UNIT IV  MACRO PROCESSORS

1. BASIC MACRO PROCESSOR FUNCTIONS

2. MACRO DEFINITION AND EXPANSION

3. MACRO PROCESSOR ALGORITHM AND DATA STRUCTURE

4. MACHINE-INDEPENDENT MACRO PROCESSOR FEATURES

5. CONCATENATION OF MACRO PARAMETERS

6. GENERATION OF UNIQUE LABELS

7. CONDITIONAL MACRO EXPANSION

8. KEYWORD MACRO PARAMETER

9. MACRO WITHIN MACRO

10. IMPLEMENTATION EXAMPLE

11. MASM MACRO PROCESSOR

12. ANSI C MACRO LANGUAGE.
MACROPROCESSORS

A macro name is an abbreviation, which stands for some related lines of code. Macros are useful for the following purposes:

- to simplify and reduce the amount of repetitive coding
- to reduce errors caused by repetitive coding
- to make an assembly program more readable.

Figure 1. Macro expansion on a source program.

Macro expansion

Ex:

User program  Macro Definition  User program  (after macro expansion)

INITZ MACRO  header
        MOV AX, @data
        MOV DS, AX
        MOV ES, AX
          A template
          DS, AX
          (code)
        ENDM  trailer

prototype (macro name)  macro call
Using parameters in macros:

**Ex:** displaying a message. (Assume that: MES2 DB ‘Enter the data as mm/dd/yy’)

**User program Macro Definition**

```
PROMPT MACRO MESSGE ;dummy argument
   MOV AH, 09H
   LEA DX, MESSGE
   MOV AH, 09H
   INT 21H
   MOV AH, 09H
   INT 21H
   LEA DX, MES2
ENDM ;end of macro
```

**User program (after macro expansion)**

```
PROMPT MES2
```

```
MES2 DB 'Enter the data as mm/dd/yy'
```

**DATA STRUCTURE**

![Diagram of data structures](image)
ALGORITHMs

begin {macro processor}
  EXPANDING := FALSE
  while OPCODE ≠ 'END' do
    begin
      GETLINE
      PROCESSSLINE
    end {while}
  end {macro processor}

procedure PROCESSSLINE
  begin
    search NAMTAB for OPCODE
    if found then
      EXPAND
    else if OPCODE = 'MACRO' then
      DEFINE
    else write source line to expanded file
  end {PROCESSSLINE}

procedure DEFINE
  begin
    enter macro name into NAMTAB
    enter macro prototype into DEFTAB
    LEVEL := 1
    while LEVEL > 0 do
      begin
        GETLINE
        if this is not a comment line then
          begin
            substitute positional notation for parameters
            enter line into DEFTAB
            if OPCODE = 'MACRO' then
              LEVEL := LEVEL + 1
            else if OPCODE = 'MEND' then
              LEVEL := LEVEL - 1
          end {if not comment}
        end {while}
    store in NAMTAB pointers to beginning and end of definition
  end {DEFINE}
One-Pass Macro Processor

- Prerequisite
  - every macro must be defined before it is called
- Sub-procedures
  - macro definition: DEFINE
  - macro invocation: EXPAND

Data structure Global variable

- DEFTAB
- NAMTAB
- ARGTAB
Nested Macro:

- Macro definition within macros
  - process macro definition during expansion time

1. **MACROS** MACRO {Defines SIC standard version macros}
2. **RDBUFF** MACRO &INDEV, &BUFADR, &RECLTH
   . . {SIC standard version}
3. **MEND** {End of RDBUFF}
4. **WRBUFF** MACRO &OUTDEV, &BUFADR, &RECLTH
   . . {SIC standard version}
5. **MEND** {End of WRBUFF}
6. **MEND** {End of MACROS}
Nested Macro Definition

- Sub-procedures
  - macro definition: DEFINE
  - macro invocation: EXPAND
- EXPAND may invoke DEFINE when encounter macro definition

```
procedure DEFINE
begin
  enter macro name into NAMTAB
  enter macro prototype into DEFTAB
  LEVEL := 1
  while LEVEL > 0 do
    begin
      GENLINE
      if this is not a comment line then
        begin
          substitute positional notation for parameters
          enter line into DEFTAB
          if OPCODE = 'MACRO' then
            LEVEL := LEVEL + 1
          else if OPCODE = 'MEND' then
            LEVEL := LEVEL - 1
          end {il not comment}
        end {while}
      end {GENLINE}
      store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}
```
procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition (prototype) from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end ;while)
        EXPANDING := FALSE
    end (EXPAND)

Procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end ;if)
        else
            read next line from input file
        end (GETLINE)

Figure 4.5 (cont’d)

---

1-Pass Macro Processor

![Diagram of 1-Pass Macro Processor]
Comparison of single & 2-Pass design

- **Single pass**
  - every macro must be defined before it is called
  - one-pass processor can alternate between macro definition and macro expansion
  - nested macro definitions may be allowed but nested calls are not

- **Two pass algorithm**
  - Pass 1: Recognize macro definitions
  - Pass 2: Recognize macro calls
  - nested macro definitions are not allowed

Concatenation of parameter

(a)

1. SUM MACRO &ID
2. LDA X&ID→1
3. ADD X&ID→2
4. ADD X&ID→3
5. STA X&ID→S
6. MEND
- Pre-concatenation
  » LDA X&ID1
- Post-concatenation
  » LDA X&ID→1

**Generation of unique labels**

- Example
  » JEQ *-3
  » inconvenient, error-prone, difficult to read
- Example Figure 4.7
  - $LOOP TD =X’&INDEV’
  » 1st call:
    - $AALOOP TD =X’F1’
  » 2nd call:
    - $ABLOOP TD =X’F1’

```
25 RDBUFF MACRO &INDEV, &BUFADR, &RECLTH
30 CLEAR X CLEAR LOOP COUNTER
35 CLEAR A
40 CLEAR S
45 +LOT #4096 SET MAXIMUM RECORD LENGTH
50 $LOOP TD =X’&INDEV’ TEST INPUT DEVICE
55 JEQ $LOOP LOOP UNTIL READY
60 RD =X’&INDEV’ READ CHARACTER INTO REG A
65 COMPR A, S TEST FOR END OF RECORD
70 JEQ $EXIT EXIT LOOP IF EOR
75 STCH &BUFADR, X STORE CHARACTER IN BUFFER
80 TIXR T LOOP UNLESS MAXIMUM LENGTH
85 JLT $LOOP HAS BEEN REACHED
90 $EXIT STX &RECLTH SAVE RECORD LENGTH
95 MEND
```

(a)

**RDBUFF** F1, BUFFER, LENGTH

```
30 CLEAR X CLEAR LOOP COUNTER
35 CLEAR A
40 CLEAR S
45 +LOT #4096 SET MAXIMUM RECORD LENGTH
50 $AALOOP TD =X’F1’ TEST INPUT DEVICE
55 JEQ $AALOOP LOOP UNTIL READY
60 RD =X’F1’ READ CHARACTER INTO REG A
65 COMPR A, S TEST FOR END OF RECORD
70 JEQ $AEXIT EXIT LOOP IF EOR
75 STCH BUFFER, X STORE CHARACTER IN BUFFER
80 TIXR T LOOP UNLESS MAXIMUM LENGTH
85 JLT $AALOOP HAS BEEN REACHED
90 $AEXIT STX LENGTH SAVE RECORD LENGTH
```

(b)
Conditional Macro Expansion

- Macro-time conditional statements
  - Example: Figure 4.8
  - *IF-ELSE-ENDIF*

- Macro-time variables
  - any symbol that begins with the character & and that is not a macro parameter
  - macro-time variables are initialized to 0
  - macro-time variables can be changed with their values using SET
    - &EORCK SET 1

- Macro-time looping statement
  - Example: Figure 4.9
  - WHILE-ENDW

- Macro processor function
  - %NITEMS: THE NUMBER OF MEMBERS IN AN ARGUMENT LIST

Only part of the macro is copied out into the code. Which part is copied out will be under the control of the parameters in the macro call.

CONMB (condition) branch address

**Ex:** Line

<table>
<thead>
<tr>
<th>no.</th>
<th>Assembly instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>CONMB (&amp;C&gt;2), 15</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

If condition is true, skip the code up to line 15.
If condition is false, expansion continues from line 9.
Design for a simple Macroassembler

Figure 2. A macroprocessor front end for an assembler
write it to the output file

LP = LP + 1
the statement

LP = LP + 1

LP = line number in

Y
more lines
N
return

Figure 3. Macro Expander Routine

Ex:

Source program

Mac

ro definition

…
ALPHA A, 2, C
…

Parameter table

<table>
<thead>
<tr>
<th>Dummy parameter</th>
<th>Real parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG1</td>
<td>A</td>
</tr>
<tr>
<td>ARG2</td>
<td>2</td>
</tr>
<tr>
<td>ARG3</td>
<td>C</td>
</tr>
</tbody>
</table>
Keyword Macro Parameters

- Positional parameter: parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement
- Keyword parameters: each argument value is written with a keyword that named the corresponding parameter
- Each parameter name is followed by an equal sign, which identifies a keyword parameter
- The parameter is assumed to have the default value if its name does not appear in the macro invocation statement

```assembly
25  REDUFF  MACRO  &INDEV=F1, &BUFFADR=, &RECLTH=, &EOR=04, &MAXLTH=4096
26    IF      (&EOR NE ' ') 
27    &EOCK   SET  1
28   ENDIF
30   CLEAR  X          CLEAR LOOP COUNTER
35   CLEAR  A
38   IF      (&EOCK EQ 1)
40   LDCH    =X'&EOR'    SET EOR CHARACTER
42   RMO     A, S
43   ENDF
47   +LIT    #&MAXLTH    SET MAXIMUM RECORD LENGTH
50  $LOOP  TD     =X'&INDEV'   TEST INPUT DEVICE
55   JBQ    $LOOP     LOOP UNTIL READY
60   RD      =X'&INDEV'   READ CHARACTER INTO REG A
63   IF      (&EOCK EQ 1)
65   COMP    A, S     TEST FOR END OF RECORD
70   JEQ    $EXIT      EXIT LOOP IF EOR
73   ENDF
75   STCH    &BUFFADR, X STORE CHARACTER IN BUFFER
80   TIXR    T         LOOP UNLESS MAXIMUM LENGTH
85   JLT    $LOOP      HAS BEEN REACHED
90  $EXIT  STX    &RECLTH   SAVE RECORD LENGTH
95  EEND
```

Macro calls within Macros

- Use a stack to keep the order of the macro calls
- Create a parameter table at each macro call
- If a dummy parameter appears in the real parameter field of a called macro, search the parameter table of the calling macro and replace it with the appropriate real parameter.
Find the appropriate macro definition

Build a parameter table to relate dummy and real parameters

Push LP onto the stack

LP = first line in the template

Examine the line

? (instruction, conditional, macro call, macro end, assembly call)

substitute real parameters appropriate
evaluate the Boolean expression

Find the macro definition

Pop from stack

and set LP and current param
write it to the output file table pointer

LP = LP + 1

LP = LP + 1

LP = line number in the statement

Build a parameter table

Push LP and the pointer to the last parameter table onto the stack

LP = first line in the template
Figure 4. Macro Expander Routine that allows macro calls within macros

1. MACROS MACRO {Defines SIC standard version macros}
2. RDBUFF MACRO &INDEV,&BUFADR,&RECLTH
   .
   . {SIC standard version}
   .
3. MEND {End of RDBUFF}
4. WRBUFF MACRO &OUTDEV,&BUFADR,&RECLTH
   .
   . {SIC standard version}
   .
5. MEND {End of WRBUFF}
6. MEND {End of MACROS}
ARGTAB

1 MACROX MACRO {Defines SIC/XE macros}
2 RDBUFF MACRO &INDEV,&BUFADR,&RECLTH
   .
   . {SIC/XE version}
   .
3 MEND {End of RDBUFF}
4 WRBUFF MACRO &OUTDEV,&BUFADR,&RECLTH
   .
   . {SIC/XE version}
   .
5 MEND {End of WRBUFF}
   .
   .
6 MEND {End of MACROX}

ARGTAB

DEFTAB NAMTAB

MACRO Definition

DEFINE

GETLINE

EXPAND

PROCESSLINE

Macro Invocation

ARGTAB
I-PASS macroprocessor

Allowing Nested Macro Invocation
Macro Assembler

Macro Processor Design Options

- Recursive Macro expression
- General-Purpose Macro Processors
- Macro Processing within Language Translators

Recursive Macro Expansion

- Processor is being written in a programming language that allows recursive calls
- The compiler would be sure that previous value of any variables declared within a procedure were saved when that procedure was called recursively
- If would take care of other details involving return from the procedure

General Purpose Macro Processor

- Advantages of general-purpose macro processors:
  - The programmer does not need to learn about a different macro facility for each compiler or assembler language—the time and expense involved in training are eliminated
  - The costs involved in producing a general-purpose macro processor are somewhat greater than those for developing a language-specific processor
However, this expense does not need to be repeated for each language; the result is substantial overall saving in software development cost

- user to define the specific set of rules to be followed
- Comments should usually be ignored by a macro processor, however, each programming language has its own methods for identifying comments
- Each programming language has different facilities for grouping terms, expressions, or statements—a general-purpose macro processor needs to taking these grouping into account
- Languages differ substantially in their restrictions on the length of identifiers and the rules for the formation of constants
- Programming languages have different basic statement forms—syntax used for macro definitions and macro invocation statements

**ELENA**

```
Software: Practice and Experience
```

**Macro definition**

```
header:
  - a sequence of keywords and parameter markers (%)
  - at least one of the first two tokens in a macro header must be a keyword, not a parameter marker

body:
  - the character & identifies a local label
  - macro time instruction (.SET, .IF .JUMP, .E)
  - macro time variables or labels (.)
```

**Macro invocation**

```
There is no single token that constitutes the macro “name”

- Constructing an index of all macro headers according to the keywords in the first two tokens of the header

Example
  - DEFINITION:
    - ADD %1 TO %2
    - ADD %1 TO THE FIRST ELEMENT OF %2
  - INVOCATION:
    - DISPLAY TABLE
```

**Macro Processing within Language Translators**

- The macro processor reads the source statements and performs all of its functions, the output lines are passed to the language translator as they are generated
- The macro processor operates as a sort of input routine for the assembler or compiler
The line-by-line approach avoids making an extra pass over the source program, so it can be more efficient than using a macro preprocessor.

Some of the data structures required by the macro processor and the language translator can be combined.

A line-by-line macro processor also makes it easier to give diagnostic messages that are related to the source statement containing the error.

An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

An integrated macro processor can support macro instructions that depend upon the context in which they occur.

Line-by-line macro processors must be specially designed and written to work with a particular implementation of an assembler or compiler, which results in a more expensive piece of software.

The assembler or compiler will be considerably larger and more complex than it would be.

The additional complexity will add to the overhead of language translation.

**MASM**

The macro processor of MASM is integrated with Pass 1 of the assembler.

MASM generates the unique names of local labels in the form ??n, where n is a hexadecimal number in the range 0000 to FFFF.

.ERR: signals to MASM that an error has been detected.

.EXITM: directs MASM to terminate the expansion of the macro.

&: is a concatenation operator.

;; is a macro comment, serves only as documentation for the macro definition.

; is an ordinary assembler language comment, included as part of the macro expansion.

IRP: sets the macro-time variable to a sequence of values specified in <...>.

The statements between the TRP and the matching ENDM are generated once for each value of the variable.

```assembly
1  ABSDFR MACRO OP1,OP2,SIZE
2       LOCAL EXIT
3       IFNBER <SIZE>,<S> ; ; IF SIZE IS NOT BLANK
4       IFDFIF <SIZE>,<S> ; ; THEN IT MUST BE E
5 ; ERROR -- SIZE MUST BE E OR BLANK
6       .ERR
7       EXITM
8       ENDFIF ; ; END OF IFDFIF
9       ENDFER ; ; END OF IFNBER
10      MOV SIZE&AX,OP1 ; COMPUTE ABSOLUTE DIFFERENCE
11      SUB SIZE&AX,OP2 ; ; SUBTRACT OP2 FROM OP1
12      JNS EXIT ; ; EXIT IF RESULT GE 0
13      NEG SIZE&AX ; ; OTHERWISE CHANGE SIGN
14      EXIT:
15      ENDFR
```

Numerical data is not present in the image.
ABSDIF J, K

\[ \downarrow \]

MOV AX, J ; COMPUTE ABSOLUTE DIFFERENCE
SUB AX, K
JNS ??0000
NEG AX

??0000:

ABSDIF M, N, E

\[ \downarrow \]

MOV EAX, M ; COMPUTE ABSOLUTE DIFFERENCE
SUB EAX, N
JNS ??0001
NB3 EAX

??0001:

ABSDIF P, Q, X

\[ \downarrow \]

; ERROR -- SIZE MUST BE E OR BLANK

<table>
<thead>
<tr>
<th>1</th>
<th>NODE</th>
<th>MACRO</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>IRP</td>
<td>S, '&lt;LEFT', 'DATA', 'RIGHT'&gt;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NAME6S</td>
<td>DW</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ENDM</td>
<td>; END OF IRP</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ENDM</td>
<td>; END OF MACRO</td>
<td></td>
</tr>
</tbody>
</table>

(a)

NODE X

\[ \downarrow \]

XLEFT DW 0
XDATA DW 0
XRIGHT DW 0

(b)

**Figure 4.13** Example of MASM iteration statement.
ANSI C Macro Language

- In the ANSI C language, definitions and invocations of macros are handled by a preprocessor, which is generally not integrated with the rest of the compiler.
- `#define ABSDIFF(X,Y)  X>Y ? X-Y : Y-X`
- `#define DISPLAY(EXPR) printf(#EXPR"=%d\n",EXPR)`
- Macro in ANSI C may contain definitions or invocations of other macros.
- `DISPLAY(ABSDIFF(3,8))`
- `ABSDIFF(3,8)=5`
- The ANSI C preprocessor also provides conditional compilation statements.
- `#ifndef BUFFER_SIZE`
  - `#define BUFFER_SIZE 1024`
  - `#endif`