MACRO 88 Assembler User's Guide

PolyMorphic Systems

460 Ward Drive Santa Barbara California 93111 (805) 967-2351

The MACRO-88 Assembler described in this manual was created by Roger L. Deran. The assembler is provided as a standard part of the System 88 System Disk, which is available in four versions: PolyMorphic Systems part no. 000386 (which provides eight digits of precision in software), 000389 (fourteen digits in software), 000502 (which provides eight digits of precision in hardware), and 000503 (fourteen digits in hardware). This manual is PolyMorphic Systems part no. 810146.

Section 1

THE MACRO-88 ASSEMBLER: INTRODUCTION

MACRO-88 is an assembler that runs on the PolyMorphic Systems System 88 line of microcomputers, which use the 8080A microprocessor and flexible-disk drives. The System 88 computers have their own characteristics, inherited in part from the disk drives, the architecture of the microprocessor, the organization of the main bus-- the S-100-- used in the system, and the structure of the System 88 disk operating system. The characteristics of any system help to determine the character of any large program written to run on it, and this assembler is no exception. This section discusses not only the actual assembler, but also some reasons why it is the way it is.

1.1 WHAT THE ASSEMBLER DOES

The purpose of an assembler is to simplify the creation of object-code programs. An object-code program is a sequence of data bytes which can serve as instructions for the CPU directly. No matter what computer language a program is first written in, the processor can understand only object code (machine language), so an object-code program is the final step of any translation process that is supposed to produce runnable programs. An assembler translates assembly-language statements into object code. The assembler is a special kind of translator in that there is a one-to-one correspondence between the assembly-language statements of the program you write and the object-code statements that the assembler produces. Thus when the programmer writes an assembly-language program, he knows exactly what machine instructions will go to the processor, and in exactly what order-- and exactly where each instruction will be in memory when the program is loaded.

This control over the final object code gives the assembly-language programmer great power over the computer's operations. Anything that can be translated into an object-code program by any language translator can be written in assembly language, although with some (perhaps vast) increase in the length of the program text. The special power of assembly language that makes it the almost universal choice of systems programmers is that it enables them to write very efficient programs, very small, "optimized" object code, or some series of machine operations that can only result from a very specific sequence of instructions. In fact, the whole purpose of an assembler is to produce a known sequence of instructions in memory.

Assembly languages-- including MACRO-88-- also have special facilities to overcome some of the more painful inconveniences of writing a program an instruction at a time. These facilities

include labels, macros, assembly-time mathematics, and several other things, all of which will be discussed later.

1.2 FILES USED BY THE ASSEMBLER

For now, we will concern ourselves with the problem of handling the text-- the assembly-language program that you write-- and the object code-- the machine-language translation that the assembler produces-- of the assembly program.

System 88 has special abilities for handling object-code programs as files, which it automatically identifies by putting the extension .GO on their disk file names.

Files containing the text of a program-- the human-readable assembly-language form of the program-- are stored with .TX extensions. You create and correct them by using the system text editor. The text file for an assembly-language program might be only a few sectors in length, or so long it has to reside on more than one disk! It can be too large to fit in the computer's main memory all at once, so the assembler must resort to devious means to translate the program without being able to examine more than a small section of it at a time. Note that the text for a program may be very long, but the object code will usually be far shorter!

1.3 THE ASSEMBLY PROCESS

During assembly of a program, the assembler reads the text version of the program from the disk and writes the object-code version of the program onto a disk (not necessarily the same disk). At the same time, the assembler can generate a "listing," which is a printout of the statements the assembler read and the numerical values of the object code it generated for each statement-- all arranged in an easy-to-read form.

1.3.1 The Two Passes

The text of the program is actually read twice, once in "Pass one" and then again in "Pass two." The reason it takes two passes is this: Before the assembler can actually assemble the assembly-language program-- translate it into object code-- it has to build up a "symbol table." The symbol table is a list of the "symbols" used in the assembly program with the associated "values" of each of these symbols stored beside the symbol. The problem is that frequently a symbol's value is not defined in a program until some time after it is first used or "referenced." Thus during the first pass, the assembler collects all the definitions of the labels throughout the program, so as to use them on the second pass, when it actually writes out the object code and prints the listing. The first pass, then, is a quick scan of the program, during which the actual meanings of

the statements are largely ignored. For the second pass, the input file is "rewound" and the program re-read; this time the assembler puts in the values of the symbols and generates the object code.

The assembler handles the two passes over the text automatically. You don't need to rewind the input file or do anything yourself between the two passes. Once the assembler knows what file is to be assembled, it can proceed on its own until it finishes.

1.3.2 Using Multiple Source Files

The disk does relieve you of most of the hard work involved in an assembly, but there are still limits, primarily those imposed by the size of the disk storage. A very large program can be too large for one disk, so you must somehow change disks in the middle of assembly. There is a special facility in MACRO-88 for assembling programs that are distributed among multiple files or even multiple disks.

To handle multiple input files, the assembler asks you for the name of a continuation file after it finishes reading a file. But there still must be a way to have the assembler automatically rewind and go into pass two if the entire source text is in one file. Here is how these two requirements are both fulfilled at once. All assembly-language programs to be translated by the MACRO-88 must end with an END statement. If the assembler runs out of input text before it finds the mandatory END statement, it assumes that there is a continuation file, where the rest of the input text can be found. It will ask you to specify the name of the continuation file. You take this opportunity to exchange disks in the drive if necessary, and then type in the name of the continuation file. When the END statement is finally found, you replace the original disk and re-specify the original source file. If the END statement is found on the first source file, the assembler will immediately rewind and start pass two automatically.

1.4 THE ASSEMBLY LISTING

You will probably want to make a listing of your program while it is being assembled. The assembly listing shows the original program, as well as the translation of it. It serves as a base for your bug-hunting efforts. MACRO-88 writes out listings through the system's standard "printer wormhole" port to a line printer of some kind. The listing also includes error summaries, symbol table printouts, the number of errors assembled, and so on.

A section of a typical assembly listing is shown below. This is the portion of the listing that shows the object code and the original text or source code. Note that each kind of information item falls into its own "field."

```
3367  3AD34C    SCNPT   LDA     BSAVE   ;Is there any more t
336A  B7                ORA     A
336B  C8                RZ
336C  3D                DCR     A       ;Yes - count it.
336D  32D34C            STA     BSAVE
3370  7E                CPI     TAB     ;If TAB, go scan for a
```

1.5 MACROS

The MACRO-88 Assembler includes "macros" or ways to name a block of program text so that it can be used later simply by referring to it by name. Doing this involves the use of two pseudo-ops, MACRO and ENDM, which begin and end the defined block of text. A block of text so defined by MACRO and ENDM can then be used by using its name in a statement as if it were an instruction.

Macros are explained at length in Section 2.3.2.4.

Section 2

ASSEMBLY-LANGUAGE PROGRAMMING

This section discusses the form and meaning of the statements that MACRO-88 accepts and translates. It assumes that you are already familiar with the function of the 8080 instructions themselves and only need to know how to make the assembler generate the desired instructions.

2.1 ASSEMBLY-LANGUAGE STATEMENTS

An assembly-language program is written as a sequence of statements corresponding to the sequence of instructions the programmer wants to have in memory when the object-code program is loaded in to be run. MACRO-88 translates the statements one by one and writes the resulting object-code data to the object-code file byte by byte. Thus when the object-code file is loaded into memory, the sequence of intructions is the same as the sequence of statements in the source text.

2.1.1 Statement Fields

Statements in assembly language all have the same basic structure. Each program line is a made up of several "fields" (sub-parts oF the line of text) used for certain dedicated functions. Thus when you read a line of assembly-language program, the item you encounter first-- the one over on the left-- is likely to be a label, since the label field is the first field. It's important to note that the field is not the same as the thing that's in the field; the label field may not have anything in it. A field is like a box with a tag on it; anything inside the "label" box is a label, but the "label" box can be empty.

The three main fields are the label, opcode, and operand fields.

2.1.2 The Label Field

A label is one of several different kinds of symbols used in assembly language. Symbols are created in a very strict way from letters, digits, and special characters (exact specifications for symbols are given in a later section). The symbol used in the label field is called a "label symbol" to distinguish it from symbols used in other fields.

The label field is the first field on the line. It can contain a label, or it can be blank. The label field is considered blank if there is no symbol at the left end of the statement line. If there are any spaces or TABs before the first symbol, or in fact if there are ANY characters that are not allowed as part of a symbol at the front of the statement, the label field is considered blank.

After the label, a colon : may be used to make the program compatible with other assemblers that require the colon after labels. The colon is ignored completely by MACRO-88.

2.1.3 The Opcode Field

The opcode field is the next field. It always has a space or TAB in front of it, either just after the label symbol in the label field, or (if the label field is blank) as the very first character of the statement. Thus we say that the opcode field is "delimited" by a space or TAB; the label field is ended and the opcode field begun by one of these characters. As many spaces or TABs as desired may precede the opcode field. The opcode field may also be blank. The opcode field is considered blank if there is no text at all after the label field or if there is only a semicolon followed by comment text.

Opcodes are the "words" or vocabulary of assembly language. An opcode-- that is, the symbol in the opcode field-- can be either an instruction mnemonic, a pseudo-opcode (also "pseudo-op" or "popcode" or even "pop"), or a macro reference. An instruction mnemonic found in the opcode field gets translated into the binary bytes of a machine code instruction, which are written to the output file. Pseudo-opcodes, on the other hand, normally produce no bytes for the output file, but rather serve as control signals to the assembler. Pseudo-opcodes are known also as assembler directives, assembly control codes, assembler commands, etc. Macro references are symbols which the user gives a special meaning.

The meanings of the label symbols and the operands (see 2.1.4, Operand Field) are determined by the symbol in the opcode field. If the opcode is an instruction mnemonic, the label symbol is interpreted as a statement label and can be used elsewhere in the program to refer to the address of the instruction it is attached to. The operand field determines the various subparts of the object code instruction, such as the address field of a branch instruction.

If the opcode symbol is a pseudo-op or macro reference, the meanings of the symbols in the label and operand fields are determined by the particular opcode symbol used. The meaning of each of the fields is defined for each pseudo-op elsewhere in this manual.

Macro symbols can cause the label and operand fields to be interpreted in a multitude of ways, depending on the user's definition of the meaning of the macro symbol.

```
 Label   Opcode
 field   field
-------------------------------------------------------------
(We use a label symbol HELLO and an opcode symbol NOP)

HELLO    NOP      ;Comment text after the semicolon.
         NOP      ;Comment text can go after the opcode,
HELLO             ;or after the label. (Two TABs used here.)
         ; This is comment text.  There is no label or opcode.
;This is also comment text.  There is no label or opcode.
HELLO    ; This statement has only a label, but no opcode.
         NOP; Only an opcode, with no spaces before the ; .
 NOP;A space before the opcode, no space before the comment.
HELLO:   NOP      ;The optional colon after the label. This is
                  ;the exact same line as the top line above.
```

2.1.4 The Operand Field

The third field is the operand field. It starts after the opcode field and is separated from it by a space or TAB. It is subdivided by commas into separate parts called operands. The symbol in the opcode field determines the meaning of the symbol in the operand field and the number of operands needed. The NOP instruction requires no operands at all, so the operand field is left empty. A branch instruction might require one operand to determine the machine address to go to. The DB pseudo-op will accept any number of operands. A macro reference could accept, say, three operands if the first operand were included, or none at all if the first operand were not included (again, it depends on the user's definition of the macro).

2.1.5 Comments

A comment is a string of text started by a semicolon and ended by the end of the line. The semicolon may be used anywhere in any field that is blank. The comment field acts like a blank field.

2.2 OPERANDS

An operand is a modifier for the opcode. Each opcode requires different operands and uses them in different ways. Some operands may be symbol names, others integer-valued expressions. Some contain string expressions, some not. The next few paragraphs discuss the different types of operands and their uses.

2.2.1 Expressions

An expression that evaluates to an integer is the most common kind of operand. These operands are used for describing the immediate data fields of instructions. For example, the MVI opcode expects an eight-bit value to be specified in the operand field for use as the second byte of the instruction:

```
34B6  3E06              MVI     A,6
```

In the above example of an assembly-language program line, the MVI opcode has two expected operands. The first one, A, is the register specification, while the second one, 6, is the specification of the value of the immediate byte after the opcode (see object code field in listing). They are separated by a comma, as usual (and no space, although a space between operands is permitted).

The next few paragraphs consider the construction of expressions. They may be very complex, or just a number or a symbol name, as shown in the example.

2.2.1.1 Numbers

A number is a sequence of letters and digits that starts with a digit. It can have no imbedded blanks, TABs, non-letters, or non-digits. The last character of the number can be a letter indicating the base to be used in interpreting the rest of the number. This last character is called the base specifier, and is optional. If it is left off, the number is assumed to be in base 10 (decimal). Some valid numbers are shown below:

123	Value is 123 decimal.
123D	Same as above, with base specifier explicitly indicated.
200H	A hexadecimal number (base 16) whose main part is 200.
0AC6FH	A hex number showing the use of the letters A-F as digits 10-15. Note that a number must start with a digit, so we had to put a zero in front of this one.
377Q	Octal 377.
1101101B	Binary 1101101 (Hex 6D).

As shown by the 0AC6FH hexadecimal example above, a number can always be made to obey the rule that the first character be a digit by simply putting a zero in front. The initial digit is important, because a number with an initial letter would be taken to be a symbol name.

2.2.1.2 Symbols

Another kind of expression is a "symbol reference," which is just a sequence of characters that conforms to the rules for a valid symbol name. A symbol name is a sequence of letters, digits, or the characters ? and @, with a first character that

is not a digit (to distinguish the symbol name from a number). Symbol names may be any length. Here are some valid symbol names:

```
HELLO
Label
Start
BiNgO
VeryLongSymbolName
Hello124
?questionmarks?
@atsigns@andmoreatsigns@
@@@?@?134
```

And here are some invalid symbol names:

```
1way            starts with a digit
go away         imbedded blank
new$try         illegal imbedded character
```

The value of the expression represented by just a single symbol reference is the value that is stored in the symbol table in association with that symbol name. If a symbol is not defined anywhere in a program and yet is referred to in an expression, a value of zero is used and an error message is printed:

```
** ERROR 1101:  Undefined symbol reference **
```

If a symbol has been defined twice in one program, the value of the symbol is undefined. The assembler actually uses the first value assigned to the symbol for the value in the expression, and the message

```
**ERROR 1102: Doubly defined symbol**
```

is printed.

2.2.1.3 Dollar Sign

The dollar sign $ is a valid expression whose value is the load address of the instruction being assembled at the point where the dollar sign is used. Thus if we assemble a

```
JMP     $
```

instruction, the result will be an opcode for a jump with an immediate address that is the same as the address of the jump instruction itself. When executed, the computer will loop forever. The dollar sign is valid wherever an expression is valid.

2.2.1.4 Short strings

An expression may have a value which is an ASCII character constant. In this case, the character that is to be used can be

expressed in single quotes, or apostrophes:

```
'A'             41 Hex
'1'             6C Hex
'HI'            4849 Hex (Note 'H' is most significant.)
'BINGO'         4249 Hex ('BI' used, rest lost.)
```

Since the value of an expression is limited to 16 bits, there cannot be more than two characters in a short string. Usually, a short string is used where only the least significant eight bits of the expression value will be needed, as in the MVI instruction:

```
0604 3E25       MVI     A,'%'   ;Load accum with a percent.
```

If the string is too long for the instruction, a special error message will be printed appropriate to that instruction. However, if the string is too long for an expression altogether, we get:

```
** ERROR 0203: String is too long for a double-word **
```

There is a difficulty in expressing the single quote sign itself within a string, since it is used as the delimiter for the end of the string. If it is used in the middle of a string, the assembler would normally think it has found the end of the string. To circumvent this problem, a special facility has been added. If the end of a string is found, but another string follows immediately, then the two strings are considered as one string with a single quote mark in between. Examples:

```
'I don''t know'         (value is I don't know)
''''                    (value is just single quote)
''''''                  (value is two single quotes)
''                      (value is null string)
```

Note that everything we have said about expression lists in short strings also applies to long strings.

2.2.1.5 Functions

Functions are normally used in macro programming to do things to parameters passed in to a macro. They have 16-bit values just like any other expression primary, but operate in some special way on the things between the two following brackets. The brackets delimit the boundaries of the text "looked at" by a function. The left bracket immediately following the function name is actually part of the function name, so no spaces should be used in between.

2.2.1.5.1 LEN[]

The LEN[] function evaluates the contents of its bracketed text area as if they were an expression list to be submitted to the DB pop. Instead of assembling the data bytes, however, LEN

merely counts them and returns their total number. In other words, if the same string given to LEN[] is substituted into a DB pop in the operand field, the outputted object code will occupy the number of bytes in memory that LEN[] returns as its value.

As an example, consider the following:

```
        DB      LEN['Hi there!']+2, 'Hi there!',CR,LF
```

This statement will assemble first a byte with value 11 (decimal), then 9 bytes of ASCII character information followed by 2 symbolically defined constants.

2.2.1.5.2 NULL[]

In a macro, it is possible to pass a parameter which contains no characters at all. This happens when a position in the operand field of the macro invocation is left empty. The parameter is then considered non-existent or "null." The NULL[] function allows a macro to take appropriate action when an optional parameter has been omitted. It returns a logical value of TRUE (-1 or 0FFFF hex) when there is a null parameter between the brackets and FALSE (0) when any number of valid parameters can be found there.

Example:

```
        DW      NULL['I am not null'],NULL[]
```

This assembles a 0000H followed by a 0FFFH.

Example of use inside a macro:

```
MAYBE   MACRO
        IF      NOT NULL[#1]    ;If there's a parameter #1,
        DW      #1              ;then assemble it.
        ENDIF
        ENDM
```

Thus the invocation

```
        MAYBE   5
```

expands into

```
        DW      5
```

But with a null operand field, MAYBE assembles nothing.

2.2.1.6 Mathematical expressions

It was said above, in the discussion on numbers, symbols, the dollar sign, and short strings, that each of these things is a

form of expression. These are "primary" expressions. A generalized expression may be much more complex than a single primary expression because of the ability to combine the primaries into mathematical formulae.

For example, the following is a valid expression:

```
(5 * (HELLO SHR 3) AND X/2 + (COUNT MOD 256 -3))
```

Parentheses can be used to combine operations in any way. Without parentheses, evaluation proceeds with calculation of the "higher precedence" parts of the expression first. Multiplication of two numbers, for example, would be calculated before the addition of that product to another number, even if the addition is written first. This is because the multiplication operator has the higher precedence:

```
5+4*6          value is 29 (not 54)
```

If two succesive operations have the same precedence, as do multiplication * and division / for example, then they will be evaluated from left to right. The relational operators are an exception: they cannot be grouped together, as in:

```
BINGO > BINGO2 > BINGO3
```

which is illegal.

Operators available for expressions are arranged on the next page in order of increasing precedence (the operators nearest the bottom of the table are calculated first). Operators with the same precedence are grouped together.

Operator	No. of Arguments	Function
OR	2	Logical bitwise 16-bit OR.
XOR	2	" " " " exclusive OR.
AND	2	" " " " AND.
NOT	1	" " " " complement.
=	2	Relational equivalence.
>	2	" greater than.
<	2	" less than.
>=	2	" greater than or equal to.
<=	2	" less than or equal to.
!=	2	" inequality.
+	2	16 bit addition.
-	2	" " subtract (two's complement).
*	2	16 bit multiply (only low 16 bits of result are available).
/	2	16 bit positive divide.
MOD	2	Modulus (remainder of 16 bit division of first argument by second).
SHL	2	Shift Left. First argument shifted left the number of binary places given in second argument.
SHR	2	Shift Right. Similar to SHL.
-	1	Unary minus. Single argument is negated--two's complement. Example: -1=0FFFF Hex

The order of calculation may be fixed by using parentheses around sub-expressions that are to be calculated first. No more than five pairs of parentheses can be nested at the deepest level. If more are nested, a zero value is used for the expression, and the message:

```
** ERROR 0801: Expression nested too deep - sorry **
```

will be printed. If the parentheses are not paired properly, we get:

```
** ERROR 0201: Expression has unbalanced parentheses **
```

2.2.1.6.1 Arithmetic Operators.

Arithmetic operators all operate on 16 bits at once. Some operate as if their operands were two's complement binary integers, others as if they were positive binary integers; some operators work just as well when their operands are considered to be either one. The arithmetic operations are: +, -, (unary)-, *, /, MOD, SHL, and SHR.

Addition works on positive numbers, but since two's complement numbers can be added as if they were positive, it works just as well for them. If the result of addition is greater than 65535, then it cannot be expressed in 16 bits, so this error results:

```
** ERROR 0E01: Number has value > 65535 **
```

Since a two's complement number cannot exceed +32767 or -32768 without having a non-zero bit 17, the error detection works properly for both number systems.

Subtraction and unary minus (-X) both assume their operands are two's complement numbers. Subtraction is exactly equivalent to negation using unary minus followed by addition using the addition described above. Thus the overflow error message works according to the add. Unary minus is exactly equivalent to a logical complementing of each bit followed by an incrementing by one using the add.

```
X-Y => X+-Y              ;Definition of subtraction.
-Y  => (NOT Y) + 1       ;Definition of unary minus.
```

Any number of unary minuses may precede any kind of valid primary expression to change its sign. Each occurrence of a minus sign negates the value once.

Multiplication is a positive multiplication of the two 16-bit operands to form a 16-bit product. If the result is larger than 65535, the bottom 16 bits are returned as the product. This allows numbers which are to be regarded as negative two's complement numbers to be multiplied and to have their signs effectively exclusive-ORed. In other words, -1*-1=1, -1*1=-1, 1*-1=1, and 1*1=1.

For positive numbers (non-two's complement), however, a result larger than 65535 will appear to be within range using this scheme. It is the programmer's responsibility to insure that the result can never exceed the 16-bit representability limit or to insure that the overflow can have no dangerous effects.

Division and modulus (X MOD Y) are similar to multiplication in the way they handle two's complement versus positive-only representations. They differ in that they produce:

```
** ERROR 0101: Division by zero **
```

when their second operands are zero. The modulus operator returns the remainder of the division of its left operand by its right. Thus, the full-32 bit result of a 16-bit by 16-bit division is available by performing the division twice, once with the / and once with the MOD.

The SHL operator copies the bits of its left operand to the left

by the number of places given as its right operand. The new bits created at the right are zero. Thus SHL acts as a base two exponential function of the form:

```
X SHL Y  => X * (2 ^ Y)
```

Of course, MACRO-88 has no actual exponential operator ^ as used above, but the description is clearer with it.

The SHR operator acts very much like SHL, except that it does not appear to work properly with two's complement, as SHL does. SHR copies the bits to the right, effectively dividing by a power of two. However, since the created bits at the left are all zeroes, negative two's complement numbers are destroyed.

2.2.1.6.2 Relational operators

Primarily for macro use, but useful in many other special cases, the relational operators all produce values intended for examination by the IF pseudo-op. This means they produce either a TRUE (-1 o ØFFFFH) or a FALSE (Ø) value, regardless of their operands. The values of TRUE and FALSE are chosen so that the logical operators will work nicely with them (as if they were one-bit flags instead of 16 bit integers). One can thus write:

```
IF      (X >= Y) AND (Z != Ø) OR OKFLAG
```

and the results can be interpreted easily.

The relational operators are: =, !=, >, >=, <, and <=. They are all 16-bit positive integer comparisons. The positive-integer-only and the two's complement representations are interchangeable with respect to comparisons of relative values, so these operators all work for both.

2.2.1.6.3 Logical operators

All the logical operators work on the 16 bits of their operands separately. That is, the corresponding bits of each 16-bit number operand are operated on and returned in the corresponding bit of the 16-bit result. The four operators work according to the following "truth table" for each bit. The table shows, for instance, that Ø ANDed with Ø produces Ø.

Binary op.	Ø Ø	Ø 1	1 1
AND	Ø	Ø	1
OR	Ø	1	1
XOR	Ø	1	Ø

Unary op.	Ø	1
NOT	1	Ø

2.2.2 Symbol Name Operands

A few special situations demand that a symbol name be used for an operand; an expression is not allowed. The REF and DEF pseudo-ops are the primary users of the symbol-name type of operand. The reason is that the REF and DEF pops are used for referencing and defining the value of a particular symbol in an external "symbol table" file. Thus the use of the general-expression type of operand would not be suitable because it yields a numerical value, when what we want is just the name of a symbol. For more information on the REF and DEF pops, see section 2.3.2 on pseudo-opcodes.

2.2.3 Expression List Operands

Certain pseudo-ops (DB and TITLE, for example) will allow an arbitrary number of expressions in the operand field. The whole operand field in such cases is considered to contain a single arbitrarily long operand called an expression list. Essentially, an expression list is just a sequence of normal expressions or "long strings" separated by commas. The value of an expression list is a sequence of eight-bit quantities. When long strings are evaluated, the characters between the quotes are used as the eight-bit values. When expressions are found in the expression list, the bottom (least significant) eight bits are used for the value. If the value of an expression is too large to fit in eight bits, then the error:

```
** ERROR 12Ø3: Value > 255 - truncated to eight bits **
```

results, but the rest of the expression list is evaluated as usual. The criterion for "fitting into eight bits" is actually a test of the high order byte of the 16-bit word: if it is all zeroes or ones, the word will "fit."

Now the distinction between "long" and "short" strings can be clarified. We have defined a string as a legal component of an expression list. However, strings of one or two characters are also recognized as expressions. Therefore, when reading an expression list, MACRO-88 must decide whether a string it finds is to be considered part of an expression, in which case the mathematical operations defined in expressions may be used with it, or whether the string is to be considered a "long string" as defined above. It is very useful to have certain strings be part of an expression so that such operations as setting the eighth bit of the ASCII character are available. For example:

```
        DB      'Hello ther','e'+8ØH
```

will assemble "Hello there" with the eighth bit of the "e" high.

The problem is resolved in MACRO-88 by defining a one-character

string to be part of an expression (a "short string") and any longer string to be self contained (a "long string"). This allows math on single characters as shown above.

Expressions are allowed to have two-character strings as operands because they can be considered 16-bit quantities. Then why was the boundary of short versus long chosen at one rather than two characters? The reason is that a two-character string would evaluate differently when considered an expression than when considered a string. This is because the first character of a two-character string in an expression would be put in the higher order byte of the expression value and would be thrown away (as well as generate an error!). One-character strings can be considered either expressions or strings, with no change in the results. MACRO-88 looks for strings while scanning an expression list. When it finds one that has only one character in it, it backs up and assembles it as an expression.

Here are some examples of expression lists:

```
'T','h','e',' ','h','a','r','d',' ','w','a','y',CR
'The easy way',CR
1,2,3,4                    (simple expression list)
3+BINGO AND 0FFH,'TWO' (combined expression and string)
'This is one line',CR,0  (The common NUL terminator)
NULL[#1],LEN['HELLO'],'HELLO'
LABELSYM / 256 , LABELSYM AND 0FFH (backwards address)
```

For more information on the format of a string, see Section 2.2.1.4 on short strings used in expressions.

2.2.4 File Name Operands

On the System 88, there is a special format for specifying a disk file. This format, known as a file specifier, has been transferred to the syntax of some of the operands for pseuo-ops that need to work with disk files. The REFS and DEFS pops use these operands. They are described in detail in the System 88 User's Manual, but are shown here for completeness. The format looks like:

```
<optional drive number>Filename.EX
```

where the optional-drive-number is a single digit from 1 through the highest drive number, the Filename is a character sequence up to 31 characters long, and .EX is a two-character extension that indicates the file type. The System Drive is the default drive; its number need not be stated. If the drive number is omitted, the angle brackets are too. Some examples:

<3>Bingo-Ive-got-it.TX	(text file on disk in drive 3)
<2>HELLO.GO	(object code file on disk in drive 2)
MY-SYMBOLS.SY	(a file on System Drive, with the .SY extension used by the REFS and DEFS system)

2.3 OPCODES AND PSEUDO-OPCODES

This section discusses the function of opcode statements (which generate machine instructions in the binary output file) and pseudo-opcode statements (assembler directives).

2.3.1 The 8080 Instructions

Here we consider the way the assembler is used to specify any of the 8080 instruction codes (opcodes) with the desired register or immediate data fields. For a complete listing of the instructions themselves, see Appendix C: The 8080 Opcodes.

Some 8080 instructions accept a second byte after the opcode. These two-byte instructions are called "immediate" byte instructions because the one-byte operand follows the opcode immediately in memory. Any valid expression can be used in the operand field to specify the value of such an immediate operand, as long as it will "fit" in one byte. If it will not, we get:

```
** ERROR 1203: Value > 255 - truncated to eight bits **
```

and the lower eight bits of the value are used.

Instructions that refer to absolute memory addresses in the 8080 memory space or that put immediate data into a register pair have two bytes following the opcode. The two bytes are called a double word, an address, or just a word. The expression in the operand field follows any register pair specification operand. The value of the operand must not be greater than can be expressed in 16 bits, or an error results. There are only a few ways to get a value that is too big. One of them is to specify a number that is too large, e.g. 65536D. In this case, we get the error:

```
** ERROR 0D04: Number has value > 65535 **
```

Most 8080 instructions have some form of register specification imbedded in the opcode byte. These instructions may be one, two, or three bytes long, owing to immediate fields as described above. Register specifications are NOT generalized expressions as used for any of the immediate data operands. Instead, a register spec can be any of the following designations:

A B C D E H L PSW SP

Here are some examples of operand usage in instructions:

```
        LXI     H,Ø             ;Loads H and L regs. with Ø.
        CALL    AGAIN           ;Subroutine call to "AGAIN."
        JMP     DOWN            ;GOTO "DOWN."
AGAIN   SHLD    ROCK            ;Put HL into ROCK and ROCK+1.
        LHLD    STONE           ;Get HL from STONE and STONE+1.
        LXI     SP,TOS          ;Put "TOS" address into 8Ø8Ø
                                ;stack pointer.
        JM      ITSMINUS        ;Jump if accum minus.
        STA     SMALLROCK       ;Put accum value in SMALLROCK.
        LDA     SMALLSTONE      ;Get accum value.
```

The label field of a statement that has an instruction mnemonic in the opcode field (an instruction statement) is optional. If the label field is used, the symbol name it contains is entered into the symbol table as a label symbol with the value given by the address of the first byte of the instruction. Thus the JMP in the following code will take the machine to the CALL instruction rather than to the IN instruction.

```
        JMP     LAB1                    ;Jump to it.
.
.
LAB1    CALL    SUBR1                   ;Do the routine.
        IN      8
.
.
```

2.3.2 Pseudo-Ops in MACRO-88

2.3.2.1 EQU

The EQU (Equate) pop has the form:

```
        <symbol.name>   EQU     <expression>
```

A new symbol is created in the symbol table and given the value that the expression evaluates to. If the symbol name has already been used for a label or in another equate, a double definition error results.

2.3.2.2 ORG

The ORG (Origin) pop has the form:

```
        <optional.label>        ORG     <expression>
```

The value of the expression is used as the new assembly location. Enough zeroes are put into the object code file to fill the space from the address of the last instruction assembled (before

the ORG pop) to the address given by the expression in the ORG. Instructions generated after the ORG will thus be loaded starting at this address. Note that there is no way to skip backwards, since the assembler can only ouput more object code and cannot retract what it has previously output. If the value of the expression is less than the current location counter, we get an error:

** ERROR 1001: Cannot ORG backwards. ORG ignored **

However, if the backwards ORG occurs before any code is generated, MACRO-88 will allow it, and will only change the location counter. This makes it possible to put a group of DS statements in an area of memory beyond the program area, but to have these DS statements occur at the front of the program if desired. There will be no extraneous zeroes placed in the object code file as there would be if the ORG were placed after the program text.

The label field on the ORG instruction may contain a symbol name, in which case the symbol is defined and given the value of the new location counter (the same value as the value of the expression).

2.3.2.3 END

The END pseudo-op has the following form:

<optional.label> END

It serves to indicate the physical end of the source code to the assembler. If the END statement is not found, MACRO-88 will ask the operator for a continuation file. If the operator does not want a continuation file but has simply forgotten to use the mandatory END statement, he may type just a carriage return with no file specification, and the result will be just as if there had been an END statement. If there are any program statements after the END statement, an error results:

** ERROR 1601: Statements found after END **

2.3.2.4 Macro system pops

The following pseudo-opcodes are for controlling the macro processor system of MACRO-88. Since macro programming is an advanced process and rather difficult to describe, it has been explained separately in Appendix E by means of direct examples. The entire appendix is the listing of an assembly program. Thus the effects of the macros can be seen as interpreted by MACRO-88 itself. Also, there are conventions of style and format that are best represented by example. Here we give the descriptions of the pops themselves, but leave out, for example, the way the REF, DEF, REFS, and DEFS pops are used to create macro libraries.

2.3.2.4.1 MACRO

The MACRO pseudo-op has the form:

```
        <macro.name>    MACRO
```

The macro name label is put into the symbol table as a macro symbol, and all statements in the text following the MACRO pop up to (but not including) the ENDM pop are stored in the symbol table as the value of the symbol. The sequence of statements between the MACRO and the ENDM pops is called the "macro definition" or "prototype." None of the statements are assembled normally: they are just read in and checked to see if one is the ENDM pop. Errors in the statements of the definition will not be detected: they are just scanned and read. Example:

```
BINGO   MACRO           ;Begin the definition of BINGO.
        DB      'HI'    ;This statement is the "prototype."
                        ;You could have several instructions
                        ;here.
        ENDM            ;This terminates the definition.
        .
        .
        .               (Some intervening program... )
        .
        BINGO           ;Reference the MACRO.
```

After the definition in the text, there may be any number of "macro references," which are statements that use the macro symbol name as if it were an opcode. The result of the macro reference is to substitute the prototype into the text as if it had been included in the original source. The example above would assemble the

```
        DB      'HI'    ;This statement is the "prototype."
```

after any reference to the macro.

As with any other symbol name, macro names must be unique. No opcode, pseudo-opcode, label, or set variable may have the same name as any other. If a name in use is used on something else, the double definition error results:

```
        ** ERROR 1302: Doubly defined symbol **
```

or:

```
        ** ERROR 1301: Cannot redefine system symbol **
```

MACRO/ENDM pairs can be nested to any depth. This allows a macro to define a macro. A MACRO/ENDM pair contains the prototype for a macro, so if there is another MACRO/ENDM pair within that pair, the inside pair(s) will be part of the prototype. Then, when the prototype is evaluated by doing a reference to the symbol

name for the containing MACRO/ENDM pair, the inside MACRO pop will be seen "at the top level" and will define another macro. This is shown in Appendix E.

2.3.2.4.2 ENDM

ENDM has the form:

<ignored.label> ENDM <ignored.stuff>

It terminates a macro definition as discussed above. If used without a corresponding MACRO pop, the error:

** ERROR ØFØ1: ENDM without previous MACRO **

results. Any number of ENDMs may occur with no intervening MACROs in practice, because the MACRO/ENDM pairs may be nested to any depth desired. The ØFØ1 error occurs when an ENDM is used at the "top level"-- in other words, when there are already as many preceding ENDMs as MACROs.

The converse problem occurs when there are too few ENDMs as compared with MACROs in a program. However, there is no way for MACRO-88 to know that this was not intended, because the programmer may have the rest of his macro in another file. Thus MACRO-88 will sometimes ask:

No END found. Continuation file =

if an ENDM is left off a macro definition that seems to merit one. This results because it is allowable to put the END pop inside a macro definition. Leaving off ENDM just makes the definition very long!

2.3.2.4.3 MACLIST

MACLIST has the form:

MACLIST <expr>

MACLIST has the same form as LIST, and in fact has a similar effect. The difference is that it affects only the listing of macro expansions, where LIST controls the listing of all listing lines.

Like LIST, MACLIST sets an internal variable according to the value of the expression in the operand field. But this variable is distinct from the LIST variable, so they may have different values. During macro expansion, the value of the LIST variable is saved and set to the value of the MACLIST variable. When the expansion ends, the old value of LIST is replaced.

Some useful capabilities arise from this scheme. Inside a particular macro, for example, the LIST pop can be used to control

the listing of the text of the macro itself. But when the macro is done expanding, the outside text will continue as it was before, either being listed or not. A typical technique is to use inside the macro a LIST pop with operand field containing the name of a set variable that is used to control the printing of a certain class of macros-- maybe all the macros gotten from a particular library. At the top of the program, setting the single set variable will control the printing of all the associated macros. If the macros do not use this convention, they can all be controlled at once with the MACLIST at the top of the program, or individually by separate MACLIST pops before each macro reference.

2.3.2.5 SET

The SET pseudo-op has the form:

```
<symbol.name>   SET     <expression>
```

It is used for defining the value of a special kind of symbol known as a set variable. Set variable symbols can be redefined during assembly; no "doubly defined symbol" error will result, as with normal label symbols. The first time the SET pop is used, the new variable is created, and each subsequent time it is used, the value will just be changed. The set variable symbol can be used in an expression for an operand anywhere a label symbol can be used. If a set variable is used before it is defined, no error results, and an undefined value is used in the reference. If a set variable is redefined as a label after it has been created, an error results:

```
** ERROR 1304: Cannot use a SET VARIABLE as a LABEL **
```

If a label symbol is redefined as a set variable using the SET pop, this error results:

```
** ERROR 1303: Cannot SET a previously defined LABEL **
```

2.3.2.6 DS

The form of the DS (Define Storage) pseudo-op is:

```
<optional.label>   DS   <expression>
```

The expression value is used as a skip distance: the location counter will advance that number of bytes before the next statement is read. The output object code file gets a string of zeroes as long as the skip distance. The optional label will define a symbol at the location counter address that is in effect before the skipping takes place. Thus the label gets the address of the first byte in the storage area.

The DS pseudo-op works somewhat differently in the special case when no object code has been written to the output file at the

point of use. In this case, instead of outputting the string of zeroes, it just moves the location counter. This way, the output file is smaller because it does not have redundant zeroes in front.

2.3.2.7 DB

The form of the DB (Define Byte) pseudo-op is:

```
<optional.label>   DB   <expression.list>
```

The value of the expression list is a string of bytes written directly to the object code file. See Section 2.2.3 on the expression list type of operand for more information. If the optional label is used, the label symbol defined will have the address of the first byte in the contiguous string of bytes defined.

2.3.2.8 DW

The form of the DW (Define Word) pseudo-op is:

```
<optional.label>   DW   <expression.list>
```

The DW pop works very similarly to the DB pop, but the expression list is used as a string of double words rather than bytes to be placed directly in memory. A double word is a sequence of two bytes in memory that represents a 16-bit integer. The most significant byte of the 16-bit number is stored in the higher-addressed byte of memory. Each element of the operand field of the DW pop is evaluated and its value written to the object code file, with the low order byte first. The location counter advances by two places for each element produced in this way. The low-byte-first format standard for 16 bit numbers in memory used by the 8080 is called by some the "byte-reversed" format, because the bytes appear to be backwards in storage when memory byte values are written from left to right (ascending addresses).

Example: The statement

```
DW      1234H
```

is stored in ascending-order addresses as 34 Hex, then 12 Hex.

Note that the ability of the DB pop to combine a sequence of ASCII character-valued elements in the operand list into one long quoted string is not available with the DW pseudo-op:

```
DB      'hello there'              (legal)
DW      'hello there'              (illegal)
DW      'eh','ll',' o','ht','re','e' (same as DB)
```

2.3.2.9 PRBASE

The PRBASE (Print Base) pseudo-op has the following form:

```
<optional.label>   PRBASE    <expression>
```

The numeric printing base of the object code in listings generated by MACRO-88 may be set to any base from 2 to 36 by using the PRBASE pseudo-op. The value of the expression is used as the new base if it is within this range. If it is not, the error:

```
** ERROR ØD02: Base is out of range - must be 2 -> 36 **
```

is printed. PRBASE changes the address and object code printouts on the assembly source listing. Also, the base of the value field of each element in the symbol table printout is given by the last PRBASE in the program. See Section 1.4 on the assembly listing for more information on the effects of the base change by the PRBASE pseudo-op.

The default print base is 16 (hexadecimal).

2.3.2.10 LIST

The general form for the LIST pseudo-op is:

```
<optional.label>  LIST  <expression>
```

If the value of the expression is zero, the assembly listing is suppressed. If the value of the expression is non-zero, the assembly listing is turned back on, or if already on, is left on. The default condition is as if a LIST 1 had been executed (just before the user's source code is read); in other words, the assembly source listing will normally not be supressed unless a LIST Ø is found.

A common way to use the LIST pop is to have the expression be a set variable or a label symbol whose value has been defined by an EQUate pop at the beginning of the program. In this way, the programmer can make control structures that allow him to prevent selected sections of his program from being printed by merely changing the value of the EQUate symbol. For example:

```
HELPLIST   EQU  1        ;Print the HELP subroutine.
.
.
           LIST  HELPLIST
HELP       (the code for the 'HELP' subroutine...)
.
.
           LIST  1
```

will print the "HELP" subroutine only if the HELPLIST symbol is

EQUated to 1. To turn off the "HELP" printout, one would EQUate to Ø.

2.3.2.11 Conditional assembly

The following pseudo-ops allow selective skipping of text. They can be controlled by the value of an arbitrary expression, such as a symbol name, or a relational expression (BINGO>=QUASI*5-3). Skipped text is read in but completely ignored except for a quick check to see if it contains another conditional pop. There are four conditional pops: IF, ENDIF, ELSE, and ELSEIF. Sections of text are delimited by the IF/ENDIF pair, with possible ELSEs and ELSEIFs in between. The sections can be nested within one another to an almost arbitrary depth. The nested pairs can be skipped over as a unit-- in other words, MACRO-88 knows which ENDIF goes with which IF. Note that since even the END statement may be skipped, an IF without an ENDIF can sometimes skip all the way to the end of the program and result in the operator getting the familiar prompt:

```
No END found.  Continuation file =
```

Conditional assembly is typically used for testing the value of a symbol, as when an EQU statement is used at the beginning of a program to control the overall configuration of the assembled program. Certain segments of text can be selected for inclusion or omission on the basis of the value of a single symbol, so the entire program can be given a massive editing job merely by changing the single EQU statement.

Another common use of conditionals is in macros. A macro can be written to produce different results if it is used in different contexts. As an example, consider a macro that is to assemble a particular subroutine the first time it is referenced, then on successive expansions is to produce CALLs to the subroutine. One way to do this is to have the macro first test the value of a certain set variable. If the value is -1, say, then the subroutine is assembled and the set variable is set to Ø. Otherwise, a call to the subroutine is assembled and the set variable is left alone. (Note that some way to set the set variable to -1 initially is needed.)

2.3.2.11.1 IF

The form of the IF pseudo-op is:

```
<optional.label>   IF   <expression>
```

If the value of the expression is zero, the statements following the IF pop are skipped until a matching "ENDIF" pop is found. If the value of the expression is non-zero, the statements following the IF are assembled normally. Within each IF/ENDIF pair there may be another pair. The maximum nesting level is 256.

If an IF pop is found but a matching ENDIF is not found, no error results. But if a lone or unmatched ENDIF is found, we get:

```
** ERROR 1501: ENDIF/ELSE/ELSEIF without IF **
```

2.3.2.11.2 ENDIF

ENDIF ihe matching pseudo-op for the IF pseudo-op above. Its general form is:

```
<optional.ignored.label>  ENDIF
```

2.3.2.11.3 ELSE

ELSE is an internal delimiter pop for IF/ENDIF pairs. By itself, the ELSE statement is illegal and gets the 1501 error shown above. The form is:

```
<optional.ignored.label> ELSE
```

The ELSE pop acts to switch from skipping to non-skipping and non-skipping to skipping mode between an IF and an ENDIF. Two ELSEs in a row inside a single IF/ENDIF pair result in the second one being ignored.

Normally, ELSE is used in a structure with the other conditional pops as follows:

```
IF      <expr>
.
<subsection.A>
.
ELSE
.
<subsection.B>
.
ENDIF
```

The effect is to assemble subsection A if <expr> is non-zero or subsection B if it is zero. One and only one of the subsections will assemble. Any number of IF/ENDIF pairs (and hence of the above structure) can be imbedded within subsections A or B.

2.3.2.11.4 ELSEIF

ELSEIF is exactly the same as ELSE, except that instead of just reversing the skip condition (skip-no skip, no skip-skip), it tests the value of the expression in its operand field just as IF does. The form, then, is:

```
<optional.ignored.label> ELSEIF <expr>
```

A sequence of ELSEIFs between IF and ENDIF do not result in the last ones being ignored, as results from a sequence of ELSEs. Instead, the ELSEIFs are evaluated until one of them (or the original IF) "succeeds" (doesn't skip). After one has succeeded, the other ELSEIFs or ELSEs are skipped up to the ENDIF. The normal sequence of the four pops is:

```
IF      <expr>
.
<subsection.A>
.
ELSEIF  <expr>
.
<subsection.B>
.
ELSEIF  <expr>
.
.
.
ELSE
.
<default.subsection>
.
ENDIF
```

This structure will select one and only one subsection of the IF/ENDIF delimited area to be assembled. All the other subsections are skipped. The conditions that are tested have a priority structure: if any <expr> "fails" (evaluates to zero), then the rest of the tests are ignored. Conversely, if any test is to be made, all the <expr>s above it must have failed. The subsection delimited by the ELSE/ENDIF combination denotes the "default" condition for this structure, as it is the section which will always be assembled if all the others are skipped.

Any of the subsections may contain another IF/ENDIF pair or a structure similar to the one above, nested up to the depth limit of 256 levels.

2.3.2.12 PAGE

The PAGE pop is "invisible": it does not appear on the listing. However, it is apparent whenever it has been used, because after a PAGE the listing skips to the top of the next page before continuing. When the listing is being put on the video display rather than the printer, PAGE is not invisible and has no effect.

Form:

```
    <ignored.optional.label> PAGE       <optional.ignored.comment>
```

2.3.2.13 Library system pops

Normally, the problem of assembling extremely large programs is handled by providing a "linkage editor" system which allows separate assembly of subsections of the program, followed by a "linking" step which connects the object code for the subsections into a single program. The System 88 has no such linkage editor, and the problems involved in providing one are large (such as the necessary complexity of relocation formats and the resultant large volume of dictionary information associated with a relocatable segment for the 8080). Discussion of the design of relocation formats, linkage dictionaries, and linker/loaders is beyond the scope of this manual, though some appreciation of the principles involved is important for the advanced assembly-language programmer.

MACRO-88 makes up partially for the lack of a linkage editor system on the System 88 by inclusion of four pseudo-ops. They constitute a complete library generation and reference mechanism for all kinds of data. As such, they are actually more useful than a linkage editor. They are commonly used not only for passing the addresses of labels from one program to the next, but also for defining large libraries of macros and global constants or system labels.

The four pops are DEFS, DEF, REFS, and REF. The first two are used to create a library file, and the second for accessing it. They way they work is described below. Here we consider how to use them.

Any symbol in the symbol table can be written into a library file if it is defined within a program. DEFS and DEF do this. Another program can later access the library file with REFS and REF and re-create the same symbol definition within its own symbol table. Since macros, labels, register names, set variables, and even opcodes and pseudo-opcodes are all defined as symbols, they can all be put into library files and accessed later.

Two examples suggest the usefulness of the system. The first is the common situation in which a program is assembled to run in the "user" address area starting at 3200H. The addresses of utility subroutines and data areas within this program are to be shared with a collection of "overlays." The overlay mechanism on the System 88 is described in the System Programmer's Guide. Basically, the overlays all run at 2000H and can only run one at a time. Therefore, they cannot communicate between each other, but act as independent, very large swappable subroutines to the main program. The library system is used in this example to pass address information from the main program (the "root") to the overlays. The root creates the library by using DEFS and DEF. The library file is referenced by the overlays when the assembler finds REFS and REF. The classic example of this use is in the definition of the System 88's own root,

which has a library file called SYSTEM.SY. This library is commonly available to systems programmers.

The second example of the use of the library system is in building macro libraries. In this case, the symbols saved in the library are all macro definitions. The program which builds the macro library generates no object text at all; it only builds the library file. Later, unwritten programs will use the library even after the defining program is retired to the archives. Here is where the MACRO-88 library system beats the linkage editor. Examples of actual macro library definition and reference are given in Appendix E.

2.3.2.13.1 REFS

The form for the REFS (Reference File Specify) pseudo-op is:

<optional.ignored.label> REFS <file.name.operand>

For more information on the structure of the file name operand, see the System 88 User's Manual or section 2.2.4 of this manual on file name operands.

The action of REFS is to open a special kind of input file, called a "library file." This kind of file is usually created by using the DEFS and DEF pseudo-ops described later. Once a REFS has been seen and accepted by MACRO-88, the REF pseudo-op may be used. If there is no REFS, then the use of REF will generate:

** ERROR 1702: REF without previous REFS **

If the input file is not found or is specified improperly, assembly stops, and the video display gives the System 88 error message corresponding to the error.

2.3.2.13.2 REF

The form of the REF (Reference) pseudo-op is:

<optional.ignored.label> REF <symbol.name.operand>

For more information on valid symbol name operands, see Section 2.2.2, Symbol Name Operands.

This pop is used as an external reference system for assembly-time definition of symbols created and stored in a library file by a previously assembled program. The symbol whose name is given in the operand field is created in the program currently being assembled. Its value and type are given by its entry in the library file; the value and type are unrestricted except that they must make sense as symbol table elements. In other words, any symbol put into the library file by a previous use of DEFS and DEF came out of the symbol table from a previous

assembly, and thus is guaranteed to make sense as a symbol table element. Library files created by other means than using the DEFS and DEF in an assembly program are not checked for validity, and can produce anomalous results.

If the symbol being defined through a REF is defined elsewhere in the program, then we get:

** ERROR 1701: Double definition by REF **

If there is no symbol with the given name in the library file, then the symbol is left undefined after the REF, and we get the error:

** ERROR 1704: Symbol not found in REF file **

2.3.2.13.3 DEFS

The form of the DEFS (Definition File Specify) pop is:

<optional.ignored.label> DEFS <file.name.operand>

The function of DEFS is to open an output file for subsequent output definitions of external symbols via the DEF pseudo-op. The extension of .SY (see The System 88 User's Manual for information on extensions) is standard for library files. When no extension is given for the file name in DEFS, .SY is used by default. If one is given, it is used instead. If the file to be opened already exists, or if the name is illegally constructed, or if for any other reason the system cannot open the desired file, the assembly terminates, and the appropriate System 88 error message is printed on the video display.

Each use of DEFS closes a previous DEF file if one was open. If a DEF file is still open at the end of an assembly, it is closed automatically.

2.3.2.13.4 DEF

The form of the DEF (Define) pseudo-op is:

<optional.ignored.label> DEF <symbol.name.operand>

The DEF pop is used to write the definition of the symbol whose name appears in the operand field out to the symbol table file currently open (see DEFS above). If the symbol with the given name is not defined in the program, we get:

** ERROR 1802: DEF given an undefined symbol **

If the symbol given has been defined twice or more in the program, then we get:

** ERROR 1803: DEF given a doubly defined symbol **

If the DEF pop is used before a valid DEF file has been established by the DEFS pop, then the error:

```
** ERROR 1804: DEF without previous DEFS **
```

results.

The REF/DEF combination is very powerful in that it can be used to define any kind of symbol except labels that are doubly defined or undefined. This means that MACROs or set variables can be stored in a library file and read back in. They will maintain their types when passing through the REFS/DEFS system. Note that the set variables must not be defined some other way before their values are defined by REF, or REF will think they have been defined twice (REF does not understand the difference between set variables and any other kind of variable) and will give a double definition error.

2.3.2.14 IDNT

The form of the IDNT (Identify) pseudo-op is:

```
<optional.label>  IDNT  <expression>,<expression>
```

The optional label, if present, defines a label symbol with a value equal to the current location counter value, which is not changed. The first expression in the operand field gives the load address (La) for the output file, and the second gives the start address (Sa). These two values are associated with the directory entries on every file stored on the System 88, but only runnable files use non-zero values for them. The .GO file that the assembler creates will have La and Sa fields as given in the last IDNT pseudo-op assembled.

The load address of a file is the address where it will be placed in memory when it is called to be executed. The start address is the address of the first instruction that the system will call by after the file has been loaded in order to begin its execution. The start address may be any address at all, but is most often the first instruction in the program. The fourth byte of the program is assumed to be the "warmstart" location of programs which run in user memory space. This means that if for any reason the program is interrupted, it can be restarted without its re-initializing all of its internal data areas. For example, a text editor should restart with its text buffer containing the text that was in it when the editor was exited. The use of byte four comes about due to the convention of having a JMP START followed by a JMP WARM as the first two instructions of a user program. Thus the first few instructions of programs to be run in "user" space usually look like:

```
        ORG     3200H   ;Start of user space in memory.
        IDNT    $,START ;La=3200H, Sa=START.
        JMP     START   ;Redundant for START command.
        JMP     WARM    ;For REENTER command.
        REFS    SYSTEM.SY       ;Open the system library.
        REF     ...             ;Start getting globals.
        .
        .
START   ;Entry point for complete initialization.
        .
        .
WARM    ;Entry point for clean re-initialization.
```

Section 3

RUNNING THE ASSEMBLER

This section discusses the activity called "running the assembler" or "doing an assembly." The purpose of running the assembler, of course, is to produce a correctly translated object-code program suitable for execution.

3.1 INVOKING THE ASSEMBLER

To bring the assembler into memory and have it assemble an assembly-language program, you type the "invocation" line as a command to the Exec system, giving the file name of the source program and the name you want to assign to the object-code program:

```
$Asmb <2>SOURCE-ASSEMBLY-FILE,<3>OBJECT-ASSEMBLY-FILE
```

The first file specification will be used as the input source text, and the second as the object code output file. The output file should not already exist, or else a message will be printed:

```
Output file already exists!
$
```

and the system returns to Exec. Likewise, the input file must already exist, or the system prints:

```
Input file does not exist!
$
```

and returns to Exec. If the output file is not specified (i.e. only one file is specified) then no object code will be generated. This is not an error.

3.2 THE OPTIONS CONVERSATION

Upon invocation, the assembler is loaded into memory and starts executing. The first thing it does is talk to the user about how to do the assembly. This is called the options conversation. Some yes/no questions are asked; the user responds by typing Y or N, followed by a carriage return. When MACRO-88 is satisfied, it will begin reading the source file, and the drive containing it will emit an audible "clunk" as the read/write head is "loaded" (brought up to the disk surface). The red "active" light will also come on, and the "Pass one" message printed. Below is the display on the video screen for an example conversation and complete assembly.

```
$Printer Terminet
$Asmb <2>RAKE7 <3>RAKE
MACRO-88 Version  vvv: mo/day/yr.
Hardcopy? (default is video display) (Y or N):Y
Full listing? (else errors only) (Y or N):Y
Symbol table printout? (Y or N):Y

Pass one.
Pass two.                          (listing is being printed)

Error total =      0
$                                  (Return to Exec)
```

In this conversation, the user first asks the printer configuration program to install a printer driver program called "Terminet." Once the driver is installed, the system understands how to "talk" to a particular kind of printer. MACRO-88 sends characters down the printer "wormhole," which is its means of communication with the currently installed printer driver. Thus MACRO-88 does not know what is being done with the listing it produces, and the user has complete control over the fate of the listing text.

In the example, after the MACRO-88 invocation, the user types Y to the first question. This tells MACRO-88 to send the listing to the wormhole. An N answer would send the listing to the video screen, and the wormhole would not have to be set up for the printer driver Printer Terminet. The video display mode prints the listing on the screen, stopping every 14 lines, printing a period, and waiting for the user to type a character before continuing the listing.

The next question determines whether each line of the original text will be printed in the listing or only the lines that generate errors. The errors-only mode will omit correct source text lines but will print all other output lines, such as the symbol table printout and error summary.

The last question establishes whether the symbol table listing will occur.

Any of the questions can be answered no with:

N, n, or nothing at all

and can be answered yes with:

Y or y

All responses are followed by a carriage return.

After the last question, the user sees

Pass one.

so he or she sits back and waits for

Pass two.

when the listing starts. Pass one takes about one minute for each 1000-3000 lines, depending on the nature of the program text. Pass two takes somewhat longer, but is usually limited by the speed of the printer anyway. When printing is complete, the error total is displayed as a check for the user. Finally, the output file is closed (the drives clunk back and forth for a while) and the system returns to Exec and gives the $ prompt.

If the assembler does not find an END statement, it asks the user to specify a continuation file. This makes it possible to have the source text for a program in many different files or even on separate disks. After the options conversation, the screen might look like this:

```
Pass one.                          (Normal start of assembly.)
No 'END' found.  Continuation file = <2>SOURCE-2.TX
                                   (User replaces disk and gives
                                   next file name.)
Pass two.                          (Continuation file had an END.)
Re-specify original input file please:<2>SOURCE-1.TX
                                   (User replaces first disk and
                                   types first file name.)
No 'END' found.  Continuation file = <2>SOURCE2.TX
                                   (User replaces disk again, but
                                    types wrong name.)
That file does not exist.
No 'END' found.  Continuation file = <2>SOURCE-2.TX
                                   (Now assembly continues.)


Error total =     0                (Hurray!)
$                                  (...back to Exec ...)
```

3.3 LISTING FORMATS

A listing is the text printed out by the assembler while it is translating the user's program. In the next few paragraphs, we will discuss the things that are printed by MACRO-88 in the listing.

3.3.1 The Source Code Listing

Each statement or line of the input source text is formatted or shaped by the assembler into a "source listing line," which is a copy of the original line along with the assembler's interpretation of it in terms of the object code that was produced by it. The source listing line consists of the address field, the object code field, and the source text field. The address field

is the print area for the address assigned to the instruction in the assembly statement. The object code field shows the actual data that was generated to go in the printed address. The source text field is just the original text line itself, with some of the right end of it cut off (usually only comments are lost) so that it will fit on the line.

```
Address   Object code          Source text
field       field                 field
-----------------------------------------------------

388C  FE04              CPI     4       ;If base <4, then no
388E  D29938            JNC     PUTLS8  ;next time through.
3891  CDD245    PUTLS3  CALL    PUTLINE ;Write out the line.
```

3.3.2 Effect of the PRBASE Pseudo-Op

The PRBASE pop, described in section 2.3.2.10, changes the value of a variable within the assembler (PTBASE) that indicates the current numeric base being used for all printed values except error counts. Its effect on the assembly source listing is to change the address field and object code field printouts. None of the source text field is changed. If the base is less than 4, then an address takes up so much space in the object code field that there is not enough left for even one byte of object code. In this case, the object code is printed on the following lines, as shown in this example using base 2:

```
1011000101011110        CALL    COLLECT ;After 8:00-- it's
      11001101
      00010100
      10001001
1011000101100001        ANI     0F8H    ;cheaper
      11100110
      11111000
```

When PTBASE is large enough that numbers can fit two on a line, they are packed together without any space in between. This is easy to understand with hexadecimal, but is a little strange, perhaps, in octal:

```
067303 346201           ANI     LITTLEORPHAN
```

especially since the address field is printed in "packed" octal rather than the preferred "split" octal.

The PTBASE in effect when the assembly also effects the printout of a symbol table listing if one is requested. The normal base 16 (hexadecimal) printout fits conveniently on the 64-character video display line, but is too long for lower bases. This is normally no problem with hardcopy, since most printers have at least 80 columns. The symbol table listing displays eleven characters of the symbol name followed by the value and a space. Thus the printout for hex is 11+4+1 characters, or 16 characters

per element. We print a fixed four elements per line. For base two, then, the line is 11+16+1 characters per element * 4 elements, or 112 characters. A possible way to avoid problems with this expansion is to use a PRBASE 16 just before the END statement.

3.3.3 Full and Errors-Only Listings

In the options conversation, the user is asked whether he wants a "full" or "errors-only" listing. In a full listing, all the lines in the source program are printed, whereas in an errors-only listing only the lines that contain errors are printed. The errors-only listing is a handy way to check a program for validity or to find the errors in it without having to wait for the whole printout or having to inspect all the lines of a listing for error messages. Each error line printed in an errors-only listing is followed by the usual error message with the English text summary of the problem.

3.3.4 Error Message Format

Error messages in MACRO-88 are plain text, rather than single-character codes as used by most assemblers. A typical error message in a listing might look like:

```
** ERROR 0401H: String too long - sorry **
```

Each statement in an assembly listing can generate up to two error messages. The first one may be a "non-fatal" error, but the second is fatal. A non-fatal error does not stop the translator from working on the rest of the input text on that line, while a fatal error results in the translator giving up completely and starting on the next input statement. The user does not need to be concerned with the two types of error, as there will almost always be only one error in one statement of the program. It may be important to know, however, that even if a statement has many things wrong with it, only two errors will cause messages to appear in the printout.

3.3.5 The Symbol Table Printout

The symbol table printout is a table that shows the names of the symbols found in the program and the values they were assigned. There are actually four tables that may appear in the printout, but usually most of them contain no elements, and so they are not printed. The four tables list the undefined variables, the doubly defined variables, the set variables, and the properly defined labels. Usually, the last table, the label table, is the only one that prints out. It may be anywhere from one or two elements to several hundred, and can require several pages of printout. Elements are printed four to a line. The base of the value printout is settable by the user and thus may change, shortening or lengthening the whole line. With base 16 (hexadecimal-- normal default) the print line is 64 characters

long and looks good on the video display. Thus:

```
Error total = 3

        Macros defined in this assembly:

big         lpg         db          sort

        Undefined symbols:

DEFSCLOSE   PLOVER      READC

        Doubly defined symbols:

DOORG2      0456

        SET variables used:

NCOUNT      0001 BCOUNT     0005

        Labels defined in this assembly:

ADDEXPR     3C5D ADDL1      47BB ADDLIST    47A5 ADDRCODE   0002
ADDRTYPE    4589 ADRTP1     458F AFIELD     5161 AFLDBLK    5163
.
.
```

and so on. The base of the number printout is the same as the last base used for the assembly object code printout. This is settable by the user through the PRBASE pseudo-op.

Appendix A

ERROR MESSAGES

Error messages included in the MACRO-88 Assembler are shown below, with the internal codes generated by the assembler. The actual message text is generated by the Amsg.OV overlay when it is passed the error code. Error messages in upper case are error system errors and should never be seen by the user. If you do get an all-capital error message or the message

!!!! No message for this error !!!!

please contact PolyMorphic Systems to report a system bug. The proper procedure is to fill out an SIDR form (Software Improvement or Difficulty Report), available from PolyMorphic Systems.

One particularly convenient feature of the System 88 is the ability of the user to alter message texts by means of the "Emedit" error message editor. Emedit is described in the System Library volume System 88 System Programmer's Guide. It can be used to change any of the messages in the following summary table. These are the error messages printed on the listing for errors generated by statements in the source text. None of the operator conversation messages are available in the Amsg.OV overlay.

ERROR MESSAGE SUMMARY

Error number.	Message text.
0101	Division by zero
0201	Expression has unbalanced parenthesis
0203	String is too long for a double-word
0204	Invalid operand - I can't figure it out
0301	Opcode field - symbol undefined as a macro
0302	!! OPCODE FIELD NON-OPCODE FOUND !!
0303	Opcode field is not a valid symbol name
0304	!! BAD OPCODE SUBTYPE !!
0305	Opcode field has symbol of illegal type
0401	String too long - Sorry!
0501	Expression too complex - Sorry!
0701	Number has a bad base specifier
0702	!! NUMBER HAS NON-NUMERIC FIRST CHAR !!
0801	Expression nested too deep - Sorry!
0B01	Missing comma
0B02	Register pair specification - value illegal
0B03	Register specification - value illegal
0B04	B/D register specification - value illegal
0C01	Symbol table full - Sorry!
0C02	!! NON-IDENTIFIER PASSED SYM FUNCTION !!
0C03	!! BAD INDEX INTO SYMBOL DATA FIELD !!
0C04	!! STINSERT CONFUSED !!
0C05	!! SYMFIND CAN'T FIND SYMBOL !!
0D01	Number has illegal digit
0D02	Base is out of range - must be 2 ->36
0D03	!! PRINT FIELD ERROR IN @NUM !!
0D04	Number has value > 65535
0E01	LABEL HAS NO ENTRY IN PASS 2
0F01	ENDM without previous MACRO
0F02	!! PSEUDO-OP HAS ILLEGAL SUBTYPE CODE !!
1001	Cannot ORG backwards. ORG ignored
1101	Undefined symbol reference
1102	Illegal symbol type reference
1203	Value > 255 - truncated to eight bits
1301	Cannot redefine system symbol
1302	Doubly defined symbol
1303	Cannot SET a previously defined LABEL
1304	Cannot use a SET VARIABLE as a LABEL
1306	Phase error - sym. val. changed in pass 2
1501	ENDIF/ELSE/ELSEIF without IF
1601	Statements found after END
1701	Double definition by REF
1702	REF without previous REFS
1703	!! REF LOST DEF BETWEEN PASSES !!
1704	Symbol not found in REF file
1801	!! DEF CAN'T FIND SYMBOL !!
1802	DEF given an undefined symbol
1803	DEF given a doubly defined symbol
1804	DEF without previous DEFS
1805	REF/DEF without symbol name

```
1901    Parm. list - unmatched brackets.
1902    Parm. list - unmatched quotes.
2001    MSTACK overflow - Sorry!
2002    !! MPEEK BUFFER SHORT !!
2003    !! MACPOP ON DE=0 !!
2201    Illegal character after '#'
0000    !!! No message for this error !!!
```

(Any unlisted code gives a zero as a code message.)

Appendix C

8080 OPCODES

REFERENCE LIST:

FUNCTIONAL LISTING OF THE 8080 INSTRUCTION SET

MOVE				ACCUMULATOR *			
40	MOV B,B	60	MOV H,B	80	ADD B	A0	ANA B
41	MOV B,C	61	MOV H,C	81	ADD C	A1	ANA C
42	MOV B,D	62	MOV H,D	82	ADD D	A2	ANA D
43	MOV B,E	63	MOV H,E	83	ADD E	A3	ANA E
44	MOV B,H	64	MOV H,H	84	ADD H	A4	ANA H
45	MOV B,L	65	MOV H,L	85	ADD L	A5	ANA L
46	MOV B,M	66	MOV H,M	86	ADD M	A6	ANA M
47	MOV B,A	67	MOV H,A	87	ADD A	A7	ANA A
48	MOV C,B	68	MOV L,B	88	ADC B	A8	XRA B
49	MOV C,C	69	MOV L,C	89	ADC C	A9	XRA C
4A	MOV C,D	6A	MOV L,D	8A	ADC D	AA	XRA D
4B	MOV C,E	6B	MOV L,E	8B	ADC E	AB	XRA E
4C	MOV C,H	6C	MOV L,H	8C	ADC H	AC	XRA H
4D	MOV C,L	6D	MOV L,L	8D	ADC L	AD	XRA L
4E	MOV C,M	6E	MOV L,M	8E	ADC M	AE	XRA M
4F	MOV C,A	6F	MOV L,A	8F	ADC A	AF	XRA A
50	MOV D,B	70	MOV M,B	90	SUB B	B0	ORA B
51	MOV D,C	71	MOV M,C	91	SUB C	B1	ORA C
52	MOV D,D	72	MOV M,D	92	SUB D	B2	ORA D
53	MOV D,E	73	MOV M,E	93	SUB E	B3	ORA E
54	MOV D,H	74	MOV M,H	94	SUB H	B4	ORA H
55	MOV D,L	75	MOV M,L	95	SUB L	B5	ORA L
56	MOV D,M		-------	96	SUB M	B6	ORA M
57	MOV D,A	77	MOV M,A	97	SUB A	B7	ORA A
58	MOV E,B	78	MOV A,B	98	SBB B	B8	CMP B
59	MOV E,C	79	MOV A,C	99	SBB C	B9	CMP C
5A	MOV E,D	7A	MOV A,D	9A	SBB D	BA	CMP D
5B	MOV E,E	7B	MOV A,E	9B	SBB E	BB	CMP E
5C	MOV E,H	7C	MOV A,H	9C	SBB H	BC	CMP H
5D	MOV E,L	7D	MOV A,L	9D	SBB L	BD	CMP L
5E	MOV E,M	7E	MOV A,M	9E	SBB M	BE	CMP M
5F	MOV E,A	7F	MOV A,A	9F	SBB A	BF	CMP A

* = All flags (C,Z,S,P,AC) affected.

RETURN

C9 RET
C0 RNZ
C8 RZ
D0 RNC
D8 RC
E0 RPO
E8 RPE
F0 RP
F8 RM

LOAD/STORE

0A LDAX B
1A LDAX D
2A LHLD Adr
3A LDA Adr

02 STAX B
12 STAX D
22 SHLD Adr
32 STA Adr

RESTART

C7 RST 0
CF RST 1
D7 RST 2
DF RST 3
E7 RST 4
EF RST 5
F7 RST 6
FF RST 7

ACC IMMEDIATE *

C6 ADI d8
CE ACI d8
D6 SUI d8
DE SBI d8
E6 ANI d8
EE XRI d8
F6 ORI d8

LOAD IMMEDIATE

01 LXI B,d16
11 LXI D,d16
21 LXI H,d16
31 LXI SP,d16

INPUT/OUTPUT

D3 OUT d8
DB IN d8

SPECIALS

EB XCHG
27 DAA *
2F CMA
37 STC **
3F CMC **

INCREMENT a

04 INR B
0C INR C
14 INR D
1C INR E
24 INR H
2C INR L
34 INR M
3C INR A

03 INX B
13 INX D
23 INX H
33 INX SP

STACK OPS

C5 PUSH B
D5 PUSH D
E5 PUSH H
F5 PUSH PSW

C1 POP B
D1 POP D
E1 POP H
F1 POP PSW *

E3 XTHL
F9 SPHL

DECREMENT a

05 DCR B
0D DCR C
15 DCR D
1D DCR E
25 DCR H
2D DCR L
35 DCR M
3D DCR A

0B DCX B
1B DCX D
2B DCX H
3B DCX SP

Adr = 16 bit address
* = All flags (C,Z,S,P,AC) affected.
** = Only CARRY flag affected.
a = All flags except CARRY affected (exception: INX & DCX affect no flags).
d8 = Constant, or logical/arithmetic expression that evaluates to an 8-bit data quantity.
d16 = Constant, or logical/arithmetic expression that evaluates to a 16-bit data quantity.

JUMP

```
C3  JMP Adr
C2  JNZ Adr
CA  JZ  Adr
D2  JNC Adr
DA  JC  Adr
E2  JPO Adr
EA  JPE Adr
F2  JP  Adr
FA  JM  Adr
E9  PCHL
```

MOVE IMMEDIATE

```
06  MVI B,d8
0E  MVI C,d8
16  MVI D,d8
1E  MVI E,d8
26  MVI H,d8
2E  MVI L,d8
36  MVI M,d8
3E  MVI A,d8
```

CALL

```
CD  CALL Adr
C4  CNZ  Adr
CC  CZ   Adr
D4  CNC  Adr
DC  CC   Adr
E4  CPO  Adr
EC  CPE  Adr
F4  CP   Adr
FC  CM   Adr
```

**

DOUBLE ADD

```
09  DAD B
19  DAD D
29  DAD H
39  DAD SP
```

CONTROL

```
00  NOP
76  HLT
F3  DI
FB  EI
```

**

ROTATE

```
07  RLC
0F  RRC
17  RAL
1F  RAR
```

FLAG BYTE STACK FORMAT

7	6	5	4	3	2	1	0
S	Z	0	A C	0	P	1	C

Adr = 16 bit address

d8 = Constant, or logical/arithmetic expression that evaluates to an 8-bit data quantity.

** = Only CARRY flag affected.

REFERENCE LIST: NUMERICAL LISTING OF THE 8080 INSTRUCTION SET

00	NOP	20	---	40	MOV B,B	60	MOV H,B
01	LXI B,d16	21	LXI H,d16	41	MOV B,C	61	MOV H,C
02	STAX B	22	SHLD Adr	42	MOV B,D	62	MOV H,D
03	INX B	23	INX H	43	MOV B,E	63	MOV H,E
04	INR B	24	INR H	44	MOV B,H	64	MOV H,H
05	DCR B	25	DCR H	45	MOV B,L	65	MOV H,L
06	MVI B,d8	26	MVI H,d8	46	MOV B,M	66	MOV H,M
07	RLC	27	DAA	47	MOV B,A	67	MOV H,A
08	---	28	---	48	MOV C,B	68	MOV L,B
09	DAD B	29	DAD H	49	MOV C,C	69	MOV L,C
0A	LDAX B	2A	LHLD Adr	4A	MOV C,D	6A	MOV L,D
0B	DCX B	2B	DCX H	4B	MOV C,E	6B	MOV L,E
0C	INR C	2C	INR L	4C	MOV C,H	6C	MOV L,H
0D	DCR C	2D	DCR L	4D	MOV C,L	6D	MOV L,L
0E	MVI C,d8	2E	MVI L,d8	4E	MOV C,M	6E	MOV L,M
0F	RRC	2F	CMA	4F	MOV C,A	6F	MOV L,A
10	---	30	---	50	MOV D,B	70	MOV M,B
11	LXI D,d16	31	LXI SP,d16	51	MOV D,C	71	MOV M,C
12	STAX D	32	STA Adr	52	MOV D,D	72	MOV M,D
13	INX D	33	INX SP	53	MOV D,E	73	MOV M,E
14	INR D	34	INR M	54	MOV D,H	74	MOV M,H
15	DCR D	35	DCR M	55	MOV D,L	75	MOV M,L
16	MVI D,d8	36	MVI M,d8	56	MOV D,M	76	HLT
17	RAL	37	STC	57	MOV D,A	77	MOV M,A
18	---	38	---	58	MOV E,B	78	MOV A,B
19	DAD D	39	DAD SP	59	MOV E,C	79	MOV A,C
1A	LDAX D	3A	LDA Adr	5A	MOV E,D	7A	MOV A,D
1B	DCX D	3B	DCX SP	5B	MOV E,E	7B	MOV A,E
1C	INR E	3C	INR A	5C	MOV E,H	7C	MOV A,H
1D	DCR E	3D	DCR A	5D	MOV E,L	7D	MOV A,L
1E	MVI E,d8	3E	MVI A,d8	5E	MOV E,M	7E	MOV A,M
1F	RAR	3F	CMC	5F	MOV E,A	7F	MOV A,A

d16 = Constant, or logical/arithmetic expression that evaluates to a 16-bit data quantity.

d8 = Constant, or logical/arithmetic expression that evaluates to an 8-bit data quantity.

Adr = 16 bit address

80	ADD B	A0	ANA B	C0	RNZ	E0	RPO
81	ADD C	A1	ANA C	C1	POP B	E1	POP H
82	ADD D	A2	ANA D	C2	JNZ Adr	E2	JPO Adr
83	ADD E	A3	ANA E	C3	JMP Adr	E3	XTHL
84	ADD H	A4	ANA H	C4	CNZ Adr	E4	CPO Adr
85	ADD L	A5	ANA L	C5	PUSH B	E5	PUSH H
86	ADD M	A6	ANA M	C6	ADI d8	E6	ANI d8
87	ADD A	A7	ANA A	C7	RST 0	E7	RST 4
88	ADC B	A8	XRA B	C8	RZ Adr	E8	RPE
89	ADC C	A9	XRA C	C9	RET	E9	PCHL
8A	ADC D	AA	XRA D	CA	JZ Adr	EA	JPE Adr
8B	ADC E	AB	XRA E	CB	---	EB	XCHG
8C	ADC H	AC	XRA H	CC	CZ Adr	EC	CPE Adr
8D	ADC L	AD	XRA L	CD	CALL Adr	ED	---
8E	ADC M	AE	XRA M	CE	ACI d8	EE	XRI d8
8F	ADC A	AF	XRA A	CF	RST 1	EF	RST 5
90	SUB B	B0	ORA B	D0	RNC	F0	RP
91	SUB C	B1	ORA C	D1	POP D	F1	POP PSW
92	SUB D	B2	ORA D	D2	JNC Adr	F2	JP Adr
93	SUB E	B3	ORA E	D3	OUT d8	F3	DI
94	SUB H	B4	ORA H	D4	CNC Adr	F4	CP Adr
95	SUB L	B5	ORA L	D5	PUSH D	F5	PUSH PSW
96	SUB M	B6	ORA M	D6	SUI d8	F6	ORI d8
97	SUB A	B7	ORA A	D7	RST 2	F7	RST 6
98	SBB B	B8	CMP B	D8	RC	F8	RM
99	SBB C	B9	CMP C	D9	---	F9	SPHL
9A	SBB D	BA	CMP D	DA	JC Adr	FA	JM Adr
9B	SBB E	BB	CMP E	DB	IN d8	FB	EI
9C	SBB H	BC	CMP H	DC	CC Adr	FC	CM Adr
9D	SBB L	BD	CMP L	DD	---	FD	---
9E	SBB M	BE	CMP M	DE	SBI d8	FE	CPI d8
9F	SBB A	BF	CMP A	DF	RST 3	FF	RST 7

Adr = 16 bit address

d8 = Constant, or logical/arithmetic expression that evaluates to an 8-bit data quantity.

d16 = Constant, or logical/arithmetic expression that evaluates to a 16-bit data quantity.

Appendix E

SAMPLE PROGRAM

The following program was assembled using the MACRO-88 Assembler.
It illustrates some of the techniques the assembler makes possible.

```
                ;*********************************************************
                ;*                                                       *
                ;*              MACRO TESTER / DEMONSTRATOR              *
                ;*                                                       *
                ;*  This is a demonstration of the use of the            *
                ;*  macro facility of the MACRO-88 assembler for the     *
                ;*  PolyMorphic Systems System 88 microcomputer.         *
                ;*  Illustrated are:                                     *
                ;*                                                       *
                ;*       1)      Basic macro definition format.          *
                ;*                                                       *
                ;*       2)      Referencing formal arguments within     *
                ;*               body of macro text.                     *
                ;*                                                       *
                ;*       3)      Basic technique of recursion and        *
                ;*               macro nesting.                          *
                ;*                                                       *
                ;*       4)      Using a macro to define a macro.        *
                ;*                                                       *
                ;*       5)      Listing control of macro expansions.    *
                ;*                                                       *
                ;*       6)      Format for actual arguments (params)    *
                ;*               using nested brackets, strings... .     *
                ;*                                                       *
                ;*       7)      The LEN[] and NULL[] functions.         *
                ;*                                                       *
                ;*********************************************************
                ;
* 3200                  ORG     3200H
  3200                  IDNT    3200H,3200H
  FFFF          TRUE    SET     -1
  0000          FALSE   SET     0
  000D          CR      EQU     0DH
                ;
                ;
                ;  A simple text-substitution macro.
                ;
                ;  The invocation line is the first line, the termination
                ;  line is the one with the ENDM on it, and the body is
                ;  the collection of lines in between.  Label field of the
                ;  invocation line is the name of the macro.
                ;
                bizzarre MACRO
                        DB      'WOW!!!',0
                        ENDM
                ;
                ;  A reference to the macro results in assembly of
                ;  the body of the macro as if it were in the input file.
                ;
                        bizzarre
  3200  574F5721          DB      'WOW!!!',0
  3204  212100
                ;
                ;
                ;  Parameters are passed into a macro in the operand
                ;  field of the reference.  The pound-sign operators used
                ;  within the body of the macro can be used to indicate
                ;  where to plug in the parameters for a particular ref-
                ;  erence into the text before it is assembled.  Pound sign
                ;  can also give the label that was used on the reference, and
```

```
                ;  the name that was used to reference the macro.
                ;
                flap    MACRO
                        DW      #1      ;The first parameter in operand field.
                        DW      #2      ;The second.
                #L      DS      Ø       ;The label on the reference.
                        DB      '#O'    ;The opcode field ("flap").
                        DB      '##'    ;A quoted pound sign.
                        DB      #A      ;All the operand field params at once.
                        ENDM
                        ;
                LAB1    flap    1,2
3207  0100              DW      1       ;The first parameter in operand field.
3209  0200              DW      2       ;The second.
320B            LAB1    DS      Ø       ;The label on the reference.
320B  666C6170          DB      'flap'  ;The opcode field ("flap").
320F  23                DB      '#'     ;A quoted pound sign.
3210  0102              DB      1,2     ;All the operand field params at once.
                ;
                ;
                ;  A recursive macro.
                ;
                ;  The first one starts a set-variable off at an initial
                ;  value.  The second one calls itself and decrements the
                ;  variable before each call, so that eventually it stops
                ;  calling itself and returns.  Then each call defines
                ;  a character string and returns to the previous call.
                ;
                repeat  MACRO           ;Initialization of recursion macro.
                COUNT   SET     #1
                        repeat1 #2
                        ENDM
                ;
                repeat1 MACRO           ;Descender macro.
                        IF      COUNT>Ø
                COUNT   SET     COUNT-1
                        repeat1 #1
                        DB      #1
                        ENDIF
                        ENDM
                ;
                        repeat  2,'HI'  ;Build 3 'HI's.
0002            COUNT   SET     2
                        repeat1 'HI'
FFFF                    IF      COUNT>Ø
0001            COUNT   SET     COUNT-1
                        repeat1 'HI'
FFFF                    IF      COUNT>Ø
0000            COUNT   SET     COUNT-1
                        repeat1 'HI'
0000                    IF      COUNT>Ø
                COUNT   SET     COUNT-1
                        repeat1 'HI'
                        DB      'HI'
                        ENDIF
3212  4849              DB      'HI'
                        ENDIF
3214  4849              DB      'HI'
                        ENDIF
                ;
```

```
                        ;  Another way to do it, this one less saving
                        ;  of MSTACK space.  We pass the updated count to the next
                        ;  lower macro call in a parameter.  The count for each level
                        ;  must be stored on the MSTACK, so we run out of storage
                        ;  quicker.  It does look better, though.
                        ;
                        many    MACRO
                                IF      #1>0
                                DB      #2
                                many    #1-1,#2
                                ENDIF
                                ENDM
                        ;
                                many    4,'FOO'
FFFF                            IF      4>0
3216  464F4F                    DB      'FOO'
                                many    4-1,'FOO'
FFFF                            IF      4-1>0
3219  464F4F                    DB      'FOO'
                                many    4-1-1,'FOO'
FFFF                            IF      4-1-1>0
321C  464F4F                    DB      'FOO'
                                many    4-1-1-1,'FOO'
FFFF                            IF      4-1-1-1>0
321F  464F4F                    DB      'FOO'
                                many    4-1-1-1-1,'FOO'
0000                            IF      4-1-1-1-1>0
                                DB      'FOO'
                                many    4-1-1-1-1-1,'FOO'
                                ENDIF
                                ENDIF
                                ENDIF
                                ENDIF
                                ENDIF
                        ;
                        ;       Pretty slick, huh?
                        ;
                        ;  Here is what happens if you call too many
                        ;  macros from inside a macro.  The MSTACK overflows:
                        ;
                                many    9,'WIZLE'
FFFF                            IF      9>0
3222  57495A4C                  DB      'WIZLE'
3226  45
                                many    9-1,'WIZLE'
FFFF                            IF      9-1>0
3227  57495A4C                  DB      'WIZLE'
322B  45
                                many    9-1-1,'WIZLE'
FFFF                            IF      9-1-1>0
322C  57495A4C                  DB      'WIZLE'
3230  45
                                many    9-1-1-1,'WIZLE'
FFFF                            IF      9-1-1-1>0
3231  57495A4C                  DB      'WIZLE'
3235  45
                                many    9-1-1-1-1,'WIZLE'
FFFF                            IF      9-1-1-1-1>0
3236  57495A4C                  DB      'WIZLE'
323A  45
```

```
                        many    9-1-1-1-1-1,'WIZLE'
FFFF                    IF      9-1-1-1-1-1>0
323B  57495A4C          DB      'WIZLE'
323F  45
                        many    9-1-1-1-1-1-1,'WIZLE'
FFFF                    IF      9-1-1-1-1-1-1>0
3240  57495A4C          DB      'WIZLE'
3244  45
                        many    9-1-1-1-1-1-1-1,'WIZLE'
FFFF                    IF      9-1-1-1-1-1-1-1>0
3245  57495A4C          DB      'WIZLE'
3249  45
                        many    9-1-1-1-1-1-1-1-1,'WIZLE'
        ** ERROR 2001: MSTACK overflow - Sorry! **
                        ENDIF
                        ENDIF
                        ENDIF
                        ENDIF
                        ENDIF
                        ENDIF
                        ENDIF
                        ENDIF
                ;
                ;
                ;  Now we use a macro to define two other macros.
                ;  This is primarily an exercise, but should help the
                ;  user gain confidence in the system, etc.
                ;
                define  MACRO
                first   MACRO
                        DB      'FIRST MACRO'
                        ENDM
                second  MACRO
                        DB      'SECOND MACRO'
                        ENDM
                        ENDM
                ;
                        define
                first   MACRO
                        DB      'FIRST MACRO'
                        ENDM
                second  MACRO
                        DB      'SECOND MACRO'
                        ENDM
                ;
                ;       Now the two macros should be available:
                ;
                        first
324A  46495253          DB      'FIRST MACRO'
324E  54204D41
3252  43524F
                        second
3255  5345434F          DB      'SECOND MACRO'
3259  4E44204D
325D  4143524F
                ;
                ;
                ;  In order to control the printing of macros, the
                ;  MACLIST pseudo-op can be used with an argument that
                ;  is an expression.  If the expression evaluates to FALSE
```

```
                ;  (0000H), then macros aren't printed.  Otherwise, a TRUE
                ;  value (TRUE!=0000H) will list them.  Inside a macro,
                ;  it is just as if a LIST pop had been used with the same
                ;  argument as was on the MACLIST pop.  Indeeed, the LIST
                ;  pop can be used to change this default for a particular
                ;  macro by including it in the macro body.  The value of
                ;  the LISTing is saved on entry to macro expansion and
                ;  is restored to its previous value on exit.
                ;
                listxmpl  MACRO
                          DB      'HI THERE'
                          ENDM
                ;
                          listxmpl
3261  48492054            DB      'HI THERE'
3265  48455245
                ;
                          MACLIST    FALSE
                          listxmpl
                          MACLIST     TRUE
                ;
                ;  Now for some examples of the format of the
                ;  parameters in a macro reference.  There are two kinds
                ;  of parameters in the operand field of a macro invocation
                ;  line:
                ;
                ;       1)  A string of characters delimited by a comma,
                ;       right bracket, or semicolon.  The actual string of
                ;       characters substituted in the macro has blanks
                ;       stripped from the ends.  Any of the three special
                ;       characters can be included within a quoted character
                ;       string without delimiting the param.
                ;
                ;       2)  A bracketed parameter.  A pair of outside brackets
                ;       surrounding a string of characters which has a properly
                ;       matched set of brackets inside (or no inside brackets
                ;       at all).  Commas and semicolons are also allowed.
                ;       Inside a quoted character string ( 'hi[],;') brackets
                ;       need not be nested properly.  The outside brackets
                ;       are not passed on into the macro, but the spaces inside
                ;       the brackets are passed on (if there is any space).
                ;
                ;       Here are some parameter examples:
                ;
                params  MACRO
                ;#1
                ;#2
                ;#A
                          ENDM
                          params  1,2
                ;1
                ;2
                ;1,2
                          params  [ 1 ][ 2 ]
                ; 1
                ; 2
                ; 1 , 2 ,
                          params  [A,B,C],[D,E,F]
                ;A,B,C
                ;D,E,F
```

```
                ;A,B,C,D,E,F,
                        params  [A[B]C],D
                ;A[B]C
                ;D
                ;A[B]C,D
                        params  [A]B
                ;A
                ;B
                ;A,B
                        params  A[B]
                ;A
                ;B
                ;A,B,
                        params  ['[;,'],A
                ;'[;,'
                ;A
                ;'[;,',A
                ;
                ;  There are two parameter functions in MACRO-88 version
                ;  002.  The LEN[] function returns the number of bytes in
                ;  memory that would be taken up if its argument were
                ;  submitted to a DB pseudo-op.  LEN[] can be used anywhere
                ;  an expression is permitted.  Its value is 16 bits long, but
                ;  can be shortened as any other 16 bit value can by putting
                ;  it in a DB pop.  The NULL[] function checks its argument
                ;  for existence.  The argument can be one or more comma
                ;  separated Parms, each constructed according to the rules
                ;  as described above.  If any parm can be extracted from
                ;  NULL[]'s arguments, the value is 0000H.  Otherwise, the
                ;  value is 0FFFFH.
                ;
                ;  Here is an example which builds entries in the MACRO-88
                ;  initial symbol table.  An entry is specified by the 16-bit
                ;  total length, the 8-bit name length, the ASCII name, the
                ;  type code byte, and the variable length data field, which
                ;  is 16 bits except for macro entries.  The macro we use
                ;  accepts just the name, typecode, and value.
                ;
0000            POPGEN  SET     0               ;Popcode key generator.
                ;
0001            OPCTYPE EQU     1
0002            POPTYPE EQU     2
0008            MACTYPE EQU     8
                ;
                symel   MACRO   Name,Typecode,Value{,Subtype}
                        IF      #2=OPCTYPE
                        DW      LEN['#1']+7
                        DB      LEN['#1'],'#1',OPCTYPE
                        DW      #3
                        IF      NOT NULL[#4]
                        DB      #4
                        ELSE
                        DB      0
                        ENDIF
                        ELSEIF  #2=POPTYPE              ;Pop type
                POPGEN  SET     POPGEN+1
                #1CODE  EQU     POPGEN
                        DW      LEN['#1']+6
                        DB      LEN['#1'],'#1',POPTYPE
                        DW      #1CODE
```

```
                        ELSEIF  #2=MACTYPE
                        DW      LEN['#1']+LEN[#3]+4
                        DB      LEN['#1'],'#1',MACTYPE
                        DB      #3
                        ENDIF
                        ENDM
                ;
                        symel   PRBASE,POPTYPE
0000                    IF      POPTYPE=OPCTYPE
                        DW      LEN['PRBASE']+7
                        DB      LEN['PRBASE'],'PRBASE',OPCTYPE
                        DW
                        IF      NOT NULL[]
                        DB
                        ELSE
                        DB      0
                        ENDIF
FFFF                    ELSEIF  POPTYPE=POPTYPE         ;Pop type
0001            POPGEN  SET     POPGEN+1
0001            PRBASECODE      EQU     POPGEN
3271  0C00              DW      LEN['PRBASE']+6
3273  06505242          DB      LEN['PRBASE'],'PRBASE',POPTYPE
3277  41534502
327B  0100              DW      PRBASECODE
                        ELSEIF  POPTYPE=MACTYPE
                        DW      LEN['PRBASE']+LEN[]+4
                        DB      LEN['PRBASE'],'PRBASE',MACTYPE
                        DB
                        ENDIF
                        symel   wow,MACTYPE,[' MVI A,''!''',CR,' CALL WH0',CR]
0000                    IF      MACTYPE=OPCTYPE
                        DW      LEN['wow']+7
                        DB      LEN['wow'],'wow',OPCTYPE
                        DW      ' MVI A,''!''',CR,' CALL WH0',CR
                        IF      NOT NULL[]
                        DB
                        ELSE
                        DB      0
                        ENDIF
0000                    ELSEIF  MACTYPE=POPTYPE         ;Pop type
                POPGEN  SET     POPGEN+1
                wowCODE EQU     POPGEN
                        DW      LEN['wow']+6
                        DB      LEN['wow'],'wow',POPTYPE
                        DW      wowCODE
FFFF                    ELSEIF  MACTYPE=MACTYPE
327D  1C00              DW      LEN['wow']+LEN[' MVI A,''!''',CR,' CALL WH0',CR]+4
327F  03776F77          DB      LEN['wow'],'wow',MACTYPE
3283  08
3284  204D5649          DB      ' MVI A,''!''',CR,' CALL WH0',CR
3288  20412C27
328C  21270D20
3290  43414C4C
3294  20574830
3298  0D
                        ENDIF
                        symel   GET,OPCTYPE,2
FFFF                    IF      OPCTYPE=OPCTYPE
3299  0A00              DW      LEN['GET']+7
329B  03474554          DB      LEN['GET'],'GET',OPCTYPE
```

```
329F  01
32A0  0200              DW      2
0000                    IF      NOT NULL[]
                        DB
                        ELSE
32A2  00                DB      0
                        ENDIF
                        ELSEIF  OPCTYPE=POPTYPE         ;Pop type
                POPGEN  SET     POPGEN+1
                GETCODE EQU     POPGEN
                        DW      LEN['GET']+6
                        DB      LEN['GET'],'GET',POPTYPE
                        DW      GETCODE
                        ELSEIF  OPCTYPE=MACTYPE
                        DW      LEN['GET']+LEN[2]+4
                        DB      LEN['GET'],'GET',MACTYPE
                        DB      2
                        ENDIF
                ;
                        END
```

```
Error total =     1

        Macros defined in this assembly:

bizzarre    define      first       flap
listxmpl    many        params      repeat
repeatl     second      symel

        Set variables used:

COUNT       0000 FALSE      0000 POPGEN     0001 TRUE       FFFF

        Labels defined in this assembly:

CR          000D LAB1       320B MACTYPE    0008 OPCTYPE    0001
POPTYPE     0002 PRBASECODE 0001
```