chart of PASONE.</td>
<td>69</td>
</tr>
<tr>
<td>4.8</td>
<td>Flow Chart for Setting up Macro Definition, Table, and Macro Name Table</td>
<td>71</td>
</tr>
<tr>
<td>4.9</td>
<td>Organization of File &quot;MADEFT&quot; and &quot;MANAMT&quot;.</td>
<td>74</td>
</tr>
<tr>
<td>4.10</td>
<td>Flow Chart of PASTWO.</td>
<td>78</td>
</tr>
<tr>
<td>4.11</td>
<td>Flow Chart of Subroutine for Checking Macro Name, Label, and Setting up Stack</td>
<td>79</td>
</tr>
<tr>
<td>4.12</td>
<td>Flow Chart of Subroutine 1430 for Setting up the Argument List Array</td>
<td>80</td>
</tr>
</tbody>
</table>
## LIST OF ILLUSTRATIONS (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>Flow Chart of the Macro Expansion Section</td>
<td>81</td>
</tr>
<tr>
<td>4.14</td>
<td>Flow Chart of CORECT.</td>
<td>88</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

The Macro Definition

The word "macro" is borrowed from the Greek word meaning "large". In the context of computer programming languages, this word means an extended programming facility, but there is no fixed definition for it. In order to lay our hands on its meaning we could start from its historical background, its usages, and its relationship with computer programming language to get to know it better. Its properties have been discussed by several authorities in this field.

"A "macro" is a stored string in which particular sites of the string are marked so that other strings can be inserted in the sites when the stored string is brought forth from storage," commented Mooers (14).

Mooers provides us a clear and broad idea about what a macro is. The concept of macro is concerned with the technique of character string manipulation, although the relationship between the macro and the computer programming language is still not very clear from that description.
From Brown's information (2), 'In its simplest form, a macro instruction (which is usually called simply a "macro") is a single computer instruction that stands for a given sequence of instructions..."; a single macro instruction can represent a series of pre-arranged computer instructions. That clarifies the "macro" as being a kind of computer instruction and the relation between the macro and the computer programming language. We can now say that "macro" has something to do with computer programming language.

D. W. Scott has made a further explanation about the nature of a Macro.

The word "MACRO" means "LARGE". This word is suggested by the fact that the definitions of possibly long sequences of detailed machine instructions are written by the user only once. Thereafter in the same source program the user need only refer to ("call") the previously defined sequence with a single short statement in order to create long ("macro") sequences of new source statements. These new source statements are substituted for the calling statement. The new sequences may become quite large when, during an expansion, one macro calls another (18).

Now, we can see that a macro can be constructed from multiple computer instructions, from other macro definitions, or from a combination of the two.

By defining the appropriate macro instructions, an assembly language programmer can tailor his own higher level facility in a convenient manner, at no cost in control over the structure of his program. He can achieve the conciseness and ease in coding of high level languages without losing the basic advantage of assembly programming. Integral macro operations simplify debugging and program modification and the facilitate standardization (3).

Donovan has narrowed down the macro facility for assem-
bly language only; but, as a matter of fact, the macro facility is used in the high level source programming language also, such as PL/I and ALGOL 68. (1)

From those foregoing paragraphs, we can conclude that a "macro" implies that we can insert, replace, or expand character string texts according to a previously defined procedure and a previously fixed format. These strings can define data structures and attributes, as well as program algorithms. The facility we use to define and to process macro definitions is called a macro processor. For the computer programming language translator itself, the macro processor is its preprocessor. The macro processor could look like a programming language, but why do we use the name macro processor rather than the name macro language? The major reason is that macro definitions are processed and expanded by their syntactic definition; there is no semantic implication, so the name macro processor, rather than macro language, is more appropriate.

The Macro Structure

The shape and the format of a macro definition may vary among different macros, but the main skeletons of their composition are quite consistent. Therefore we may divide the structure of a macro definition into three major components.
First of all, we must let the macro processor know where the starting point of a macro definition is. That means, the component will trigger the whole macro processing. Of course, we can imagine those triggers may have more than one form.

There is a convenient and popular way to show the starting point of a macro definition, by the use of the word "macro" (6, 9, 12) to declare the starting point of macro processing or by the use of any other functional equivalent identifiers.

The second component of a macro definition is the macro name (5, 10, 11) with or without formal arguments (parameters) (3) and the macro definition text itself. The macro name is given by the user; this furnishes a means to call forth the macro form later in the source statements by referring to its name. Different macro processors have their own sets of rules about the macro name constructions; but the major part of those rules provide for restrictions of the length and the format of a macro name.

A predefined symbol is used to separate arguments, and the last argument is followed by a symbol called terminator. Both of those symbols are called delimiters or break characters.

The macro definition text is followed by the macro name; all the information processing (the macro text
expansion) happens in this section. This processing affects three separate groups of information.

The first group is the text into which new text will be inserted: the base language or the base instruction. The base language could be a low-level language, such as IBM assembly language, or a high-level programming language, such as PL/I (6, 7).

The second group of information is the formal arguments (parameters) of the definition. Arguments are replaced by assigned or by default values when the macro is called.

The third type of information in a macro definition is the local facilities, which can control or modify dynamically the macro expansion process. The complexity of the macro definition expansion facilities depends upon the design goals for the macro processor. Generally speaking, macro variables are adopted by most macro processors. It is desirable to have both local macro variables and global macro variables.

Local variables are used for internal working within the evaluation of an individual macro call; global variables are needed to relay information from one macro call to another macro call. If recursion is permitted, local variables are almost obligatory.

Conditional generation is one of the macro definition expansion time facilities. With mathematical or logical
relationships among those local variables or global variables, a macro processor can control a macro definition expansion routes.

Macro library facilities can serve to augment a macro processor. A macro library serves as a general purpose macro definition pool, permitting the user to insert a block of macro statements automatically by calling the macro, without defining it in the first place.

The final component of the macro definition structure signifies its end.

The above paragraph gives a general overview about the macro structure processing capability, rather than trying to use a template to get every kind of macro squeezed into it. As a matter of fact, there is an unusual macro processor constructed somewhat differently from that discussed in the foregoing paragraph. This macro processor is called TRAC (Text Reckoning And Compiling), and it will be introduced later.

The Usage of Macros

The macro facility could be used as the programming language extension, just like the macro facility in various assembly languages. It could be used in a programming language translation, such as translating a high level programming language to a low level assembly language, and
it could be used in the text generation and editing process.

The Design Goals

The design goals of a macro processor are briefly described in the following paragraphs.

A macro processor can be used to provide an optional inclusion within a macro definition; so there could be more than one logical route during the macro definition expansion. The determination of the logical path for expansion is controlled by a given condition along with the macro call.

A macro processor cannot hope to match the power of a special-purpose report generator. However, it can generate limited reports or messages. The user can use this to do some documentation along with the macro definition; or he can set up conditional output messages within the macro definitions.

For ease of formulating and correcting it, as well as to reduce (possibly) the amount of code generated by the macro processor, a relatively big definition should be divided into several smaller logical units. That means a macro processor should handle the macro calls within a macro call to a certain level of nesting. Furthermore, a macro processor may have the recursive feature both during definition and expansion processing.
In order to communicate with the outside world and the internal chain reaction, a macro definition sometimes must be equipped with one or more communication areas. These are called "arguments" or "parameters".

The editing of the input macro definition is one of the basic functions performed by a macro processor. This function is utilized to be sure the macro definition is following the predefined rules.

The Prior Work and Examples

As an example, consider this IBM SYSTEM/360 macro definition (9):

```
MACRO
SWING &DATA
LCLC &PRE,&A
&PRE SETC 'A'
&A SETC 'A'
L 5,&DATA&PRE
ST 5,&DATA&A
MEND
```
machine dependent if it is a low level programming language. "MEND" is the end of this macro definition. If we have the macro call "SWING GANG", then the result of the macro expansion is:

\[
\begin{align*}
&L 5, GAN A \\
&ST 5, GAN GB
\end{align*}
\]

Let us take this high-level programming language PL/I macro definition as another example (7):

```pli
%DECLARE I FIXED; %I = 1;
%LAB: Z(I) = X(I) + Y(I);
%I = I + 1;
%IF I <= 100
  %THEN %GO TO LAB;
%DEACTIVATE I;
```

"%" in PL/I symbolizes the beginning of a macro definition, the "I" is the macro variable and "%DEACTIVATE I;" is the end of the macro definition. In this example, we can see that the equation \( Z(I) = X(I) + Y(I) \) will be generated one hundred times, with \( I = "1", "2", "3", \ldots "100" \), as controlled by the conditional generation facility statement, "%IF I <= 100".

Now, here is the unusual general-purpose macro generator, "TRAC" (1, 4, 15, 16, 18):

TRAC was created by Calvin Mooers, who started its development started in 1960. TRAC is a general-purpose macro generator designed specially for the processing of unstructured texts in an interactive mode (on line). In addition to character strings, TRAC can handle recursive
procedures, integers, and Boolean variables.

The design goals of TRAC included high capability of dealing with back-and-forth communication between a man at a keyboard and his work on the machine, maximum versatility in the definition and performance of any well-defined procedure on text, the ability to define, store, and subsequently use a procedure, and its straightforward format so that it could be easily adopted by the user.

The basic operations of TRAC are dominated by two functions: the definition of a macro forms and the expansion of the forms. Expansion of forms involves two distinct concepts, that of an active function and that of a neutral function. These operations are denoted by "#(...)
 and "##(...)", respectively. (The function list of TRAC is given in APPENDIX A). A primitive instruction format for the active function is

```
#(FUNa, K1, K2, ..., Km)
```

After substitution of actual text parameter values for formal arguments, the text of active function expansion is then rescanned to cause further evaluations (if such are present in expanded text). This means that the function is denoted (named) by FUNa and uses the parameters K1, K2, ..., Km. A primitive instruction format for the neutral function is

```
##(FUNn, K1, K2, ..., Km)
```
DELETE ")", SET A POINTER TO THIS END OF ARGU-
MENT SUBSTRING AND THE END OF THE FUNCTION.
GET THE POINTER OF THE BEGINNING OF THE 
CURRENT FUNCTION. NOW THE COMPLETE SET OF 
ARGUMENT SUBSTRING FOR THE FUNCTION HAS 
BEEN DEFINED.

Y

THE RESULT OF THE FUNCTION IS INSERTED TO THE 
LEFT OF THE UNSCANNED STRING. AND THE POINT-
ER RESET BACK TO THE 
FIRST CHARACTER OF THIS RESULT.

N

IT MUST BE THE NEUTRAL 
FUNCTION. THE VALUE 
OF THIS SUBSTRING IS 
INSERTED INTO THE LEFT 
SIDE OF THE UNSCANNED STRING.

FIGURE 1.1 Flow Chart of TRAC. 
(CONTINUED)
SUBROUTINE GET NEXT CHAR.

CHAR. IS "(*)? Y

DELETE ", SET A POINTER POINTS TO THIS ACTIVE FUNCTION STRING.

SUBROUTINE GET NEXT CHAR.

CHAR. IS "+"? N

SUBROUTINE GET NEXT CHAR.

CHAR. IS "(*)? Y

DELETE "#(", SET A POINTER TO THIS NEUTRAL FUNCTION STRING.

SUBROUTINE GET NEXT CHAR.

3

(CONTINUED)

FIGURE 1.1 Flow Chart of TRAC.

(CONTINUED)
SUBROUTINE GET NEXT CHAR.

ANY CHAR. LEFT TO BE SCANNED?

CHANGE IT TO IDLING PROCEDURE

"("  ?

CR, LF, OR TAB?

DELETE THE "(" AND FIND THE FIRST ")" DELETE IT, PUT THE REST OF STRING INTO THE NERIAL STRING.

DELETE CR, LF, OR TAB.

"."  ?

DELETE "." SET A POINTER POINTS TO ITS RIGHT SIDE ARGUMENT SUBSTRING.

")"  ?

(continued)

FIGURE 1.1 Flow Chart of TRAC.
After substitution of actual text parameter values for formal arguments, the text of neutral function expansion is not subject to further evaluation. This implies that the immediate text is produced by FUNn using the parameters K1, K2, ..., Km, but there is no further evaluation of the immediate result.

From the flow chart of TRAC (figure 1.1), the design technique is achieved through the use of the special characters "#", "("", ")", and ",". The process consists of finding the next delimeter, performing the predefined function, and then going back to repeat the same procedure until there are no predefined functions left. Because of the way its processing is performed the internal data form must be stored in the form of linear strings or binary trees.

Mooers' philosophy seems very simple and easy to understand; however, the result of the interaction among those predefined functions could be very complex, yet we can use it to solve relatively sophisticated text processing problems. The following example will give us some impression about TRAC:

1. #(DS,TEXT,THIS IS TRAC)'
2. #(DS,PROGRAM,(#(PS,(**))#(PS,(4.)#(CL,#(RS)))#(CL,PROGRAM)))'
3. #(CL,PROGRAM)'
4. **TEXT'
5. THIS IS TRAC
6. **
Line one defines the text "THIS IS TRAC", names it "TEXT", and stores it in the TRAC processor memory. Lines two through four define a retrieving program named "PROGRAM". Finally, line five of this program is typed in. It calls PROGRAM, so the program prints out "**" at the beginning of an output line, and then waits for an input string (*(RS)). After the user types in "TEXT", the predefined string is called and printed (line seven). After this the program leaves *(CL,PROGRAM) in line four only; however, through this program call, TRAC rebuilds the "PROGRAM" and repeats the whole process if the user calls "TEXT" again. The "**" in line eight indicates TRAC's interactive and recursive feature.

Because of TRAC's define-recall, storing, and self-referencing features, it needs considerable secondary storage.

OLMP (On-Line Macro Processor)

The macro processor OLMP (see APPENDIX B) which I have developed is designed for the MOTOROLA 6800 microprocessor assembly language. The macro definitions and the assembler source statements are the input text to OLMP. After interaction with the user during OLMP processing, the expanded assembler source statements are the output of OLMP. This design is based upon the assumption of a small computer.
system, with its relatively limited storage and the negative impact of time consuming input/output activities. If we had a larger computer to generate the executable machine code for MOTOROLA 6800 micro processor, then the weak points of this micro processor could be compensated considerably. In the same time, we can take advantage of the multi-user environment and the better file management (better file generation, file protection, file updating, and etc).

The two-pass method was chosen to develop OLMP, the Hewlett-Packard 2000 Access System was chosen to serve as the preprocessor for the MOTOROLA 6800, and the Hewlett-Packard 2000 Access Basic was chosen to construct the OLMP (8). By the use of HP/2000 system resource (hardware and software), the OLMP can be called by as many as 32 users at the same time and the HP/2000 offers file creation, update, and protection features.

We can establish a macro definition library without spending one single memory unit of MOTOROLA 6800 to store it, and we can manipulate this library without consuming one single MPU (Micro Processing Unit) time of MOTOROLA 6800 to perform it.

It was a challenge to use HP/2000 Access Basic to construct a text processor (such as OLMP); not only are relatively advanced character string manipulation functions excluded from its instruction repertoire, but also the
space allocation for working files of OLMP is another problem which I needed to pay attention to.

The input macro definition is scanned by the entire macro definition, rather than by individual statement itself. There are six major program modules of this OLMP, five of them used for macro definition processing, and one of them used for source code manipulation. Details of six major modules are described in Chapter IV.
From the following table, the contrasts between OLMP and TRAC, both "macro processors", are obvious:

<table>
<thead>
<tr>
<th></th>
<th>OLMP</th>
<th>TRAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>design purpose</td>
<td>mainly for MOTOROLA 6800 assembly language</td>
<td>general text processing</td>
</tr>
<tr>
<td>functional capability</td>
<td>expand preformatted macro definitions only</td>
<td>almost any kind of text manipulation</td>
</tr>
<tr>
<td>interactive period</td>
<td>when there are errors during macro expansion</td>
<td>through the entire macro processing</td>
</tr>
<tr>
<td>input</td>
<td>MOTOROLA 6800 assembler source codes &amp; macro definitions</td>
<td>any character string</td>
</tr>
<tr>
<td>output</td>
<td>expanded MOTOROLA 6800 assembler source statements</td>
<td>any character string</td>
</tr>
<tr>
<td>internal data organization</td>
<td>sequential, indexed sequential</td>
<td>sequential, binary tree, linked lists</td>
</tr>
<tr>
<td>hardware environment</td>
<td>Hewlett Packard System 2000</td>
<td>PDP-11, 5, 8, 8S, 9 GE DATANET-30 IBM 360/67 EUROPEAN COMPUTERS SAAB D-21, ICT-1202</td>
</tr>
</tbody>
</table>

TABLE 1.1 Contrasts between OLMP and TRAC.
CHAPTER BIBLIOGRAPHY


CHAPTER II

RULES FOR MACRO DEFINITIONS AND CALLS OF OLMP

Introduction

The information about OLMP that is necessary to produce macro definitions and calls is given in this chapter.

The user's input is called the OLMP source program (figure 2.1). The output of OLMP is a list of instructions to be translated by the usual (non-macro) M6800 assembler. The result of the normal assembler translation is an object program for the MPU (Micro Processing Unit) to execute during system operation. Mnemonic (source symbolic) instructions used by the programmer to write his program must be translated into the object machine codes which are executable by the MPU. However, if a cross assembler (the assembler in the host computer which is used to generate executable machine codes for another computer) is to be used to perform the translation, the OLMP should be called first before any assembler processing.

The source program is written in an assembler language consisting of the seventy-two instructions of the M6800 and
the assembly directives defined by MOTOROLA (4). Assembly directives are useful in generating, controlling, and documenting the source program. With exceptions of FCB (Form Constant Byte), FCC (Form Constant Characters), and FDB (Form Double Constant Byte), they

```
Control Commands

MACRO
MEND
MACRO

Macro Definitions

MEND
MEND
MEND
MEND
MEND

Assembler Source Statements
```

FIGURE 2.1 The Logical Blocks of a Source Program.
do not generate object codes. Characters which are recognized by the assembler include

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters</td>
<td>A, B, C, ..., and Z</td>
</tr>
<tr>
<td>Integers</td>
<td>0 through 9</td>
</tr>
<tr>
<td># (pound sign)</td>
<td>specifies the immediate mode of machine instruction addressing.</td>
</tr>
<tr>
<td>$ (dollar sign)</td>
<td>specifies a hexadecimal representation.</td>
</tr>
<tr>
<td>@ (commercial at)</td>
<td>specifies an octal representation.</td>
</tr>
<tr>
<td>% (percent)</td>
<td>specifies a binary representation.</td>
</tr>
<tr>
<td>' (aspostrophe)</td>
<td>specifies a single literal character, to be translated to ASCII.</td>
</tr>
<tr>
<td>SPACE</td>
<td>blank</td>
</tr>
<tr>
<td>Horizontal TAB</td>
<td>carriage return</td>
</tr>
<tr>
<td>CR</td>
<td>line feed</td>
</tr>
<tr>
<td>LF</td>
<td>comma</td>
</tr>
</tbody>
</table>

The source character set is a subset of the industry standard, ASCII (American Standard Code for Information Interchange), and includes the ASCII characters, 20 (SP) through 5F (←). The character set and instructions used in the OLMP
have the same definition as those used in the Motorola Assembly Language.

Two special characters, the question mark (?) and the quotation mark ("), have to be mentioned here. The question mark is given by the system of HP/2000 to notify the OLMP user that the HP/2000 is ready to accept input data. A quotation mark is provided by the user to mark the beginning and the ending of an input character string. Leading blanks are ignored by HP/2000 unless an opening quote is included. Trailing blanks are ignored by HP/2000 unless a closing quote is included (3). The maximum length of any record is 72 characters.

There are five different record types used in this macro definition.

The Comment Record

Comment records are identified by having an asterisk "**" as the first character of the input string. The macro processor just passes through these statements; everything following the asterisk is converted into a comment for the output of OLMP, and does not affect the machine codes generated later by the assembler.

Example:

1. ??** THIS IS THE COMMENT
2. ??"* ERROR MESSAGE "
3. ?"* GO TO SUBROUTINE.

All of the above examples are acceptable with or without quotation marks (HP/2000 suppresses the leading and/or padding blanks, if the quotation marks are absent).

The Macro Record

The record which contains simply "MACRO". If the word, "MACRO", is found in a piece of text which is preceded and followed by blanks, then it is legitimate also. See example 5. This signals the OLMP processor that the following input record is the beginning of a macro definition (1, 2, 3). The word "MACRO" must be preceded and followed by one or more blanks.

Example:
4. ?" MACRO "
5. ?"BEGINNING OF THE MACRO CALL"
6. ?" MACRO "
7. ?" MACRO"
8. ?" MACRO"
9. ? MACRO"
10. ?"MACRO "
11. ?MACRO

Examples four through six are acceptable and they are treated as the beginning of a macro definition, but examples seven through eleven have various syntactic errors and are
not treated likewise by the macro processor (they are ignored by the OLMP).

The Prototype Record

A prototype record gives the mnemonic operation code and the formal arguments of all macro instructions that are part of the macro definition. When the user makes a macro call, he states the macro prototype name and gives actual parameter values in his source program. The format of the prototype record is (examples are given after following subsections):

The Label Field

This optional field must begin from the first character of the input prototype record and be proceeded by the character "&"; no more than seven characters are allowed in this field.

The Macro Name Field

This mandatory field must start at least one blank from the label field or after the first column of the input prototype record; no more than seven characters are allowed in the field.

The Formal Arguments Field
At least one blank should precede this optional field; no more than seven characters are allowed. No blank or blanks are allowed among the formal arguments. The first character of the formal argument must be a "&". The formal arguments are separated by ","; no more than 36 arguments are allowed in one macro definition.

The Comment Field

This optional field must be separated by at least one blank from the argument field. It is not allowed if there is no argument.

The following examples have some chosen error conditions and show the messages which are given by the macro processor.

12. (a) &1223456
   &LABEL
   &A
   &A123
   &001ABC

(b) &NOCODES
   &FROMHERE
   &COMBINE TOGETHER
   &12345678
   &ABCDEFG123

(c) ABC
   %ABC
   L123

Example 12 (a) is accepted by the macro processor. In example 12 (b) the system gives the warning message "+++ WARNING -- LABEL TOO LONG, ONLY 7 CHAR. ARE USED!", but the
processing continues. In (c) the system gives the error message "*** ERROR -- WRONG LABEL FORMAT!", and the error flag is turned on. The error flag is used internally for the OLMP to verify the error conditions then to take the appropriate action (details are discussed in the Chapter IV).

13. (a) CHANT
    TAPETST
    INITIAL

    (b) CHANGEOE
    GETSUBROUTINE
    TOTHEMOVIES

Example 13 (a) is legal to the macro processor for the name field, but in example 13 (b), the macro processor gives the warning message "+++ WARNING -- MACRO NAME TOO LONG, ONLY 7 CHAR. ARE USED!". Only the leading seven characters are used in the system, but the processing continues. This means that in the symbol table example 13 (b) looks like "CHANGBO", "GETSUBR", and " TOTEMO".

14. (a) &ABC, &12144, &S007
    &ABCDEF, &WE
    &ABC

    (b) &ABC, &1234, &Y007,
    &ABCDEFG, &WE, &A
    &ABC, &DEF, &B09
    &ABC, &DEF, &B789
    &ABC, &WE

Example 14 (a) illustrates legal arguments; however, example 14 (b) is not an acceptable to the macro processor. For the third example in group (b), the system gives two messages "+++ WARNING -- ARGUMENT TOO LONG, ONLY 7 CHAR. ARE
USED!" and "*** ERROR -- WRONG ARGUMENT FORMAT!"; but for the rest of the examples in group (b) only the error messages are given.

If the input of the prototype record is "ABC", then the system gives the message "*** ERROR -- MISSING MACRO NAME AND ARGUMENT FIELDS!" and the error flag goes on.

If the number of the dummy arguments is more than 36, then the system gives the error message "*** ERROR -- TOO MANY ARGUMENTS" and the error flag is turned on.

The Model Record

The skeleton of a macro definition is the model statement (4). The user can expect predesigned sequences of assembler language statements to be generated from these statements.

The model record could be a assembler instruction or another predefined macro definition. If it is an assembler instruction, then it must follow the rules of the M6800 assembly language, and it can be divided into the label field, the operation code field, the operand field, and the comment field.

The Macro Trailer Record

The macro trailer record identifies the end of a macro definition. It should have at least one blank preceding and
following it.

Example:

15. ?" MEND "
   ?" MEND "
   ?"THIS IS THE MEND "

16. ?" MEND
   ?" MEND"
   ?"MEND"
   ?MEND "
   ?MEND

The first example is acceptable to the macro processor, but the second example is ignored by the macro processor.

Example 17:

1. ?"**STATUS CHECK SUBROUTINE IN MACRO FORM***"
2. ?"
3. ?"** THIS MACRO DEFINITION CHECKS THE CURRENT STATUS
5. ?"** THE STATUS IS COMPARED WITH THE EXPECTED
6. ?"**GOOD STATUS (AVAIL., CAS. IN PLACE, RDY.,
7. ?"** IN SYNC, AND CRC ERR.) AND THE RESULT
8. ?"** OF THE COMPARISON IS SAVED IN THE TAPE
9. ?"** STATUS BUFFER TVSTAT.
10. ?"
11. ?" CHECK TAPE STATUS MACRO DEFINITION
12. ?" MACRO "
13. ?"&TKRDST CHECKT &XP5DRA,&TVSTATT
14. ?" LDA A &XP5DRA .AND ACCUMULATOR A.
15. ?" AND A #%11011110 .AND ACCUMULATOR A.
16. ?" EOR A #%00011100
17. ?" STA A &RVSTAT .STORE ACCUMULATOR A.
18. ?" RTS
19. ?" MEND "

Statements one through eleven are comment records, statement twelve to nineteen are macro definitions. Statement twelve is a macro record, statement thirteen is a prototype record, statements fourteen through eighteen are model records, and statement nineteen is a macro trailer.
The Macro Definition within A Macro Definition

The user may call a macro by its proto name within another macro definition. This feature gives the user a nesting capability for macro definitions. In the implementation of this macro processor, the macro call within another macro definition is limited to nine nesting levels (figure 2.2).

Figure 2.2 is the logical block of a macro call within another macro definition. One thing the user must always bear in mind is that the called macro must be defined before the calling macro. In the figure 2.2, the first macro definition can not call the third macro by its prototype name. But in the third macro definition, both the first macro and the second macro can be called.

Example 18:

```assembly
?"* THIS IS THE FIRST MACRO DEFINITION.
?" MACRO 
?"&LAB1 CLEAR &REGA,&REGB 
?" LDA A &REGA
?" LDA B &REGB
?" MEND 
?"*************
?"*************
?"*************
*************
*************

?"* THIS IS THE (N)TH MACRO DEFINITION.
?" MACRO 
?"&LABN ADDER &MONE,&MTWO
?" CLEAR ZERT,&MTWO
?" ADD A #$11110000
?" ADD B &MTWO
?" STA A &MONE
```
first level | first macro definition

first level | second macro definition

second level | second - may call first macro by its prototype

level | name

first level | third macro definition

second level | second - may call first and/or second macro

level | third - this is the first macro and it

level | it called by second macro

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |

level |
After the macro expansion of above example, it looks like:

```

CLEAR ZERO,ZERO
+++ LDA A ZERO
+++ LDA B ZERO

ADDER MEN1,MEN2
+++ CLEAR ZERT,MEN2
+++ LDA A ZERT
+++ LDA B MEN2
+++ ADD A #$1111000
+++ ADD B MEN2
+++ STA A MEN1
+++ STA B MEN2
```
In the above example after the first macro call (CLEAR ZERO,ZERO), ZERO gets into the Accumulator A and the Accumulator B. After the second macro call (ADDER MEN1,MEN2), the first macro definition is expanded because of the ADDER calls CLEAR within its definition. Finally, the binary number (11110001) is stored in MEN1 and MEN2 doubles its original value.

If the nesting level of a macro call exceeds nine levels, then the HP/2000 gives "SUBSCRIPT OUT OF BOUNDS" error message, and the execution of the program is terminated abnormally. Before the user starts to make a macro call (or several macro calls) within another macro definition, it is better to be very careful about the bookkeeping of the nesting level.

The Concatenation of Formal Arguments

The user may use the concatenate technique to change the operand field in the model statements. This optional feature gives the user a lot of flexibility to define his macro formal arguments. The macro processor allows only backward concatenation.

Example 19:

```
?" MACRO "
?"&LAB CONCAT &AA,&VV,&CC
?" LDA A CH&AA
```
After the macro expansion.

```
CONCAT C7,0000,DEX
+++ LDA A CHC7
+++ LDA B #%11110000
+++ LDX DEX

CONCAT C9,0010,DOR
+++ LDA A CHC9
```
+++ LDA B  #11110010
+++ LDX  DOR

.............
.............
.............

CONCAT C0,0111,DSL
+++ LDA A  CHC0
+++ LDA B  #11110111
+++ LDX  DSL

.............
.............
.............

END

In the first CONCAT call, argument "C7" takes the place of "&AA" of the instruction loaded in the Accumulator A, argument 0000 takes the place of "&VV" of the instruction loaded in the Accumulator B, and the argument "DEX" takes the place of "&CC" of the instruction loaded in the Index Register. In the second and the third CONCAT call, they all follow the same rules. The final arguments which are generated by the concatenation must meet the rules of Motorola Assembler Language.
CHAPTER BIBLIOGRAPHY


CHAPTER III

THE HARDWARE AND SOFTWARE
ENVIRONMENT OF OLM P

Introduction

Although the addressable memory of the Motorola 6800 microprocessor is up to sixty-five thousand five hundred and thirty-six words (8 bit words), the limitation of the memory and the input-output devices that are usually available do not allow us to process a relatively large program nor a more complicated job.

The microprocessor cannot satisfy the user's request for the relatively advanced programming environment so far. It is time for us to do something to soothe the user's itching. By the use of HP/2000 as a preprocessor of Motorola 6800 microprocessor, we might narrow down the gap between the user's request and the present status of this microprocessor. On the other hand, the use of macros can improve the control structure of assembler language programming and make the structured programming in assembler language programming a lot easier (1, 5).

Under the same subject, we may show people how to use a
relatively unsophisticated programming language like HP/2000 BASIC ACCESS to develop another programming facility and in the mean time, we are going to experience the general performance of the HP/2000 and its software package. This macro processor (OLMP) is an example to show the above ideas.

The Hardware Environment of OLMP

The HP 2000 Model Access System serves as the preprocessor of the Motorola 6800 microprocessor. The typical hardware configuration of the HP 2000 Model 40 Access System (2) is organized with an HP 2108A M/20 system processor, an HP 2112A M/30 communication processor, a 12972A digital magnetic tape subsystem, a 12925A high-speed punched tape reader subsystem (this has not been furnished at N.T.S.U.), a 12962A cartridge disk subsystem, a system console terminal printer (could be any model), and two 12920B asynchronous multiplexers (figure 3.1).

This system is intended for serving as many as 32 users concurrently with multiple sets of peripherals for local and remote job entry use. The major features of this system are introduced in Table 3.1.
FIGURE 3.1 The Standard HP 2000 Access System Configuration
<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>**</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A000</td>
<td>A001</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>A099</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>A100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>A199</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>A200</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>A299</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Z800</td>
<td>Z801</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Z899</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Z900</td>
<td>Z901</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Z999</td>
<td></td>
</tr>
</tbody>
</table>

* 1 system library
+ 260 group library
** 26,000 private library

FIGURE 3.2 The lower layer of idcodes represents private libraries, the middle column of idcodes represents the group libraries, and the first column represents the system library. (The system library is also the group library for the first group and the private library for account A000, the system master's account.)
<table>
<thead>
<tr>
<th>UNIT NAME</th>
<th>MAJOR FEATURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2108A M20 SYSTEM PROCESSOR</td>
<td>. 64 Kbytes of semiconductor memory</td>
</tr>
<tr>
<td></td>
<td>. floating point firmware</td>
</tr>
<tr>
<td></td>
<td>. dual-channel port controller</td>
</tr>
<tr>
<td></td>
<td>. power fail recovery</td>
</tr>
<tr>
<td></td>
<td>. time base generator</td>
</tr>
<tr>
<td></td>
<td>. disk loader ROM</td>
</tr>
<tr>
<td>2112A M/30 COMMUNICATION</td>
<td>. 64 Kbytes of semiconductor memory</td>
</tr>
<tr>
<td>PROCESSOR</td>
<td>. dual channel port controller</td>
</tr>
<tr>
<td></td>
<td>. power fail recovery</td>
</tr>
<tr>
<td></td>
<td>. time base generator</td>
</tr>
<tr>
<td>12972A DIGITAL MAGNETIC</td>
<td>. fast data transfer to 72 Kbytes/sec</td>
</tr>
<tr>
<td>TAPE</td>
<td>. single-track error correction</td>
</tr>
<tr>
<td></td>
<td>. 1600 cpi phase-encoded data electronic</td>
</tr>
<tr>
<td></td>
<td>. tape speed to 45 ips read/write, 160 ips rewind</td>
</tr>
<tr>
<td></td>
<td>. dynamic braking</td>
</tr>
<tr>
<td></td>
<td>. 26.7 com reels (10-1/2 inch)</td>
</tr>
<tr>
<td></td>
<td>. IBM/ANSI compatible</td>
</tr>
<tr>
<td>12925A HIGH-SPEED PUNCHED</td>
<td>. 500 cps read (415 cps 50 hz power)</td>
</tr>
<tr>
<td>TAPE READER</td>
<td>. 2.5 cm (1 inch) wide tape</td>
</tr>
<tr>
<td></td>
<td>. 8 level code</td>
</tr>
<tr>
<td>12962A CARTRIDGE DISK</td>
<td>. 15 megabytes of usable storage in 39.9 (15.7 inches) of rack space</td>
</tr>
<tr>
<td></td>
<td>. 10 megabytes removable front loading cartridge</td>
</tr>
<tr>
<td></td>
<td>. microprocessor based storage control unit</td>
</tr>
<tr>
<td></td>
<td>. average track to track 5 ms</td>
</tr>
<tr>
<td></td>
<td>. average random 25 ms</td>
</tr>
<tr>
<td></td>
<td>. 3600 rpm spindle speed</td>
</tr>
<tr>
<td></td>
<td>. average rotation delay 8.3 ms</td>
</tr>
<tr>
<td></td>
<td>. transfer rate 937.5 Kbytes/sec</td>
</tr>
<tr>
<td>12920B ASYNCHRONOUS MULTIPLEXER</td>
<td>. 16 asynchronous communications channels</td>
</tr>
<tr>
<td></td>
<td>. data rate from 57 baud through 2400 baud</td>
</tr>
<tr>
<td></td>
<td>. character length from 5 to 12 bits</td>
</tr>
</tbody>
</table>

**TABLE 3.1** Major Features of HP 2000
The Software Environment of OLMP

Before the user can really sign on this macro processor (OLMP), he (or she) must have an account associated with an assigned password in the HP/2000 System. The account of this macro processor used in the HP/2000 is a third-level user's account. The organization of the user's account ID code (Identification Code) is combined with the three levels of ID code that represent the three types of users. At the first level, the very first ID code within the system (HP/2000), A000, is assigned to the user who will serve as the system master (system programmer). At the second level, the first ID code within each group (for instance A100, D300, etc.) is assigned to the user who will serve as the group master. At the third level, all other ID codes are available for assignment to private users (4). In figure 3.2, A000 is both the first ID code in the system and the first ID code of the first group. When an ID code is created, an upper limit is set as to the amount of disk space permitted to the ID code for storing programs and files. As the user allocates disk space by storing programs and creating files, he creates a "library" for his ID code. There are three types of libraries associated with those users. The first type, the library assigned to the system master is referred to as the system library. The second type, a library type, the library of every other user is
known as a private library. Each user can access elements in his own private library and to the limit of any restrictions, access programs and files in his group library and in the system library. A user can not access other libraries unless these libraries have been given Program/File Access (PFA) capability.

Since we only care about the Private Library and Group Library of OLMP (macro definition pool can be created in it), a description of these two libraries is given in the following paragraph.

A private library is created and maintained by each ID code. This library is completely controlled by the user assigned that ID code. The user can enter, modify, restrict access to, and delete programs and files within his private library. When an account has Program/File Access capability, its library can (under certain circumstances) be accessed and altered by the group master or by other users. A list of programs and files in a user's library is printed by executing the CATALOG command.

The group library is the library of the group master. The group master may remove any accessing restrictions on individual programs and files in the library so they are accessible to members of the group. If the ID code has Program/File Access, all or part of the library can be made accessible to all other system users. The group master is
responsible for creating, maintaining, deleting, and controlling access to the programs and files within his library. Programs residing in the group master's library are permitted to create, give Multiple Write Access (MWA) status to, read, write, and purge locked BASIC Formatted files in the accounts of group members with PFA, but only if the group master has been given the File Create/Purge (FCP) capability by the system operator. A list of the programs and files in the group library accessible to group members is printed by the GROUP command.

When Program/File Access has been granted to an account, each program and file saved in that library may be accessed if the status of the program or file allows access. The access status of a program or file has four possibilities:

1. Private (default); has no access from another account at all.

2. Locked; can be executed or chained by another account, but not copied or listed.

3. Protected; may be executed, chained to a line number, get, or run by another account.

4. Unrestricted; no restrictions at all.

When a program is saved it is automatically placed in the private status. To assign it another status, we may use one of three commands: LOCK, PROTECT, or UNRESTRICTED. To
(a) The physical record length and the physical record number that we can have in our individual user's account, so far.

(b) The actual data organized in a file.

FIGURE 3.3 File Structure of Disk Files.
return a program to private status, we may use the PRIVATE command.

Example:

LOC-INPUTR
LOC-EDITOR
LOC-MACPRO
LOC-PASONE
LOC-PASTWO
LOC-CORECT

The above examples are named locked programs and may be executed only by the command EXECUTE or CHAIN.

Locked programs are useful when a user wants to allow other users to run his program in its entirety but not be able to list or modify. Locked programs are also useful for accessing locked files. A user's locked program has read and write access to the user's locked files even when the user's program is executed by another user.

The macro processor (OLMP) could be put on a locked status. Only the one who has the account may modify it; another user may run this macro processor (OLMP) through using the command EXECUTE or CHAIN.

When a file is created, it is automatically placed in the locked status. To assign it another status, the user may use one of the commands PRIVATE, PROTECT, OR UNRESTRICT. To return it to locked status, the lock command is used. The status of a file is different from a program's, even if they bear the same status name. A user's created files have four statuses:
1. Private; no access by another account.
2. Locked (default); read/write, access only from owner's Locked program.
3. Protected; read only access.
4. Unrestricted; read/write access.

Why is the file status important to us? After the macro definition library is built, it is very important to protect this macro definition library against any damage or from being totally wiped out by a careless user, so that the most proper file status in a situation like this is "protected". A user can only read a macro definition from a macro definition library, but there is no chance for him to update that library accidentally or incidentally.

Example:

PRO-MPOOL
PRO-INIT
PRO-DOLOOP

The command PRO puts file MPOOL, INIT, and DOLOOP on a protected status.

The disk space that a individual account can get is assigned by the HP/2000 system programmer. The space (see figure 3.3) is organized by blocks and one block can store two hundred and fifty six words maximum (8 bits word).

What are files? A file is a collection of data organized into records and stored on a device which is external to work space, such as disk, paper tape, magnetic
tape, or a deck of cards. The HP/2000 Access System has two types of files: BASIC formatted files and ASCII files (3).

The BASIC formatted files are always on disks and are used for storing large amounts of data such as lists of employees or company inventories.

The ASCII files can be stored on paper tapes, cards, disks, and magnetic tapes; they are frequently referred to as device files. The ASCII files are most useful for directing file data to and from other devices on the system; e.g. to print data on a line printer or to read data from a card deck. They can also be used to store and access information on disks.

We use only the BASIC formatted files in our system; so let us discuss them here.

The BASIC formatted files are used primarily to store data for later retrieval and manipulation. The BASIC formatted files can store both numeric and string data. A pointer is associated with it which is initially set to the first item of the file and which advances sequentially through the data as items are read (can be read serially or directly); thus one can detect whether the next item to be read is a number or a string, or if there is no more data to be read. The BASIC formatted files may be used to store large amounts of data (a BASIC file may contain up to 16.7 million characters of data, depending on the system's
configuration) and this data may be added to, changed, and deleted while running programs.

When data is entered in a BASIC formatted file it is organized on the disk using "records" and "words". Before we can store data in a BASIC formatted file, space for that file must be reserved on a disk, using either the CREATE statement in a program, or the CREATE command. Each permits us to specify the three defining characteristics of a file:

1. Name: 1 to 6 letters or digits; must be unique from all other names in individual user's library.

2. Length in records: can be up to thirty two thousand seven hundred and sixty seven (32,767) records long, depending on the model of the disk (the model 12962A disk subsystem is installed at N.T.S.U.); automatically numbered consecutively from record one.

3. Size of each record: all records physically occupy two hundred and fifty six (256) words of disk storage, although for special programming purposes they may be set to a logical length from sixty-four to two hundred and fifty-six words (default 256 words).

To determine how long the file needs to be, we must estimate how many words of record space each of the data entries will need. Every number in a file occupies two words; every string of characters occupies \((n/2 - 1)\) words -- that is, two characters per word plus one word for the
length of the string. A string with an odd number of characters uses one word for the last character. Figure 3.3 (a) shows the actual physical record size and the record number we can get in the user's account, (b) shows the actual data organized in a file. The data entries in one record do not have to be the same length, and if a data entry cannot be stored in the remaining space of a record, that data item cannot be split up. For instance, the eighth data entry cannot be stored in record two. We cannot split it up and store part of it in the record two and another part in record three. We have to leave the unused area of record two alone and then store the entire eighth data entry into record three.

There are two methods for file access: the serial access methods and the direct access method. Of course, the direct access method is faster than the serial access method, but the shortcoming for the direct access method is that we cannot store more than one data entry in one physical record (BASIC formatted file, HP/2000). We would waste much precious storage space by the use of the direct access method. The space and the speed are two things which we cannot have at the same time. Although we need space desperately, we cannot spend too much time in the macro processing either. So a compromise method is chosen to solve these problems. We discuss how the OLMP handles these
problems in the next chapter.
CHAPTER BIBLIOGRAPHY


CHAPTER IV

THE SIX MODULES OF OLMP

Introduction

The six modules (INPUTR, COMPSR, MACPRO, PASONE, PASTWO, and CORECT) are separate logical units of OLMP. Each of them has its individual function in the OLMP (figure 4.1 shows control flow through these modules). After the cross assembler of MOTOROLA 6800 is loaded into HP/2000, we may either get rid of INPUTR or modify it to link with the input routine of the cross assembler. Major functions, the main control flow, the data structure, and symbolic names which have been used in these modules are discussed. Techniques which have been used to solve OLMP programming problems are also discussed in this chapter.

The Module of INPUTR

The major functions of INPUTR are interacting with the user from an HP/2000 terminal to input source statements (assembler statements and macro definitions), creating source file, and transferring control to the module COMPSR.

The main structures of INPUTR can be interpreted by
FIGURE 4.1 Logical Blocks of OLMP.
The source file "SOURCE" is created first, and the OIMP user has the responsibility of giving the proper block amount. If the file creation fails, the module of INPUTR gives the error message "*** DUPLICATE FILE NAME OR NO SPACE AVAILABLE." After the successful creation of the source file, the module of INPUTR begins to input the source statements from the HP/2000 terminal, until the character string "END" is detected by the module of INPUTR.

The data structure which is used by the module of INPUTR is sequentially organized and fixed in its length (seventy-two characters per logical record). A numerical header occupies the first data entry, and zero is its initial value (details in next section). If the length of one data entry (one source statement) is less than seventy-two characters, then the rest of the logical record will be left blank. The file "SOURCE" accepts up to ninety data entries. If the user's source statements are fewer than ninety, an "END-OF-FILE" mark is generated by the HP/2000 and attached behind the last logical record. Otherwise, if the user's source statements are more than ninety, the HP/2000 system posts an "END-OF-FILE/END-OF-RECORD" error message and the process of the module INPUTR is terminated abnormally. From figure 4.3 we may easily see how the user's input source statements are stored. The first physical record has a header on it, and its usage is given in the following
FIGURE 4.2 Flow Chart of INPUTR.
FIGURE 4.3 Actual Organization of File "SOURCE".
Symbols which have been used in the module of INPUTR

A the return code of file opening
C$(72) the user's source statement is stored here temporarily
CO the header record of the file "SOURCE"
M parameter corresponds to the module "COMPSR"
X The return value of transferring control to COMPSR
Z The return value of "SOURCE" creation

The Module of COMPSR

Major functions of COMPSR are the creation of the file "COMPSD" from the input file "SOURCE" and the compression of the input file "SOURCE".

The only difference between the file "COMPSD" and the file "SOURCE" is the blank padding; the file "COMPSD" has only one blank among the fields in each source statement.

The last task of COMPSR is to pass control to the module of "MACPRO".

The main flow of COMPSR accepts control and a corresponding parameter from INPUTR (the parameter "Q" is the size of the physical record of the file "SOURCE"); then the module of COMPSR creates the file "COMPSD" and makes the size of the file "COMPSD" the same as that of the "SOURCE"
file. If the space in the user's account is not big enough to build up the file "COMPSD", then the error message "*** NO SPACE AVAILABLE........" is given by the module of COMPSR, and the whole process is terminated abnormally.

After the file "COMPSD" is created, the module of COMPSR begins to compress blanks from the input file "SOURCE", leaves one blank among the fields for each source statement, and detects the END-OF-FILE condition of the file "SOURCE" at the same time. The next step is to output the compressed source string to the disk file "COMPSD". Finally control and the parameter "Q" are transferred to module "MACPRO" (figure 4.4).

The data structure used by COMPSR is sequentially organized with variable logical record length. The size of the file is equal to the file "SOURCE".

Symbols used by the module "COMPSR" are:

- A,Z return number from file creation
- B$ (74) temporary storage for compressed string
- C$ (72) temporary storage for source string
- C the header of the file "SOURCE"
- CO,Z9 initial values
- D$ a blank
- J,W character position pointer
FIGURE 4.4 Flow Chart of Rearranging of Source String.
(CONTINUED)
FIGURE 4.4 Flow Chart of Rearranging of Source String.
N, N2  blank position pointer
Q  parameter corresponds to the module "INPUTR"

The Module of MACPRO

The major functions of MACPRO are inputting compressed source statements from the file "COMPSD", creating the file "MADEFT", creating the file "MANAMT", creating the file "SOURC1", detecting the key word "MACRO", detecting the key word "END", calling macro handling routine, and purging the temporary working files (COMPSD, SOURC1, STFILE, MADEFT, and MANAMT).

After MACPRO receives control and the parameter which corresponds to the module "COMPSR", three temporary working files (MADEFT, MANAMT, and SOURC1) are created by the program-controlled file generation. If there is no space available for file creation, then the module MACPRO gives the proper error message and terminates the process. The following step reads the source statements from the file "COMPSD" one after another. Detecting of the existence of the key word "MACRO" and the key word "END" are other two functions of this module. After finding the character string "MACRO", control and two parameters are transferred to the module "PASONE". The first parameter "B6" is used as a pointer to indicate the record after the record which contains the character string "MACRO" in the file "COMPSD".
The second parameter "A$(l)" is a character and serves as a flag to show whether there are errors.

If the input source string does not contain the key word "MACRO", then MACPRO outputs it to the disk file "SOURCE1". To check the "END" condition is the next step. If the key word "END" has been detected, then the module of "MACPRO" checks the error flag; otherwise the program "MACPRO" goes back to bring a new source statement in and repeat the above processing.

The module "MACPRO" receives control and two corresponding parameters from the module "PASONE" and repeats the procedure to bring a new source statement in and to check the validity of that new statement.

If the error flag is off (no error), then control passes to the module "PASTWO". After control returns from the module "PASTWO", MACPRO purges the file "COMPSD", file "SOURCE1", file "STFILE", file "MADEFT", and file "MANAMT" before transferring control to the MOTOROLA M6800 assembler.

If the error flag is on, then control and the parameter "T" pass to the module "CORECT". The parameter "T" is used as a storage to store the size of the file "SOURCE". After control passes back from the module "CORECT", the module "MACPRO" transfers control and the parameter "T" to the module "COMPSR" again and repeats the whole procedure.

Three files (MADEFT, MANAMT, and SOURCE1) are created in
the module "MACPRO".

We store source statements without the key word "MACRO" into the file "SOURC1". The file "SOURC1" is a sequential file with variable length. The physical and logical organization of this file is the same as file "COMPSD", but without the header record (figure 4.5).

Symbols used in the module of MACPRO are
A,R,Z return codes from file creations
A$(1),B6 parameters correspond to the module "PASONE"
B5 the header record of the file "COMPSD"
C$(72) temporary storage for the compressed string
C0 initial value
C1 the header record of the file "MADEFT"
C2 the header record of the file "MANAMT"
MO$(7) string constant "MACRO"
NO$(5) string constant "END"
T this parameter corresponds to the module "COMPSR"

The Module of PASONE

Major functions of "PASONE" are to read in macro definitions from the file "COMPSD", to scan macro definitions, to give major syntax error messages, to build the definition table (MADEFT), and to set the macro name table (MANAMT).
FIGURE 4.5 Actual Organization of File "COMPSD".
FIGURE 4.6 Flow Chart of MACPRO.
The main logical flow of "PASONE" accepts control and the two corresponding parameters (C6 and F$(l)$) from MACPRO (figure 4.6). The first thing that PASONE has to do is to detect the END-OF-FILE condition of file "COMPSD". If it is the END-OF-FILE condition, then control and two parameters are passed back to MACPRO; else PASONE starts to input macro definitions from the file "COMPSD". The following steps are to check the comment field, the macro label field, the macro name field, and the formal argument field (figure 4.7). Next the control flow goes to check for error conditions. If there are one or more syntactic errors, then PASONE gives the proper error message and turns on the error flag; else it sets up the macro definition table and the macro definition counter. Checking for the key word "MEND" gives the PASONE a signal when the macro definition reaches the end. When "MEND" is detected, then control flow checks for error flags and starts to set up the macro name table and the counter; else PASONE goes to input another macro definition statement from the file "COMPSD" and repeats the above described process.

If a multiple GOTO statement is introduced, the route of control flow depends on control variable "N" (figure 4.7). If the macro definition statement is not a comment record, then we check the macro label field (N=1), the macro name field (N=2), and the formal argument field (N=3).
FIGURE 4.7 Flow Chart of PASONE. (CONTINUED)
FIGURE 4.7 Flow Chart of PASONE.

(Continued)
FIGURE 4.8 Flow Chart for Setting up Macro Definition, Table, and Macro Name Table.
Finally, after the list of formal arguments is built up, control is handed over to the next step.

In figure 4.8 (a), PASONE tries to locate the character "&" in the compressed source string "B$". If there is no character "&", then we go to check the error flag; else PASONE locates the next delimiter, a blank or a comma (,) from the compressed source string "B$". To find the macro label or the argument in the argument list is very important. We can use the argument list as a map to make sure that the newly located macro label or that the argument is already defined in the argument list (the label or argument name must be unique). If it is a unique name then the next step is to replace the macro label or argument by the mark "~" and to record its position number into the argument list; else the error flag is turned on. If there are more "&" detected by PASONE, then the above procedure is repeated; else the macro definition table and the counter are prepared after the checking of error flag.

Refering to figure 4.8 (b), in order to keep the correct record position in the file "MADEFT", we have to update the file header which stores the current available space position in the file "MADEFT". For the same reason, the file header of "MANAMT" and the file header of "COMPSD" are updated.

The creation of two files (MADEFT and MANAMT) is the
main job of PASONE. In figure 4.9 (a), C2 contains the next available empty space position for the coming data entry, C3 stores the serial number in a macro definition, and A9 has the serial number of the whole input source statements. Only five data entries can be stored in one physical block. That makes 34 words left as an unused area for the first block (header record occupies two words), and there are 36 words left for each of the remaining blocks. The amount of physical blocks in the file "MADEFT" is indicated on the corresponding parameter "T", which was transferred from the module MACPRO. The value of parameter "T" is equal to the size of the file "SOURCE". The macro definition is stored in the file "MADEFT" temporarily. The string "B$" stores 72 characters maximum and equals 36 words in the physical block. Additionally, we still need one word for the header, one word for the record length information, and one word for the END-OF-RECORD mark.

In figure 4.9 (b), before the serial number and the next available space position information can be stored into the file "MANAMT", we have to calculate their values in the first place. Two simple equations, N9+1 and N6/3+1, are used. The first equation, (N9+1), specifies the correct next available space position, and the second equation, (N6/3+1), finds the serial number for each macro name record. The first physical block may have up to
FIGURE 4.9 (a) Organization of File "MADEFT". (CONTINUED)
(CONTINUED)

**FIGURE 4.9 (b)** Organization of File "MANAMT".
twenty-eight data entries and one word left unused, but the second block has two words left unused.

The argument list "P$" is a character string; two hundred and fifty-two characters maximum are allowed. We divide P$ into 36 sections (seven characters per section) and store arguments or labels in these sections.

Symbols used in the PASONE module are:

- **A9** the position number of prototype record
- **B$(72)** temporary storage for compressed string
- **C** argument counter
- **C2,Z9** initial value
- **C3** the sequence number of a macro definition statement
- **C5,N,N9** values of the header of file "MADEFT"
- **C6,F$(1)** parameters which correspond to the module "MACPRO"
- **D$** character constant " "
- **E$** character constant ","
- **G$(2)** string constant " "

- **H,N0,N1, N3,P0,P1, P2,W** character position pointers
- **M1$(6)** string constant " MEND "
- **N** jump table control number
- **N$(7)** temporary storage for a formal argument
- **N5** value of the header of file "COMPSD"
The Module of PASTWO

Major functions of PASTWO are to detect the predefined macro definition, to expand the predefined macro definition, and to detect the macro call within another macro definition.

The module of PASTWO creates the file "STFILE" before any initialization (figure 4.10). If the END-OF-FILE condition of file "SOURC1" is detected, then control is transferred back to MACPRO, else the module begins to read source statements from file "SOURC1". After control returns from the subroutine which checks macro name and label (figure 4.11), if the source string is a predefined macro definition, then the module PASTWO pushes the source string down into the stack file "STFILE" and passes control to the second subroutine (setting up argument list array, figure 4.12). Otherwise the module simply prints out the source string $C$ and goes back to check the END-OF-FILE condition of file "SOURC1", and repeats the above journey. Next comes the step of macro expansion (figure 4.13), which depends on the successful creations of file "SOURC1", "MADEFT", "MANAMT", and the argument array $P$$. The module "PASTWO"
FIGURE 4.10 Flow Chart of PASTWO.
FIGURE 4.11 Flow Chart of Subroutine for Checking Macro Name, Label, and Setting up Stack.
FIGURE 4.12 Flow Chart of Subroutine 1430 for Setting up the Argument List Array.
FIGURE 4.13 Flow Chart of the Macro Expansion Section.
expands the predefined macro definition, which is located in the file "SOURC1", and builds the expanded source file. (OLMP only prints the expanded source file through the user's terminal, so far.)

Refering to figure 4.11, the major task of this subroutine 1430 is to check the source string C$ against the predefined macro names which are stored in the file "MANAMT". If the value of C$ is found in the file "MANAMT", then the program sets the error flag to one; else zero is assigned. After the return pointer is pushed down into the stack S (details about stack S are introduced in the next paragraph), control flow returns to the main routine "PASTWO".

In figure 4.12, how does the program (PASTWO) know where those arguments belong? Well, we have a long order and delivery list to be built before we know what an argument replacing value belongs to, where to find the corresponding argument, and where to locate that argument replacing value. The argument list array plays a vital part in this situation. If an argument is found in macro prototype string, then we move it to the argument list array P$ (7 characters are used for one argument name); else we move blanks to this position. We start to build up the rest of the argument list array by the use of the delimiter in the prototype string. If the delimiter "," is located, then we
move the argument to that list. If a blank is detected, then we know where the last argument must to join the argument list.

The expansion of the macro definition is the final purpose of OLMP. In figure 4.13, we have to determine whether the record which is from the file "MADEPT" contains the key word "MEND" or not. If the answer is no, then we start to search for the special mark "~". After the mark "~" is found, we use the number behind that mark as a position number, then compare the matched position against the argument list array. The matched argument is provided to replace the matched position number in the macro definition. After printing that expanded macro definition, the search for the macro call within another macro definition is performed. If there is a macro call within another macro definition, then the stack file "STFILE" is updated to store the return point of the calling macro definition. If the key word "MEND" appears in the string C$, then we notify the stack S and file STFILE to pop up the return pointer and the return string that contains this macro call. The final job is to check the stack. An empty stack tells us there are no other macro calls within another macro definition; otherwise, we have to go back and repeat the process.

The argument list array has the same data structure mentioned in the above paragraph of PASONE. Details of the
stack S and the stack file "STFILE" are described in the following sections.

The array variable S is introduced to function like a stack (refering 1, 2). It is a two-dimensional array (1 x n); that means that S is a one row and n columns array. How is it used as a stack in this macro processor? The following example tells us all about array S.

If there is a two-dimensional array S(1,10) and three return pointers 16, 17, and 18, then the logical structure of array S is like this:

```
| (1,1) | (1,2) | (1,3) | ... | (1,10) |
```

after initialization

```
| 0   | 0   | 0   | 0   | ... | 0   |
```

Let the first element of this array S be the stack pointer and it always points to the first empty element,

```
| 2   | 0   | 0   | 0   | ... | 0   |
```

so that, if we say S(1,S(1,1)), that means the second element of the array S (S(1,S(1,1))=S(1,2)). Now let S(1,S(1,1))=16 and S(1,1)=S(1,1)+1. At this moment the return pointer 16 is pushed into the stack S, and the stack pointer points to the third available storage.

```
| 3   | 16  | 0   | 0   | ... | 0   |
```

The same method can apply to the return pointer 17
and the return pointer 18.

Here is how the pointers pop up. The stack pointer minus one first, $S(1,1)=S(1,1)-1$. At this time the stack pointer points to the fourth element of stack $S$, which is the return pointer 18 ($S(1,S(1,1))=18$). After the return pointer 18 pops up, a zero is moved into $S(1,S(1,1))$.

The same method can be applied to the return pointers 17 and 16 also.

The file "STFILE" is used as a stack also. Its size is three physical blocks, so we can have up to eighteen prototype strings stored in this stack file. The file "STFILE" is a sequential file, but we may control the starting point (jumping over the unwanted record), so that logically the file "STFILE" is a stack. There is a statement of HP/2000 Access Basic, ADVANCE, which is used to accomplish this task.

Symbols used in this module "PASTWO" are

A,R,Z,20 \quad \text{return codes from file creation}

A9 \quad \text{initial value used to calculate the pointer which is stored in the Stack S}

B$(77), \quad \text{temporary storage for macro definition}

D$(72)
The Module of CORECT

The major functions of CORECT are updating the main source file "SOURCE", inserting new statements to the file "SOURCE", and deleting the old statements from the file "SOURCE".

The OLMP user builds up his source file first, before he can call the macro processor to expand his source program. If one or more than one error is detected in the user's source program, then what is this user to do? There are two ways in which he might proceed. The first way is to purge the source program and rebuild his new source program. The second way is correct the error statements in his source program without rebuilding it. We can easily see that most
users would prefer to correct the errors in the source program instead of rebuilding it.

Figure 4.14 tells us how the source file retrieval program is constructed. The first step is that the user has to decide whether to update his source statements, to insert new statements into the source file, or to delete the unwanted source statements. If the user wants to insert or to delete statements, then a new file "BUFFER" must be created in order to rebuild the source file internally without the user's cognizance. After the correction job is done, the user may list the new master source file, which is stored in the file "SOURCE".

The file "BUFFER" serves as a buffer or working file to store the OLMP user's source statements temporarily for retrieving. Its data structure is sequentially organized and its size is the same as the file "SOURCE". The access to this file is sequential also.

The array variable S is used to holding the user's input sequential ascending numbers, which may be employed as the pointers of the access control. These numbers are sequenced numbers of the source statements. The array S contains forty elements so far; so we can update up to forty source statements at a time.

Symbols used in the module "CORECT" are

C,K the header of the file "SOURCE"
FIGURE 4.14 Flow Chart of CORECT.
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C$(72)</td>
<td>temporary storage for source string</td>
</tr>
<tr>
<td>C0</td>
<td>the header record of the file &quot;SOURCE&quot;</td>
</tr>
<tr>
<td>I</td>
<td>source statement sequence number</td>
</tr>
<tr>
<td>M</td>
<td>internal counter</td>
</tr>
<tr>
<td>N,R,Z</td>
<td>return codes of file creations</td>
</tr>
<tr>
<td>S(40)</td>
<td>temporary storage for the user's input source statement sequence number</td>
</tr>
<tr>
<td>W</td>
<td>parameter corresponding to the module &quot;MACPRO&quot;</td>
</tr>
</tbody>
</table>

Locked status for these six modules is suggested. Locked status for the OLMP assures the integrity of this macro processor. The user who is responsible for making innovations on the OLMP should have the account number and password for OLMP.
CHAPTER BIBLIOGRAPHY


CHAPTER V

CONCLUSION

The Future Improvement of OLMP

The cross assembler of MOTOROLA 6800 microprocessor needs to be linked to OLMP. Then there will be a complete preprocessor for M6800.

In module PASONE, the conditional assembly routine (macro "IF"), the macro GOTO, and the macro variables (local and global variables) can be added into OLMP as new functions. By adding the HP/2000 ACCESS BASIC statement "GOSUB" between the statement number four hundred fifty (450) and four hundred sixty (460) in the PASONE program, the control can transfer to the conditional assembly, macro GOTO, and macro variable handling routines.

The array variable S, which is used in PASTWO as a stack, and the file "STFILE", which plays the same role as the stack S, can be enlarged. Through the stack file and the macro "IF" we might even allow recursive macro definitions.

The CORECT routine can handle up to forty changes at one time. Enhancement can be provided to allow more changes
To build a macro definition library is one of our goals. We may put our generalized macro definition into the macro definition library and keep a catalog which contains information about this macro library.

After the macro library is built, the security of that macro library is the first thing which needs to be accomplished. The command "PROTECT" of the ACCESS BASIC is recommended to do this job.

The future hardware environment of M6800 is the main key to affect our software development. Although the HP/2000 serves M6800 as a preprocessor, after the object codes are generated by HP/2000, it still leaves a problem: how can we send the object code back to M6800? Do we need a hardware interface, a software interface, or both? These questions remain to be solved.

For what we have so far, we can utilize Teletype paper tape to record executable object codes from the cross assembler. That saves us the time to key object codes into M6800 through the keyboard and reduces the introduction of errors. In addition, we can use a telephone and an acoustic coupler to transfer object codes back to the M6800 (1).

How to Connect the OILMP with the Cross Assembler

If the cross assembler is available and is loaded into
HP/2000, then the following steps must be taken to assure perfect linkage condition between this OLMP and the cross assembler:

The ACCESS BASIC statement "CHAIN" must be used to transfer control from cross assembler to the OLMP.

Use the statement "CHAIN" again to pass control back from the module "MACPRO" of OLMP to the cross assembler after a macro expansion (attach statement "CHAIN" behind statement number 1020 of the module "MACPRO").

The statement "COM" of ACCESS BASIC must be used to transfer the size parameter of the file "SOURCE".

We might get rid of the module "INPUTR", but then the cross assembler would have to take over the task of the source file generation. Otherwise, the cross assembler must use the module "INPUTR" as its input routine.

Be sure the user's source statements are stored in the file named "SOURCE". The file "MANAMT" should be reserved until the object code is generated by the cross assembler. Then the statement 950 and 990 of MACPRO should be discarded.

After macro expansion, the expanded source program must be stored somewhere for future reference (such as the source program listing).

The final job is for the user to utilize the program "CORECT" to keep his source program in shape.
Comment

Six modules of OLMP are combined as a macro processor for the M6800. The central control program "MACPRO" is used and the program control flow indicator is introduced in the meantime to help in monitoring the control flow.

A smaller program is much easier to design, to manage, to debug, and to revise, but to keep on increasing the number of modules and decreasing the size of modules does not always result in a positive impact.

Having more modules involved in a software package increases the likelihood of problems during the linkage of the modules. What is the proper number of modules in a software package? It is not a very easy question to have a short answer for, but through the concept that every program must perform a prearranged function, maybe we can find a reasonable answer.

The whole software package is a predesigned task, and this task can be broken down into several smaller subtasks. (Of course, each of these subtasks can be divided again, but we are talking about the first level of subtasks.) Because the software in a system is changed with its hardware environment and the user's new application, the task of a module should be innovated in order to meet the new environment and the user's new application. The user's new application
which implies part logic blocks of the old software package must be deleted, updated, or added as new subtasks.

The subtask should be the final result of the accomplishment of each modules in a software package. We might say that the total number of subtasks that can be carried in a task will be a proper total number of modules.

The central control method eases the problem of linking modules. Taking this macro processor, for instance, the connection between the assembler and the macro processor has many things we have to pay attention to; but as a matter of fact, only the first one has to do connections (MACPRO).

When we start to design a module, we should be aware of the following two things:

There is a central control program that serves the software as a control panel. A well structured program will save us the designing, coding, and debugging. And, most of all, it is easy revise in the future.

The control flow should be one inlet and one outlet. Multiple control outlets are much more likely to cause control trouble than one.

HP/2000 has two file access methods, the sequential file access and the direct (random) file access. In order to increase access speed, we would prefer to use the direct access method other than the sequential access method. But there is no variable record length feature in the direct ac-
cess method of HP/2000. All records in the BASIC formatted file for the direct access are fixed (256 words). We waste a lot of precious space to support this direct access method. This is not a very brilliant way to handle the file in the relatively small storage space we have.

The indexed sequential file (logically) is introduced to compensate the confliction between storage saving and access speed. Physically, we have to reinitialize the read head of the disk driver for each single access. After reading the prearranged value which is recorded in the record head, the program will control the read head of disk driver jumps over a certain amount of unwanted records. Finally, it gets to the record we need. Though it is a time-consuming job to repeat the above action to update the record head of the disk file every time we access the files, it is still faster than the sequential access method. We can assure that storage space and access time are saved by using indexed sequential access method in a condition like this.

From the designing of this macro processor, I noticed that the HP/2000 ACCESS BASIC has useful properties of file access speed, data protection, and easily understood BASIC statements. But, on the other hand, the fixed record length of the direct file access and the fixed individual account storage sometimes cause a very inconvenient condition when we use it to develop our own software system like the OLMP.
In addition, we cannot have a memory dump from abnormal termination. This increases the difficulties of the development of a relatively large and complicated software package.
CHAPTER BIBLIOGRAPHY

#(ad,K1,K2,K3)  (ADD) If no overflow, then K1+K2, else K3.

#(bc,K1)  (BOOLEAN COMPLEMENT) Not K.

#(bi,K1,K2)  (BOOLEAN INTERSECTION) K1 K2.

#(br,K1,K2)  (BOOLEAN ROTATE) Rotate K2 left K1 places. If K1 < 0, rotate right K1 places.

#(bs,K1,K2)  (BOOLEAN SHIFT) Shift K2 left K1 places. If K1 < 0, shift right K1 places.

#(bu,K1,K2)  (BOOLEAN UNION) K1 U K2.

#(cc,K1,K2)  (CALL CHARACTER) The value is the character under the form pointer. If the form is empty, the value is K2. The form pointer is moved one character ahead (segment gaps are skipped).

#(c1,K1,K2,...,Km)  (CALL) Call segmented form and has two or more arguments. The form named K1 is retrieved from storage and its segment gaps of code marking 1 are filled with string K2, the gaps of marking 2 are filled with string K3, etc.

#(cm,K1)  (CHANGE META) This null-valued function changes the meta character to the first character of the string symbolized by K1. Upon starting, the TRAC processor is loaded with a standard meta character, usually the apostrophe.
#(cn,K1,K2,K3)  (CALL n CHARACTERS) Read K2 characters of the form named K1, starting at the character pointed to by the form pointer, and increment the form pointer by K2. If K2 is negative, read characters preceding the form pointer. If no characters are available, the value of the function is K3.

#(cr,K1)  (CALL RESTORE) This null-valued function restores the pointer of the form named K1 to its initial character.

#(cs,K1,K2)  (CALL SEGMENT) The value of this function is the string form the current position of the form pointer to the end of the next segment gap. The form pointer is moved beyond the segment gap. If the pointer is initially beyond the last character of the form, the value of the function is K2.

#(da)  (DELETE ALL) This null-valued function deletes all the forms in memory and removes their names.

#(dd,K1,K2,...,Km)  (DELETE DEFINITION) This null-valued function deletes the forms named K1, K2, etc., from memory and removes their names from the list of names.

#(ds,K1,K2)  (DEFINE STRING) Define the string determined by evaluating the second parameter K2 to have as its name the string obtained by evaluating the first string K1.

#(dv,K1,K2,K3)  (DIVIDE) If no overflow, then K1/K2, else K3.

#(eb,K1)  (ERASE BLOCK) This null-valued function erases the form named K1 and also the group of forms in the block in external storage.

#(eq,K1,K2,K3,K4)  (EQUALS) If K1 is equal to K2 the function value is K3; otherwise it
is K4.

#(fb,Kl) (FETCH BLOCK) This null-valued function fetches from auxiliary storage the forms stored in the block named Kl.

#(gr,Kl,K2,K3,K4) (GREATER THAN) If the decimal number Kl is greater than K2, the function value is K3; otherwise the function value is K4.

#(in,Kl,K2,K3) (INITIAL) Search the form named Kl for the first occurrence of the string K2 following the current pointer position in the form named Kl. If a match is found, the function value is the portion of the form between the form pointer and the matched string, and the pointer is moved to the first character beyond the matched string. If no match is found the pointer is not moved and the value of the function is K3.

#(ln,Kl) (LIST NAME) List all names in the pair list with each name preceded by string Kl. If Kl is the character pair CR LF, names are listed in a column.

#(ml,Kl,K2,K3) (MULTIPLY) If no overflow, then Kl x K2, else K3.

#(pf,Kl) (PRINT FORM) This function causes the typing out of the form named Kl with a complete indication of the location and ordinal values of the segment gaps.

#(ps,Kl) (PRINT STRING) Print out the string named Kl.

#(rc) (READ CHARACTER) The value is the next character, which may be any character (including the meta character) received from the teletypewriter.

#(rs) (READ STRING) Read a string of
characters up to an END-OF-STRING symbol.

\#(sb,K1,K2,\ldots,Km)

(STORE BLOCK) This null-valued function stores the forms K2, K3, \ldots, Km as a single record of auxiliary storage and calls this block K1. All pointers are saved. A form named K1 having as content the auxiliary storage address is set up in the processor.

\#(ss,K1,K2,\ldots,Km)

(SEGMENT STRING) First evaluate the name K1 and parameters, then call the named string K1 and replace occurrence of Ki in the string by parameter markers for i=2, 3, \ldots, m. Then store the result back in the memory.
APPENDIX B

THE SOURCE PROGRAM OF OLMP

10 COM M
20 REM * THIS PROGRAM IS CALLED "INPUTR" *
30 REM * FUNCTION: *
40 REM * . INPUT TEST DATA FOR OLMP *
50 REM * . CREATE TEST DATA SOURCE FILE *
60 REM * . TRANSFER CONTROL TO PROGRAM COMPSR *
80 REM * *************************************** *
90 REM
100 FILES *
110 DIM C$[72]
120 PRINT "HOW MANY BLOCKS DO YOU ESTIMATE FOR SOURCE DATA?"
130 PRINT "*** NOTE: ONE BLOCK = 6 SOURCE STATEMENTS"
140 PRINT "*** BE SURE YOU HAVE ENOUGH SPACE FOR FUTURE"
150 PRINT " *** PROGRAM UPDATING."
160 INPUT M
170 CREATE Z,"SOURCE",M
180 IF Z=0 THEN 210
190 PRINT "*** DUPLICATE FILE NAME OR NO SPACE AVAILABLE."
200 STOP
210 ASSIGN "SOURCE",1,A
220 C0=0
230 PRINT #1;C0
240 PRINT " *** START TO INPUT YOUR SOURCE STATEMENTS, PLEASE"
250 INPUT C$
260 PRINT #1;C$[1,72]
270 IF POS(C$;" END ")=0 THEN 250
280 PRINT #1; END
290 PRINT "*** INPUTR --->"
300 CHAIN X,"COMPSR"
310 END
105

10 COM Q
20 REM *********************************************
30 REM * THIS PROGRAM IS CALLED "COMPSR" *
40 REM * FUNCTION* *
50 REM * . INPUT SOURCE STATEMENT FROM FILE-SOURCE *
60 REM * . REARRANGE INPUT STRING, AND LET THE *
70 REM * . INPUT STRING BE INSERTED ONLY ONE BLANK *
80 REM * AMMONG THOSE FIELDS *
90 REM * . CREATED FILE COMPSD *
100 REM * . PASS CONTROL TO PROGRAM MACPRO *
110 REM *********************************************
120 REM
130 FILES SOURCE,*
140 PRINT "***---> COMPSR."
150 DIM C$[72],B$[74]
160 C0=Z9=0
170 CREATE Z,"COMPSD",Q
180 IF Z=0 THEN 210
190 PRINT "*** NO SPACE AVAILABLE FOR FILE 'COMPSD'."
200 STOP
210 ASSIGN "COMPSD",2,A
220 PRINT #2;C0
230 READ #1;C
240 IF END #1 THEN 500
250 READ #1;C$
260 REM REARRANGE INPUT CHARACTER STRING..............
270 J=W=1
280 B$=""
290 D$=""
300 IF C$[1,1]=" " THEN 320
310 IF C$[1,1]<>" " THEN 340
320 W=2
330 REM THE FOLLOWING DO LOOP IS USED TO SKIP BLANKS.....
340 FOR I=J TO LEN(C$)
350 IF C$[I,I] <> " " THEN 390
360 J=J+1
370 NEXT I
380 IF Z9=0 THEN 470
390 N2=POS(C$[J],D$)
400 IF N2=0 THEN 450
420 W=W+N2
430 J=J+N2
440 IF Z9=0 THEN 340
450 B$[W,74]=C$[J,LEN(C$)]
460 IF Z9=0 THEN 380
470 N=POS(B$," ")
480 PRINT #2,B$[1,N+1]
490 IF Z9=0 THEN 250
500 PRINT #2; END
510 PRINT "*** COMPSR --->")
CHAIN A,"MACPRO"
END
COM T
COM B6, A$[1]
REM **************************************************************
REM * THIS MACRO PROCESSOR IS FOR MOTOROLA 6800 WHICH *
REM * CONTAINS THREE MAJOR PROGRAMS: *
REM * MACPRO *
REM * PASONE *
REM * PASTWO *
REM **************************************************************
REM
REM ******************************************************
REM THIS PROGRAM IS CALLED "MACPRO"
REM FUNCTION:
REM . INPUT COMPRESSED SOURCE STRING
REM . ECHO INPUT STRING
REM . CALL MACRO HANDLING ROUTINE
REM . DETECTING THE "END" CONDITION
REM ******************************************************
FILES *,*,*,COMPSD
PRINT "*** ---> MACPRO."
CREATE Z, "MADEFT", 2*T/3
IF Z=0 THEN 260
PRINT "*** NO SPACE AVAILABLE FOR 'MADEFT'."
STOP
ASSIGN "MADEFT", 1, A
CREATE Z, "MANAMT", T/3
IF Z=0 THEN 310
PRINT "*** NO SPACE AVAILABLE FOR 'MANAMT'."
STOP
ASSIGN "MANAMT", 2, A
CREATE Z, "SOURC1", 3*T/4
IF Z=0 THEN 360
PRINT "*** NO SPACE AVAILABLE FOR 'SOURC1'."
STOP
REM
C0=0
A$=" 
PRINT #1; C0
PRINT #2; C0
GOTO 430
PRINT "*** ---> MACPRO."
DIM CS[72], MO$[7], NO$[5]
ASSIGN "MADEFT", 1, A
ASSIGN "MANAMT", 2, A
ASSIGN "SOURC1", 3, A
MO$=" MACRO "
NO$=" END 
READ #4; B5
B6=B5+1
ADVANCE #4; B5, Z
108

520 REM
530 REM *-----------------------------------------------*
540 REM * INPUT TESTING DATA FROM "COMPSD" AND ECHO DATA *
550 REM *-----------------------------------------------*
560 REM
570 READ #4;C$
580 PRINT $6,C$(1,POS(C$," ")
590 NO=POS(C$,M0$)
600 IF NO <> 0 THEN 1090
610 PRINT #3;C$
620 NO=POS(C$,NO$)
630 IF NO=0 THEN 570
640 READ #1;C1
650 READ #2;C2
660 ADVANCE #1;C1,Z
670 PRINT #1; END
680 ADVANCE #2;C2,Z
690 PRINT #2; END
700 PRINT #3; END
710 IF A$ <> "E" THEN 900
720 REM
730 REM *-----------------------------------------------*
740 REM * IF ERROR FLAG IS SET, THEN GO TO 'CORECT'. *
750 REM *-----------------------------------------------*
760 REM
770 PRINT "*** THERE IS ERROR SOMEWHERE IN THE MACRO DEF."
780 PRINT "*** Macpro --->"
790 CHAIN A,"CORECT"
800 PRINT "*** ---> Macpro."
810 PRINT "*** Macpro --->"
820 CHAIN A,"COMPSR"
830 PRINT "*** 'COMPSR' CALLING FAILURE."
840 STOP
850 REM
860 REM *-----------------------------------------------*
870 REM * IF ERROR FLAG IS NOT SET, THEN CALL "PASTWO". *
880 REM *-----------------------------------------------*
890 REM
900 PRINT "*** Macpro --->"
910 CHAIN A,"PASTWO"
920 PRINT "*** ---> Macpro."
930 PRINT "PASS CONTROL TO ASSEMBLER IF IT IS THERE."....
940 ASSIGN *,1,A
950 ASSIGN *,2,A
960 ASSIGN *,3,A
970 ASSIGN *,4,A
980 PURGE R,"MADEFT"
990 PURGE R,"MANAMT"
1000 PURGE R,"SOURC1"
1010 PURGE R,"COMPSD"
1020 PURGE R,"STFILE"
1030 STOP
1040 REM
1050 REM *-----------------------------------------------*
1060 REM * IF "MACRO" IS DETECTED, CALL "PASONE". *
1070 REM *-----------------------------------------------*
1080 REM
1090 ASSIGN "COMPSD", 4, A
1100 UPDATE #4; B5+1
1110 B6=B6+1
1120 PRINT "*** MACPRO --->"
1130 CHAIN R,"PASONE"
1140 END
10 COM X
20 COM C6,PS[1]
30 REM ************************************************************
40 REM * THE SECOND PROGRAM IS CALLED "PASONE" *
50 REM * FUNCTION: *
60 REM * INPUT MACRO DEFINITION *
70 REM * SCANNING *
80 REM * CHECK COMMENT *
90 REM * CHECK LABEL FIELD *
100 REM * CHECK MACRO NAME FIELD *
110 REM * CHECK ARGUMENT FIELDS *
120 REM * BUILD MACRO DEFINITION TABLE *
130 REM * BUILD MACRO NAME TABLE *
140 REM * BUILD COUNTERS *
150 REM * BOOKKEEPING *
160 REM * OUTPUT ERROR MESSAGE *
170 REM ************************************************************
180 REM
190 REM
200 REM INITIALIZATION.............
210 REM
220 PRINT "*** ---> PASONE."
230 FILES MADEFT,MANAMT,*,COMPSD
240 DIM B$(72),N$(7),M$(6)
250 DIM P$(252)
260 DIM G$(21
270 M1$=" MEND "
280 E$="",
290 D$=""
300 F$=""
310 G$=""
320 C2=Z9=0
330 W=8
340 C3=C=H=D=1
350 READ #4;N5
360 READ #1;C5
370 A9=C5/4
380 ADVANCE #1;C5,Z
390 N9=C5
400 ADVANCE #4;N5,Z
410 IF END #4 THEN 1810
420 N=0
430 READ #4;B$ 
440 PRINT C6,C3;B$
450 C6=C6+1
460 IF B$(1,1)="*" THEN 430
470 IF N >= 3 THEN 1370
REM *--------------------------------------------------*
REM * SCANNING.                                       *
REM * CHECK COMMENT                                    *
REM * CHECK LABEL FIELD                                 *
REM * CHECK MACRO NAME FIELD                           *
REM * CHECK ARGUMENTS FIELD                            *
REM *--------------------------------------------------*
N=N+1
GOTO N OF 600,820,960
REM
CHECK MACRO LABEL FIELD....................... 
IF B$(1,1] <> " " THEN 660
NO=1
P$[1,252]="   
IF B$(1,1]=" " THEN 840
IF B$(1,1]="&" THEN 690
PRINT "*** ERROR -- WRONG LABEL FORMAT!"
IF B$(1,1] <> "&" THEN 1240
NO=POS(B$,D$)
IF NO <= 8 THEN 740
PRINT "++ WARNING -- LABEL TOO LONG, ONLY 7 CHAR. ARE USED!"
P$[1,252]=B$[1,7]
IF NO > 8 THEN 750
P$[1,252]=B$[1,NO-1]
IF B$(NO+1,NO+1] <> " " THEN 780
PRINT "*** ERROR -- MISSING MACRO NAME AND ARGUMENT FIELDS!"
IF B$[NO+1,NO+1]=" " THEN 1240
IF B$[NO+1,NO+1] <> "-" THEN 840
NO=1
IF Z9=0 THEN 1580
REM
CHECK MACRO NAME FIELD........................ 
N1=POS(B$[NO+1],D$)
IF N1 <= 8 THEN 890
PRINT "++ WARNING -- MACRO NAME TOO LONG, ONLY 7 CHAR. ARE USED!"
N$[1,7]=B$[NO+1,NO+7]
IF N1>8 THEN 900
N$[1]=B$[NO+1,NO+N1-1]
N1=NO+N1
IF B$[N1+1,N1+1] <> "-" THEN 980
N1=1
N=2
IF Z9=0 THEN 1580
REM
REM
REM
980 IF BS[N1+1,N1+1]="&" THEN 1030
990 IF BS[N1+1,N1+1]="-" THEN 1210
1000 IF POS(BS[N1+1],E$)=0 THEN 1250
1010 PRINT "*** ERROR -- WRONG ARGUMENT FORMAT!"
1020 IF BS[N1+1,N1+1]<>"-" THEN 1240
1030 IF C<=36 THEN 1060
1040 PRINT "*** ERROR -- TOO MANY ARGUMENTS!"
1050 IF C>36 THEN 1240
1060 C=C+1
1070 N3=POS(BS[N1+1],E$)
1080 IF N3<>0 THEN 1110
1090 N3=POS(BS[N1+1],D$)
1100 IF Z9=0 THEN 1120
1110 IF BS[N1+N3+1,N1+N3+1]=" " THEN 1010
1120 N0=N1
1130 N1=N1+N3
1140 IF LEN(BS[N0+1,N0+N3-1])<=7 THEN 1180
1150 PRINT "+++ WARNING -- ARGUMENT TOO LONG, ONLY 7 CHAR.
ARE USED!"
1160 P$[W,252]=BS[N0+1,N0+7]
1170 IF Z9=0 THEN 1190
1180 P$[W,252]=BS[N0+1,N0+N3-1]
1190 W=W+7
1200 IF Z9=0 THEN 980
1210 N1=1
1220 N=2
1230 IF Z9=0 THEN 1580
1240 FS="E"
1250 N=3
1260 IF Z9=0 THEN 1230
1270 REM
1280 REM *-------------------------------------------------*
1290 REM * SEP UP MACRO DEFINITION TABLE AND COUNTER.  *
1300 REM * USE ISAM TO BUILD TWO FILES:                *
1310 REM *       MADEFT                                    *
1320 REM *       MANAMT                                   *
1330 REM *-------------------------------------------------*
1340 REM
1350 REM SET UP MACRO DEFINITION TABLE AND COUNTER.......
1360 REM
1370 P0=POS(BS[H],"&")
1380 IF P0=0 THEN 1580
1390 P1=POS(BS[H+P0],")")
1400 IF P1<0 THEN 1420
1410 P1=POS(BS[H+P0],")")
1420 P"=POS(P$,BS[H+P0-1,H+P0+P1-2])
1430 IF P2<0 THEN 1470
1440 PRINT "*** ERROR -- UNDEFINED MACRO LABEL/ARGUMENT."
1450 FS="E"
1460 GOTO 1580
1470 D=LEN(P$[1,P2-1])/7+1
1480 IF D <= 9 THEN 1520
1490 D=D+55
1500 GS=CHR$(D)
1510 GOTO 1530
1520 CONVERT D TO GS
1530 BS[H+P0-1,H+P0+P1-2]="~"
1540 BS[H+P0,H+P0+P1-2]=G$  
1550 H=P0+P1+1
1560 IF POS(BS[H],"&") <> 0 THEN 1370
1570 H=1
1580 IF FS= "E" THEN 1620
1590 A9=A9+1
1600 PRINT #1;C2,C3,BS,A9
1610 C5=C5+4
1620 IF POS(BS,MI$)=0 THEN 1830
1630 IF FS= "E" THEN 1810
1640 ASSIGN "MADEFT",1,A
1650 UPDATE #1;C5
1660 ADVANCE #1;N9,Z
1670 UPDATE #1;C3
1680 REM
1690 REM SET UP MACRO NAME TABLE AND COUNTER...............
1700 REM
1710 ASSIGN "MANAMT",2,2
1720 READ #2;N6
1730 ADVANCE #2;N6,Z
1740 PRINT #2;N9+1,(N6/3)+1,N$  
1750 N6=N6+3
1760 ASSIGN "MANAMT",2,A
1770 UPDATE #2;N6
1780 ASSIGN "COMPSD",4,A
1790 UPDATE #4;C3+N5
1800 PRINT "*** PASONE --->"  
1810 CHAIN R,"MACPRO",420
1820 STOP
1830 C3=C3+1
1840 IF Z9=0 THEN 430
1850 END
**FUNCTION:**

- CHECK INPUT SOURCE STATEMENT FIND
- THE PREDEFINED MACRO DEFINITION
- EXPANSION
- DETECT THE MACRO CALL WITHIN
- THE MACRO DEFINITION
- BOOKKEEPING

**EXPANSION**

- POP UP RETURN POINTER

**BOOKKEEPING**

- Update the macro definition with the new call.

**FUNCTION:**

- Check and search for the macro call within the predefined macro definition.

**BOOKKEEPING**

- Update the macro definition with the new call.

**FUNCTION:**

- Call subroutine to handle the check and search job.
115

520 ADVANCE #1;N4+4,A
530 P1=4
540 B$[1,77]="+++"
550 P=1
560 IF END #1 THEN 580
570 READ #1;C1,C3,C$,A9
580 IF POS(C$," MEND ")=0 THEN 730
590 S[1,S[1,1]-1]=0
600 S[1,1]=S[1,1]-1
610 ASSIGN "STFILE",5,A
620 G5=G5-1
630 IF G5 <= 1 THEN 660
640 ADVANCE #5;G5-1,A
650 IF G5 > 1 THEN 690
660 IF G5 = 0 THEN 690
670 ADVANCE #5;G5-1,A
680 IF G5=1 THEN 690
690 READ #5;C$[1,72]
700 GOSUB 1560
710 IF S[1,1] <= 2 THEN 260
720 IF S[1,1] > 2 THEN 500
730 P0=POS(C$[P],"~")
740 IF P0 <> 0 THEN 860
750 B$[P1,77]=C$[P]
760 PRINT B$[1,POS(B$," ")]
770 GOSUB 1150
780 IF P=0 THEN 850
790 ASSIGN "STFILE",5,A
800 C$[1,72]=B$[4,POS(B$," ")]
810 ADVANCE #5;G5-1,A
820 UPDATE #5;B$[4,POS(B$," ")]
830 GOSUB 1560
840 IF P=1 THEN 500
850 IF P0=0 THEN 530
860 B$[P1,77]=C$[P,P+P0-2]
870 P1=P1+(P0-1)
880 CONVERT C$[P+P0,P+P0+1] TO D
890 IF D <= 9 THEN 910
900 D=D-55
910 T$=P$[(D-1)*7+1,(D-1)*7+7]
920 P5=POS(T$," ")
930 IF P5 <> 0 THEN 970
940 B$[P1,77]=T$[1,7]
950 P1=P1+7
960 IF P5=0 THEN 1000
970 B$[P1,77]=T$[1,P5-1]
980 P1=P1+(P5-1)
990 P=P+P0-1
1000 P0=POS(C$[P],",")
1010 IF P0=0 THEN 1040
1020 P=P+P0-1
1030 IF PO <> 0 THEN 580
1040 P=P+P0+3
1050 IF PO=0 THEN 580
1060 STOP
1070 REM
1080 REM *----------------------------------------------------------------------*
1090 REM * SUBROUTINE FUNCTION:                                           *
1100 REM *
1110 REM * FIND LABEL                                                   *
1120 REM * FIND MACRO NAME                                               *
1130 REM * SET UP FLAG AND STACK                                        *
1140 REM
1150 N=POS(C$," ")
1160 IF N=1 THEN 1200
1170 NO=POS(C$[N+1]," ")
1180 L$=C$(1,N-1]
1190 IF N <> 1 THEN 1210
1200 NO=POS(C$[2]," ")
1210 IF NO >= 8 THEN 1240
1220 T$[1]=C$(N+1,N+NO-1]
1230 IF NO < 8 THEN 1300
1240 T$[1]=C$(N+1,N+7]
1250 REM
1260 REM *--------------------------------------------------------*
1270 REM * IS THIS A PREDEFINED MACRO?                             *
1280 REM *--------------------------------------------------------*
1290 REM
1300 ASSIGN "MANAMT",2,A
1310 READ #2;N9
1320 IF END #2 THEN 1380
1330 READ #2;N1,N2,N$
1340 IF T$ <> N$ THEN 1320
1350 G5=G5+1
1360 F=1
1370 IF F=1 THEN 1460
1380 F=0
1390 IF F=0 THEN 1500
1400 REM
1410 REM *----------------------------------------------------------------------*
1420 REM * SET UP A STACK TO STORE RETURN POINTER                       *
1430 REM * PUSH DOWN RETURN POINTER                                    *
1440 REM *----------------------------------------------------------------------*
1450 REM
1460 S[1,S[1,1]]=N1
1470 IF S[1,1] <= 2 THEN 1490
1480 S[1,S[1,1]-1]=(A9-1)*4+1
1490 S[1,1]=S[1,1]+1
1500 RETURN
REM * SUBROUTINE FUNCTION: *  
REM * . SET UP ARGUMENT LIST ARRAY*  
REM *-----------------------------------------------*  
REM  
J=8  
B0=POS(C$," ")  
IF B0 > 1 THEN 1640  
B1=POS(C$[2]," ")  
B1=B1+B0  
P$[1,252]=" 
IF B0=1 THEN 1670  
P$[1,252]=C$[1,B0-1]  
B2=POS(C$[B0+1],",")  
B1=B1+B0  
N3=POS(C$[B1+1],",",")  
IF N3 <> 0 THEN 1710  
N3=POS(C$[B1+1]," ")  
IF N3=1 THEN 1790  
IF N3>1 THEN 1740  
T$=C$[B1+1,B1+N3-1]  
IF N3 <= 7 THEN 1750  
T$=C$[B1+1,B1+7]  
P$[J,252]=T$  
J=J+7  
B1=B1+N3  
IF Z=2 THEN 1670  
IF C$[B1+2,B1+2] <> "--" THEN 1840  
IF C$[B1+2,B1+2] = "--" THEN 1810  
READ #3;C$  
B1=1  
IF Z=2 THEN 1670  
RETURN  
PRINT "*** PASTWO --->"  
CHAIN R,"MACPRO",920  
END
REM THIS PROGRAM IS USED TO MANAGE THE SOURCE CODES.

FUNCTION:

UPDATE SOURCE PROGRAM

INSERT NEW STATEMENTS INTO SOURCE CODES

DELETE OLD SOURCE CODES

****************************************************

FILES SOURCE,*
PURGE R,"COMPSD"
PURGE R,"SOURC1"
PURGE R,"MADEFT"
PURGE R,"MANAMT"
PURGE R,"STFILE"
IMAGE 4D,3X,72A
DIM $[72]
DIM S[40]
K=0
READ #1;C0
PRINT "DO YOU WANT TO MAKE A CORRECTION? YES=1, NO=0"
INPUT N
IF N <= 0 THEN 1580
PRINT "WHAT ARE YOU GOING TO DO, SIR? PLEASE LOG IN"
PRINT "UPDATE', 'INSERT', OR 'DELETE'."
REM
REM *--------------------------------------------------*
REM * UPDATE SOURCE PROGRAM *
REM *---------------- -----------------------------------*
INPUT C$
IF POS (C$,"UPD")=0 THEN 500
PRINT "PLEASE ENTER LINE NUMBER THAT YOU ARE GOING TO CHANGE."
INPUT L
PRINT "IS THIS THE RIGHT LINE NUMBER? YES=1, NO=0"
INPUT N
IF N=0 THEN 380
ADVANCE #1;L-1,Z
PRINT "NOW! ENTER YOUR CORRECTION, PLEASE!"
INPUT C$
UPDATE #1;C$
ASSIGN "SOURCE",1,Z
GOTO 1470
IF ERROR THEN 1450
CREATEN,"BUFFER",W
IF N=0 THEN 540
119

520 PRINT "NO SPACE AVAILABLE OR OLD FILE 'BUFFER'
      HAS NOT BEEN DELETED."
530 STOP
540 REM
550 IF POS(C$,"INS") <> 0 THEN 660
560 IF POS(C$,"DEL") <> 0 THEN 1110
570 PRINT "SORRY SIR! YOUR REQUEST IS OUT OF MY CAPABILITY."
580 PURGE N,"BUFFER"
590 IF N=0 THEN 260
600 GOTO 520
610 REM
620 REM *--------------------------------------------------*
630 REM * INSERT NEW STATEMENTS INTO SOURCE CODES.     *
640 REM *--------------------------------------------------*
650 REM
660 PRINT "PLEASE ENTER THE LINE NUMBER WHICH YOU ARE GOING TO"
670 MAT S=ZER
680 M=0
690 PRINT "  INSERT NEW STATEMENT AFTER THAT LINE."
695 PRINT "***** PLEASE TYPE IN 'END.'***********"
700 FOR J=1 TO 40
710 INPUT L
720 IF L <= 0 THEN 750
730 S[J]=L
740 NEXT J
750 ASSIGN "BUFFER",2,Z
760 ASSIGN "SOURCE",1,Z
770 READ #1;C
780 PRINT "PLEASE ENTER YOUR NEW STATEMENT/STATEMENTS"
790 IF END #1 THEN 970
800 FOR I=1 TO 40
810 IF S[I] <= 0 THEN 930
820 M=M+1
830 IF X[I]-M<0 THEN 870
840 READ #1;C$
850 PRINT #2;C$
860 GOTO 820
870 INPUT C$
880 IF POS(C$,"END.") <> 0 THEN 910
890 PRINT #2;C$
900 GOTO 870
910 M=M-1
920 NEXT I
930 IF END #1 THEN 970
940 READ #1;C$
950 PRINT #2;C$
960 GOTO 940
970 PRINT #2; END
980 ASSIGN "BUFFER",2,Z
990 ASSIGN "SOURCE",1,Z
1000 PRINT #1;K
1010 IF END #2 THEN 1050
1020 READ #2;C$
1030 PRINT #1;C$
1040 GOTO 1020
1050 GOTO 1450
1060 REM
1070 REM *---------------------------------------------------------------*
1080 REM * DELETE OLD SOURCE CODES                                      *
1090 REM *---------------------------------------------------------------*
1100 REM
1110 PRINT "PLEASE ENTER THE LINE NUMBER/NUMBERS YOU ARE GOING TO DELETE."
1120 MAT S=ZER
1130 M=0
1140 ASSIGN "SOURCE",1,Z
1150 READ #1;C
1160 ASSIGN "BUFFER",2,Z
1170 FOR J=1 TO 40
1180 INPUT L
1190 IF U<=0 THEN 1220
1200 S[J]=L
1210 NEXT J
1220 REM
1230 FOR I=1 TO 40
1240 IF S[I]<=0 THEN 1320
1250 M=M+1
1260 IF END #1 THEN 1360
1270 READ #1;C$
1280 IF S[I]-M<=0 THEN 1310
1290 PRINT #2;C$
1300 GOTO 1250
1310 NEXT I
1320 IF END #1 THEN 1360
1330 READ #1;C$
1340 PRINT #2;C$
1350 GOTO 1320
1360 PRINT #2; END
1370 ASSIGN "SOURCE",1,Z
1380 PRINT #1;C
1390 ASSIGN "BUFFER",2,Z
1400 IF END #2 THEN 1440
1410 READ #2;C$
1420 PRINT #1;C$
1430 GOTO 1410
1440 PRINT #1; END
1450 ASSIGN *,2,Z
1460 PURGE Z,"BUFFER"
1470 PRINT "*** DO YOU WANT YOUR SOURCE FILE LISTED OUT? YES=1, NO=0"
1480 INPUT L
1490 IF L >= 1 THEN 1510
1500 GO TO 260
1510 ASSIGN "SOURCE",1,Z
1520 IF END #1 THEN 260
1530 READ #1;C
1540 READ #1;C$
1550 Z=Z+1
1560 PRINT USING 210;Z,C$
1570 GOTO 1540
1580 PRINT "*** CORRECT --->"
1590 CHAIN A,"MACPRO",800
1600 END
BIBLIOGRAPHY

Books


Calingaert, Peter, Assemblers, Compilers, and Program Translation, Computer Science Press, Inc., 9125 Fall River Lane, Potomac, Maryland 20854, 1979.


McQuillen, Kevin, System/360-370 Assembler Language (OS), 4905 North West Ave., Fresno, California 93705, 1975.


Articles


Publications of Learned Organizations


Yau, T. J., "The Communication Function," Loader from MOTOROLA M6800 ACIA into Memory, Department of Computer Sciences, North Texas State University, Denton, Texas 75203, 1977.

Encyclopedia Articles


Unpublished Materials