Base Ten Systems

MPL LANGUAGE REFERENCE MANUAL

M6800 MICROPROCESSOR

Motorola Microsystems
3102 N. 56th Street
Phoenix, Arizona 85018

MPL LANGUAGE

For The

M6800 MICROPROCESSOR

PRELIMINARY

MARCH 1976

Second Edition
Copyright © 1976 by Motorola Inc.

TABLE OF CONTENTS

MPL MANUAL

Preface

This compiler expands the Total Product Offering for the M6800 Microprocessor Family of Parts. The compiler was developed to meet the varied needs of microprocessor users and in particular the output of the compiler is an assembly language file (rather than machine language output). This shows the user the code generated on a compiler statement by compiler statement basis; which in turn gives the user closed loop feedback on his programming techniques.

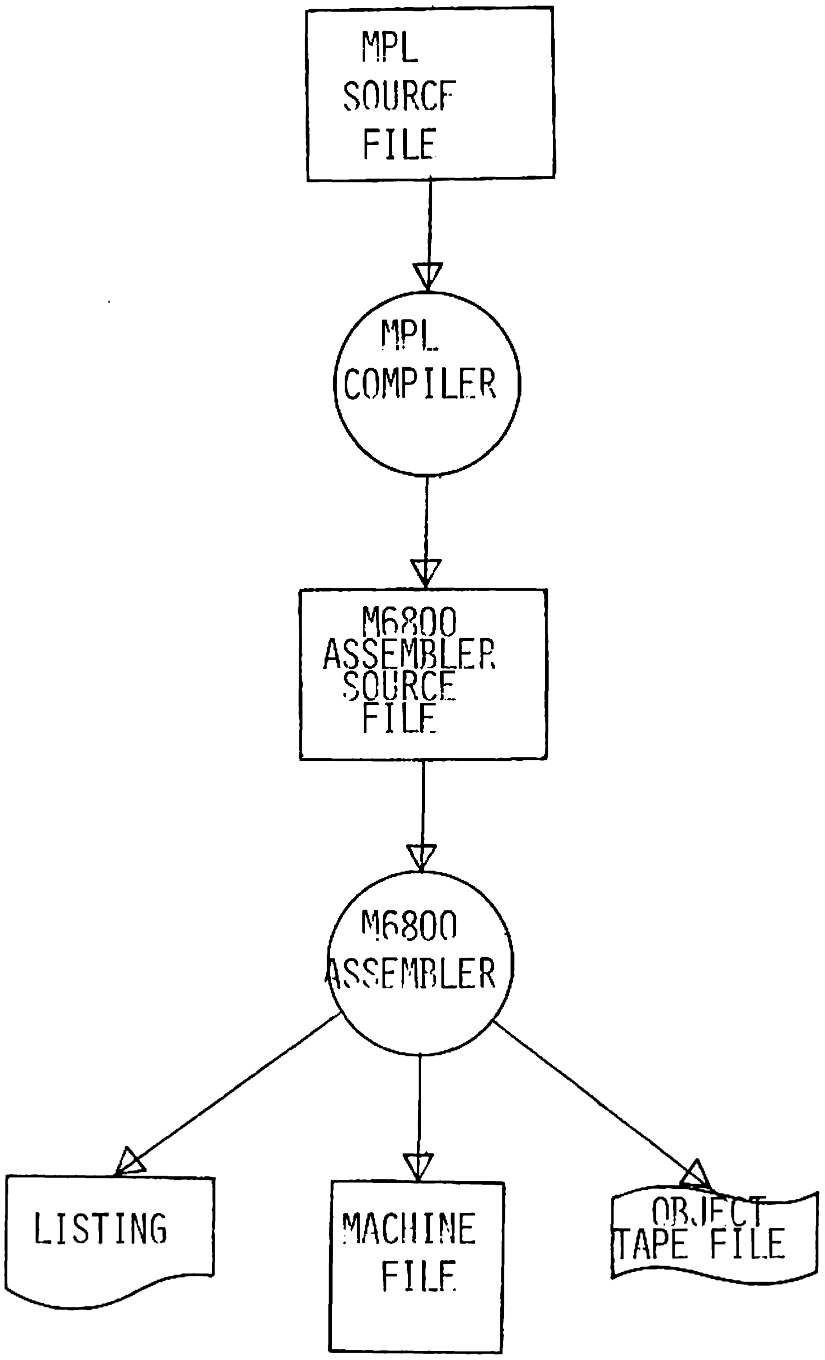
Compilers have not been software panaceas in the past, nor do we expect this compiler to solve all your programming problems. However, in the hands of a serious programmer, this compiler should serve as an effective software tool.

INTRODUCTION

The MPL language is a compiler language that is especially useful in writing programs for applications that involve mathematical computations and manipulation of numerical and string data. The language is also especially suitable for realtime microprocessor applications.

Source programs written in the MPL language consist of a set of statements constructed by the programmer from the language elements described in this publication.

In a process called compilation, a program called the MPL Compiler analyzes the source program statements and translates them into an assembly language program called the object program, which will be suitable for assembly by an M6800 Assembler. In addition, when the MPL Compiler detects errors in the source program, it produces appropriate diagnostic error messages.



COMPILER-BLOCK DIAGRAM

ELEMENTS OF THE LANGUAGE

STATEMENTS

Source programs consist of a set of statements from which the compiler generates machine instructions, constants, and storage areas. A given MPL statement effectively performs one of three functions:

1. Causes certain operations to be performed (e.g., add, multiply, branch)
2. Specifies the nature of the data being handled.
3. Specifies the characteristics of the source program.

MPL statements usually are composed of certain MPL key words used in conjunction with the basic elements of the language: constants, variables, and expressions. The categories of MPL statements are as follows:

A. Arithmetic Statements: These statements cause calculations to be performed and cause the result to replace the current value of a designated variable or subscripted variable.

B. Control Statements: These statements enable the user to govern the flow and terminate the execution of the object program.

C. Specification Statements: These statements are used to declare the properties of variables and arrays (such as type and amount of storage reserved).

D. Procedure Statements: These statements enable the user to name and define procedures, which can be compiled separately or with the main program.

The basic elements of the language are discussed in this chapter. The actual MPL statements in which elements are used are discussed in the following sections. The term procedure refers to a main program or a subprogram. The phrase executable statements refers to those statements in categories A, B, and D above.

The order of an MPL program unit is:

1. Procedure statement.
2. Specification statements, if any. (Explicit specification statements which initialize variables or arrays must appear in the same specification statements that define the variable or array name.)
3. Executable statements, at least one of which must be present.
4. END statement.

CODING MPL STATEMENTS

The statements of an MPL source program may be punched on cards or typed on a TTY terminal with each line on the TTY terminal representing one 80-column card. MPL statements are written within columns 1 through 72. If a statement is too long for one card, it may be continued on successive cards except for any valid constant. All constants must be completely contained on the same card or line.

As many blanks as desired may be written in a statement to improve its readability. Multiple blanks are ignored by the compiler. Blanks, however, that are inserted in literal data are retained and treated as blanks within the data.

It should also be noted however that the connectors and logical operators - NOT, AND, OR, GT, GE, EQ, NE, LT, LE, IAND, IOR, IEOR, SHIFT - must be preceded and followed by at least one blank. Other reserved words must be preceded and followed by a blank or some other delimiter. Labels, variable names and procedure names must not contain blanks.

The first card of a statement may contain a statement label consisting of from 1 through 6 alphabetic or numeric characters, the first of which must be alphabetic. Statement labels must be followed by a colon (:) symbol. If column 1 contains a dollar sign ($), the card is considered to be an assembly language card and is passed to the symbolic output file without being acted upon by the compiler.

Columns 73 through 80 are not significant to the MPL compiler and may therefore, be used for program identification, sequencing, etc.

Multiple statements may appear on one line, separated by blanks or by a semicolon(;).

Comments to explain a program may be written either by:

1. Enclosing the comments within /* and */ delimiters. Such comments may occur anywhere and may extend over several lines.

or

2. Terminating one or more statements on a line by an exclamation character (!) and following this by comments. The compiler considers all information after the ! to be comments. It is also possible to place the ! in column one if it is desired to make the entire card to be a comment.

DATA REPRESENTATION

Data may be represented in MPL in the following formats:

1. Bit string:

   BIT(1) - BIT(7) are one bit fields which may be tested, set or cleared. The most significant bit, bit 7, is the only bit used for the BIT (1) declaration. BIT(8) is used for an 8-bit field (one byte) which can have the following operations performed on it:

   A. Test (magnitude only)

   B. Shift (rotate shift)

   C. Logical AND

   D. Logical OR

   E. Logical EOR

2. Single or double precision integer binary: BINARY(1) or BINARY(2). One or two bytes are occupied respectively and the following operations may be performed:

   A. Test

   B. Shift (arithmetic right & left)

   C. Logical AND

   D. Logical OR

   E. Logical EOR

   F. Mulitply

   G. Divide

   H. Add

   I. Subtract

   J. Replace

3. ASCII numeric: DECIMAL(1), DECIMAL(2) ... DECIMAL(12), SIGNED DECIMAL(1), SIGNED DECIMAL(2) ... SIGNED DECIMAL(12), DECIMAL(m,n) or SIGNED DECIMAL(m,n) where _m_ indicates the total number of bytes (digits) in the representation and _n_ indicates the number of bytes (digits) after the assumed decimal point. _m_ and _n_ must not be greater than 12. This representation occupies one byte for each digit plus one for an optional sign and may be used in the following operations:

   A. Test

   B. Add

   C. Subtract

   D. Replace

4. ASCII alphanumeric: CHARACTER(m) occupies m bytes, one for each character and $1 \leq m \leq 255$.

The CHARACTER representation may be tested and replaced.

CONSTANTS

A constant is a fixed, unvarying quantity. Five types of constants can be used: integer, binary, hexadecimal, string, and address. The type of a constant is defined by its implicit type and by its usage. For example, the value 123 may have such representations as BINARY(1), BINARY(2) or DECIMAL(3) depending upon the environment of its usage.

INTEGER CONSTANTS

Definition

Integer Constant - a whole number written without a decimal point. It occupies one or two locations of storage for a binary constant and N bytes for a numeric ASCII constant where $1 \leq N \leq 12$.

Maximum magnitude for binary constant: $\pm$ 127 for one byte
$\pm$ 32767 for two bytes

An integer constant may be positive, zero, or negative; if unsigned, it is assumed to be positive. Its magnitude must not be greater than the maximum for the given representation implied by its environment in a statement and it may not contain embedded commas.

Examples:

Valid integer constants:

0

+91

91

173

-21474

Invalid integer constants:

3145903612	(exceeds the allowable range of a binary constant)
5,396	(contains an embedded comma)

BINARY CONSTANTS

Definition

Binary constant - a string of 0 and 1 bits followed by the letter B. A binary constant may occupy one or two bytes. If less than 8 or 16 bits are specified, they will be right justified in the one or two bytes required to contain the constant.

Examples:

01001011B

1001110010010111B

1011B

10011011011B

HEXADECIMAL CONSTANTS

Definition

Hexadecimal constant -- a hexadecimal number (0-9, A-F) enclosed in double quotes or not enclosed in quotes and followed by the letter H. In the latter case the first character must be numeric. The maximum magnitude is $FFFF_{16}$. One or two bytes are occupied.

Examples:

"0" or OH	"FFFF" or 0FFFFH	"5A4C" or FA4CH
"3F" or 3FH	"D24" or OD24H	"6F" or 6FH

STRING CONSTANTS

Definition

String constant -- a series of ASCII characters enclosed in quote signs.

It occupies one byte for each character and must be less than 256 bytes long. A maximum of 47 characters can be used to the right of an equal sign.

Examples:

'TOTAL'

'VALUE IS' - The full 64 ASCII character set may be used. The exclamation point (!), and apostrophe (') can be used by using two

AA='!!''' will set AA=!'

ADDRESS CONSTANT

Definition

Address constant -- a series of ASCII characters enclosed in quote marks representing the value of an address. It occupies two bytes and must have a magnitude less than 65535.

Examples:

'TABLE' - address of TABLE

'TABLE+50*10' - address of TABLE+50*10

An address constant is assumed if the data representation is BINARY(2) and the constant name is enclosed in single quotes.

SYMBOLIC NAMES

Definition:

Symbolic Name -- consists of from one through six alphameric characters i.e., numeric (0 through 9) or alphabetic (A through Z), the first of which must be alphabetic. In addition, keywords reserved by the compiler or assembler should not be used as symbolic names. The following names are reserved keywords:

A	GE	OR
AND	GO	ORIGIN
B	GOTO	PROC
BASED	GT	PROCEDURE
BEGIN	IAND	RETURN
BIT	IEOR	SHIFT
BY	IF	SIGNED

CALL	INIT	T
DCL	INITIAL	THEN
DECLARE	IOR	TO
DEF	LABEL	WHILE
DEFINED	LE	X
DO	LT	ZOOO
ELSE	MAIN	ZOO1
END	NE	.
EOR	NOT	.
EQ	OPTIONS	ZFFF

Note: All 4 character symbolic names whose first character is Z are reserved.

Symbolic names are used in a procedure to identify elements in the following classes.

. An array and the elements of that array (see "Arrays")

. A variable (see "Variables")

. A statement label

. A procedure name

Symbolic names must be unique within a program (consisting of a main procedure and optional sub-procedure(s)) and can identify elements of only one class.

VARIABLES

An MPL variable is a symbolic representation of a quantity that occupies a storage area. The value specified by the name is always the current value stored in the area. For example, in the following statement both W and Y are variables:

```
W = 5 + Y
```

The value of Y is determined by some previous statement and may change from time to time. The value of W is calculated whenever this statement is executed and changes as the value of Y changes.

VARIABLE NAMES

The use of meaningful variable names can serve as an aid in documenting a program. That is, someone other than the programmer may look at the program and understand its function. For example, to compute the distance a car traveled in a certain amount of time at a given rate of speed, the following statement could have been written:

```
W = Y * Z
```

where * designates multiplication. However, it would be more meaningful to someone reading this statement if the programmer had written:

```
DIST = RATE * TIME
```

Examples:

Valid variable names:

B292S

RATE

Invalid variable names:

B292704	(Contains more than six characters)
4ARRAY	(First character is not alphabetic)
SI.X	(Contains a special character)

VARIABLE TYPES AND LENGTHS

The type of a variable corresponds to the type of data the variable represents. Thus, a binary variable (BINARY) represents a binary data, a bit string variable (BIT) represents bit string data, a numeric ASCII variable (DECIMAL) represents numeric ASCII data, and a literal string variable (CHARACTER) represents alphanumeric ASCII data.

The number of storage locations reserved for the variable depends on the type of the variable.

A programmer must declare the type of a variable by using the DECLARE statement prior to the first usage of the variable.

ARRAYS

An MPL array is a set of variables identified by a single variable name. A particular variable in the array may be referred to by its position in the array (e.g., first variable, third variable, seventh variable, etc.). Consider the array named NEXT which consists of five variables, each currently representing the following values: 273, 41, 8976, 59, and 2.

NEXT(1) is the location containing 273

NEXT(2) is the location containing 41

NEXT(3) is the location containing 8976

NEXT(4) is the location containing 59

NEXT(5) is the location containing 2

Each variable (element) in this array consists of the name of the array (i.e., NEXT) immediately followed by a number enclosed in parentheses, called a subscript quantity. The variables that the array comprises are called subscripted variables. Therefore, the subscripted variable NEXT(1) has the value 273; the subscripted variable NEXT(2) has the value of 41, etc.

The subscripted variable NEXT(I) refers to the "Ith" subscripted variable in the array, where I is an integer variable that may be assigned a value of 1, 2, 3, 4, or 5.

To refer to any element in an array, the array name must be subscripted. The array name alone represents the entire array.

Consider the following array named LIST described by two subscript quantities, the first ranging from 1 through 5, the second from 1 through 3.

	Column 1	Column 2	Column 3
ROW 1	82	4	7
ROW 2	12	13	14
ROW 3	91	1	31
ROW 4	24	16	10
ROW 5	2	8	2

The array would be defined as:

```
     DECLARE
       01 TABLE,
          02 ROW(5),
             03 LIST(3) BINARY(1)
```

or as:

```
     DECLARE LIST(5,3)
```

Reference to the number in row 2, column 3 would be coded as:

```
     LIST (2,3)
```

Thus, LIST (2,3) has the value 14 and LIST (4,1) has the value 24.

Ordinary mathematical notation uses $LIST_{i,j}$ to represent any element of the array LIST. In MPL, this is written as LIST (I,J) where I equals 1, 2, 3, 4 or 5, and J equals 1, 2 or 3.

An additional possibility of referring to elements of an array is made available in MPL if the first of the two methods of declaring is used. ROW(2) then refers to the data in all three columns of row 2 and TABLE refers to all elements of the array.

DECLARING THE SIZE AND TYPE OF AN ARRAY

The size (number of elements) of an array is specified by the number of subscript quantities of the array and the maximum value of each subscript quantity. This information must be given for all arrays before using them in a program so that an appropriate amount of storage may be reserved. Declaration of this information is made by a DECLARE statement. This statement is discussed in detail in the chapter "DECLARE" statement. The type of an array name is determined by the specification for the type of the variable name. Each element of an array is of the type specified for the array name, but need not be the same type for all levels of the structure. The above example could be declared as:

```
01 TABLE,
   02 ROW(5),
      03 COL1 BINARY(1),
      03 COL2 BINARY(2),
      03 COL3 DECIMAL(5)
```

if it is desired to have varying data representations for each column. Within one column however the data representation is fixed.

SUBSCRIPTS

A subscript is an integer subscript quantity or a set of integer subscript quantities separated by commas, which is used to identify a particular element of an array. The number of subscript quantities in any subscript must be the same as the number of dimensions of the array with which the subscript is associated. A subscript is enclosed in parentheses and is written immediately after the array name. A maximum of three subscript quantities can appear in a subscript.

General Form

Subscript Quantities -- may be one of four forms:

v

k

v+k

v-k

Where: v represents an unsigned, nonsubscripted, integer BINARY(1) variable and must be level 01

k represents an unsigned integer constant.

Whatever subscript form is used, its evaluated result, as well as the intermediate result, must always be greater than or equal to 1 and less than or equal to 255. For example, when reference is made to the subscripted variable V(I-2), the value of I must be greater than or equal to 3 and less than or equal to 255. In any case, the evaluated result must be within the range of the array.

Examples:

Valid subscripted variables:

```
ARRAY  (IHOLD)

NEXT (19)

MATRIX (I-5)

Z(I+3,J+8,K)
```

Invalid subscripted variables:

ARRAY(-I)	(The subscript quantity I may not be signed)
ARRAY(I+2.)	(The constant within a subscript quantity must be an integer with no associated decimal point)
NEXT(-7+J)	(If subtraction is indicated, the variable must precede the constant)
W(I(2))	(The subscript quantity I may not itself be subscripted)
TEST(K*2)	(Multiplication is indicated which is an error)
TOTAL(2+K)	(If addition is indicated, the variable must precede the constant. Thus TOTAL (K+2) is correct)
Q(I,J,K,L)	(No more than three subscript quantities may be used)

POINTERS

The evaluation of subscripts generally requires multiplication. Since the M6800 does not have a multiplication instruction, an alternate form of array addressing is provided through the use of pointers. A pointer must be declared with a representation BINARY(2). It must contain the address of the level 01 entry of the item being referenced. An item is pointed by the form

```
V:P    or    P->V
```

where V represents a simple or subscripted variable and P represents a simple variable of type BINARY(2), level 01.

Valid pointed variables

ARRAY:PTR	or	PTR -> ARRAY
NEXT(19):JJ	or	JJ -> NEXT(19)
MATRIX(I-5):JJ	or	JJ -> MATRIX(I-5)

Invalid pointed variables

ARRAY:PTR(3)	(pointer may not be subscripted)
NEXT(19):JJ-2	(would be evaluated as (NEXT(19):JJ)-2)

Additionally, pointer variables are useful in handling linked lists. For example:

```
DECLARE
01 TABLE (100),
   02 LINK BINARY(2),
   02 A1 DECIMAL(3),
   02 A2 CHARACTER(5)
```

The pointer for the next entry in the chain may be obtained by the statement:

```
XX=LINK:XX
```

The following coding will link the new entry pointed to by "NEXT" into the chain. Assume the variable "PTR" points to the first entry in the chain:

```
          ZZ = 'PTR'
     L1   XX = ZZ
          ZZ = LINK:XX
          IF ZZ NE Ø AND A1:NEXT GE A1:ZZ
                THEN GO TO L1
          LINK:XX = NEXT
          LINK:NEXT = ZZ
```

Removing (unchaining) the entry that has a value of A1 equal to 35 could be accomplished by:

```
          ZZ = 'PTR'
     L1:  XX = ZZ
          ZZ = LINK:XX
          IF A1:ZZ NE 35 THEN GO TO L1
          LINK:XX = LINK:ZZ
```

EXPRESSIONS

MPL provides two kinds of expressions: arithmetic and logical. Expressions may appear in arithmetic assignment statements and in IF statements.

ARITHMETIC EXPRESSIONS

The simplest arithmetic expression consists of a primary that may be a single constant, variable, subscripted variable, or another expression enclosed in parentheses. The primary may be either type BIT, BINARY, DECIMAL or CHARACTER.

If the primary is of BIT type, the expression is BIT type. If it is of DECIMAL type, the expression is of DECIMAL type etc.

Examples:

Primary	Type of Primary	Type of Expression
3	BINARY	BINARY
AA	CHARACTER(5)	CHARACTER(5)
C(I+2)	DECIMAL(3) subscripted variable	DECIMAL(3)
(R+S*T)	Parenthesized expression	Same as R,S, and T

In the expression C(I+2), the subscript (I+2), which must always represent integer binary, does not affect the type of the expression. That is, the type of the expression is determined solely by the type of primary appearing in that expression.

More complicated arithmetic expressions containing two or more primaries may be formed by using arithmetic operators that express the computation(s) to be performed.

Arithmetic and Logical Operators

The arithmetic and logical operators are as follows:

Arithmetic Operator	Definition
*	Multiplication
/	Division
+	Addition
-	Subtraction
SHIFT	Shift
IAND	Logical AND
IOR	Logical OR
IEOR	Logical EOR

RULES FOR CONSTRUCTING ARITHMETIC EXPRESSIONS: The following are the rules for constructing arithmetic expressions that contain arithmetic operators:

1. All desired computations must be specified explicitly. That is, if more than one primary appears in an arithmetic expression, they must be separated from one another by an arithmetic operator. For example, the two variables W and Y will not be multiplied if written:

   ```
   WY
   ```

   If multiplication is desired, then the expression must be written as follows:

   ```
   W*Y
   ```

2. No two arithmetic operators may appear in sequence in the same expression. For example, the following expressions are invalid:

```
W*/Y and W IAND *Y
```

An exception is the unary minus:

```
W*-Y
```

In effect, -Y will be evaluated first and then W will be multiplied by the result.

A shift operand is written as:

```
W SHIFT k
```

where k represents a positive or negative constant number of bits to shift the variable W and $-8 \leq K \leq 8$. A positive value represents a left shift, while a negative value represents a right shift. Shifts are arithmetic for variables having data representation of BINARY(1). If the variable has a data representation of BIT(1), the shift operation will be circular. That is, bit 7 will be rotated to the right on a right shift, negative value, and rotated left on a left shift through the whole byte.

```
   R = R SHIFT 2
Z(3) = D(3) SHIFT -6
```

3. Order of Computation: Computation is performed from left to right according to the hierarchy of operations shown in the following list.

OPERATION	HIERARCHY
unary -	1
SHIFT	2
logical AND - IAND	3
logical OR - IOR	3
logical EOR - IEOR	3
multiply	4
divide	4
add	5
subtract	5

This hierarchy is used to determine which of two consecutive operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If it is not, the second operator is compared to the third, etc. When the end of the expression is encountered, all of the remaining operations are performed in reverse order.

For example, in the expression V*Y+Z*W IAND I, the operations are performed in the following order:

1.	V*Y	Call the result R (multiplication)	(R+Z*W IAND I)
2.	W IAND I	Call the result S (logical AND)	(R+Z*S)
3.	Z*S	Call the result T (multiplication)	(R+T)
4.	R+T	Final operation (addition)	

A unary plus or minus has the highest priority.

    C = -D is treated as C=O-D

    C =-D*E is treated as C=(O-D)*E

    C =-D+E is treated as C=(O-D)+E

Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be computed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used. This is equivalent to the definition above since a parenthesized expression is a primary.

For example, the following expression:

    D+((C+D)*E)+C IAND 2

is effectively evaluated in the following order:

1.	(C+D)	Call the result R	D+(R*E)+C IAND 2
2.	(R*E)	Call the result S	D+S+C IAND 2
3.	D+S	Call the result T	T+C IAND 2
4.	C IAND 2	Call the result V	T+V
5.	T+V	Final operation	

4. The type of the result of an operation depends on the type of the two operands (primaries) involved in the operation. All variables within an expression must be of the same type.

LOGICAL EXPRESSIONS

A logical expression consists of two arithmetic expressions (which may of course be simple variables) connected by one of the following relational operations:

EQ	- equal
NE	- not equal
GT	- greater than
LT	- less than
GE	- greater than or equal to
LE	- less than or equal to

Examples:

```
C EQ C
C IAND "3F" NE 21
(C+D)*E GT 50
```

It should be clearly understood here that arithmetic expressions involved in relational operations are evaluated first before the relational operation is applied. See example in item 3, Order of Computation above.

Relational operations in turn may be connected by the use of the logical connectives AND and OR:

C EQ D OR E EQ F

C NE D AND E GT F OR G EQ H

Normally AND operations have a higher hierarchy than OR operations, thus C EQ D AND E GT F OR G EQ H is evaluated as

(C EQ D AND E GT F) OR G EQ H

However parentheses may be used to change the order of evaluation -

C EQ D AND (E GT F OR G EQ H)

Additionally, the meaning of a logical operation may be reversed by the modifier NOT

Example:

NOT W EQ Y

means W NE Y

NOT (W EQ Y AND Z EQ V)

means everything but the intersection of W EQ Y AND Z EQ V

STATEMENTS

The following sections describe the statement types that are available in the MPL language.

THE ORIGIN STATEMENT

General form

```
ORIGIN  "hex constant"
```

The ORIGIN statement is used to reset the assembly address for the subsequent statements. It may appear anywhere in the program.

Example:

```
ORIGIN "3F2E"
```

The subsequent statements will be assembled starting at hex address 3F2E.

Default starting address is zero.

THE DECLARE STATEMENT

The general form is:

```
DECLARE
                                          ┌BIT             ┐
                                          │BINARY          │  ┌(m)  ┐
                                          │DECIMAL         │  │     │[DEFINED name]
[level #] name [(occurrence)]             │SIGNED DECIMAL  │  │(m,n)│
                                          │CHARACTER       │  └     ┘
                                          └LABEL           ┘
 [BASED] [INITIAL (value 1, value 2 ... )]
```

Forms in square brackets are optional. The following abbreviations are allowed:

DCL - DECLARE	CHAR - CHARACTER
BIN - BINARY	DEF - DEFINED
DEC - DECIMAL	INIT - INITIAL

Notes

1. Level number is optional, but if not used is assumed to be 01. Basic items are level 01 explicitly or implicitly. Elementary items have successively higher level numbers and must be in sequence and may not be greater than 05. Skipping a level number is not allowed. For example:

```
        DECLARE
       01   AA,
            02  BB,
                03  CC,     (indentation of levels for
                03  DD,      clarification only and are
            02  EE,          not required)
        01  FF
```

is valid, while

```
        DECLARE
        01  AA,
            02  BB,
                04  CC
```

is invalid since level 03 is skipped.

Each level except the last one must terminate with a comma.

2. Name represents any valid MPL name.

3. If occurrence is not used then it is assumed to be 1. Multiple occurrences may be used for arrays with more than one dimension.

   Example:

```
             AA
             RTS(8)
             XYZ(10,12)
```

4. If BIT, BINARY etc. is not used then the data representation is assumed to be BINARY(1) unless there are other levels defined within the current level.

   For example in the following statements:

```
            DECLARE
            01  AA,
                02  BB  DEC(3),
                02  CC  DEC(4),
            01  RR,
            01  TT(10,4 BIN(2)
```

   RR will have a data representation of BINARY(1) since there is no explicit data representation given. AA on the other hand must not have a data representation since there are other levels defined within AA.

5. m refers to the length of the data representation - 2 bits, 4 bytes etc. n refers to the number of digits after the decimal point in the case of DECIMAL or SIGNED DECIMAL data representations. If m is not used then its value is assumed to be 1.

   Permissible values for m and n are as follows:

Representation	m	n
BIT	1 ∿ 8	not used
BINARY	1 ∿ 2	not used
DECIMAL	1 ∿ 12	1 ∿ 12
CHARACTER	1 ∿ 256	not used
LABEL	not used	not used

6. LABEL is used in conjunction with a computed GO TO statement or an assigned GO TO statement; see assigned and computed GO TO statement descriptions for an example. Field definitions (m,n) DEFINED and BASED clauses may not occur with LABEL.

7. DEFINED name is used to redefine a previously used name at the same level.

   In any data structure, the name being redefined must be the last name used at the same level. For example:

```
DECLARE
01  XX,
01  YY,
    02  AA,
            03  RR,
            03  SS,
    02  BB          DEFINED  AA
```

is valid, since AA is the last previous occurrence at the 02 level

```
DECLARE
01  XX,
    02  YY,
    02  ZZ,
    02  WW  DEFINED  YY
```

is invalid since ZZ is the last previous occurrence at the 02 level.

8. BASED is used when a data structure is being defined which will not result in space being allocated by the computer.

9. INITIAL (value 1, value 2 ...) is used to initialize variables. The INITIAL statement may not be used if DEFINED has also been used.

   Example:

```
DECLARE RATE DEC(2)
DCL
01 PIA1A,
     02 CRTRDY BIT(1),
     02 CRTDWN BIT(1)

DECLARE
01   FLAG1,
     02 BFLAG1 BIN(1),
     02 XFLAG1 BIN(1) DEFINED BFLAG1
```

```
DECLARE
01    STRUCT(10),
      02 ITEM CHARACTER(5),
      02 AMT,
          03 DLRS DECIMAL(2),
          03 CENTS DECIMAL(2),
      02 USAGE(5)

DECLARE TBL(5) INITIAL(5,3,2,1,9)
DCL ENDP CHAR(3) INITIAL('END')
```

String constants used in conjunction with LABEL need not have single quotes.

For example:

```
DCL JTAB(3) LABEL INITIAL (A20,A70,A110)
```

where A20, A70, and A110 are labels used in the program.

ARITHMETIC ASSIGNMENT STATEMENT

General form

$\underline{a} = \underline{b}$

Where: $\underline{a}$ is a subscripted or nonsubscripted variable, or a series of comma separated subscripted or nonsubscripted variables

$\underline{b}$ is an arithmetic expression.

This MPL statement closely resembles a conventional algebraic equation when used in its simplest form. However, the equal sign specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable or variables to the left of the equal sign.

The type of the variable(s), represented by a, is converted according to the type of the arithmetic expression b, as shown in Table 2.

Type of b \ Type of a	BIN	DEC	CHAR
BIN	Assign	Convert to numeric ASCII and assign	Convert to numeric ASCII with zero suppression and assign, right justified, blanked filled on left
DEC	Convert to binary and assign	Assign	Zero suppress and assign
CHAR	Not allowed	Not allowed	Assign a>b left justify blank fill a b>a truncate b on the right and assign

TABLE 2

Assume that the data representation of several variables has been specified as follows:

Variable Names	Type
I, J, W	BINARY(1) variables
C, D, E	DECIMAL(3) variables
EE	CHARACTER(5) variables
F (5,5)	BINARY(2) array

Then the following examples illustrate valid arithmetic statements using constants, variables, and subscripted variables of different types:

Statements	Description
C = D	The value of C is replaced by the current value of D.
W = D	The value of D is converted to binary, and this value replaces the value of W.
C = I	The value of I is converted to an ASCII field, and this result replaces the value of C.
I = I + 1	The value of I is replaced by the value of I + 1.
C = D + E	The sum of D and E replaces the value of C.

Statements	Description
C = F(5,4)	The value of F(5,4) is converted to numeric ASCII and replaces the value of C.
J = EE	Not allowed.
F(2,3) = C	The value of C is converted to double precision binary, and this value replaces the value of F(2,3).
C,D,E = W	The current value of W is converted to numeric ASCII and is used to replace the values of C, D, and E.

CONTROL STATEMENTS

Normally, MPL statements are executed sequentially. That is, after one statement has been executed, the statement immediately following it is executed. This section discusses the statement that may be used to alter and control the normal sequence of execution of statements in the program.

GO TO STATEMENTS

GO TO statements permit transfer of control to an executable statement specified by number in the GO TO statement. Control may be transferred either unconditionally or conditionally. The GO TO statements are:

1. Unconditional GO TO statement
2. Assigned GO TO statement
3. Computed GO TO statement

Note that in all three types GO TO may be replaced by GOTO.

Unconditional Go To Statement

General Form

GO TO xxxxx

Where: xxxxx is a label on an executable statement.

This GO TO statement causes control to be transferred to the statement specified by the statement label. Every subsequent execution of this GO TO statement results in a transfer to that same statement. Any executable statement immediately following this statement should have a statement number; otherwise, it can never be referred to or executed.

Example:

```
        GO TO L25
      L10:W = Y + Z
              .
              .
              .
      L25:Z = E IAND 2
              .
              .
              .
```

In the above example, each time the GO TO statement is executed, control is transferred to statement L25.

Assigned GO TO Statement

General Form

GO TO xxxxx

Where:xxxxx is the name of a LABEL defined in the DECLARE part of the program.

Example 1

```
     DECLARE CAT LABEL
           CAT = 'A25'
              .
              .
              .
           GO TO CAT
```

Example 2

```
     DECLARE CAT(5) LABEL
           CAT(3) = 'A25'
                .
                .
                .
            GO TO CAT(3)
```

In these examples CAT and CAT(3) are defined as labels in a DECLARE statement. The values of the labels are assigned during program execution.

Computed GO TO Statement

General Form

GO TO ($\underline{x}_1$, $\underline{x}_2$, $\underline{x}_3$, ..., $\underline{x}_n$), $\underline{i}$ OR GO TO labelname(i)

Where: $\underline{x}_1$, $\underline{x}_2$, ..., $\underline{x}_n$, are the labels of executable statements.

$\underline{i}$ is a nonsubscripted BINARY(1) variable whose current value is in the range: $1 \leq i \leq n$

labelname is the name of a LABEL array defined in a LABEL declaration.

This statement causes control to be transferred to the statement labeled $\underline{x}_1$, $\underline{x}_2$, $\underline{x}_3$, ..., or $\underline{x}_n$, depending on whether the current value of $\underline{i}$ is 1, 2, 3, ..., or n, respectively.

<u>Example 1</u>

```
                    GO TO (L25, L10, L7), ITEM
                    .
                    .
                    .
            L7: C = E IAND 2 + AA
                    .
                    .
                    .
            L25:L = C
                    .
                    .
                    .
            L10:D = 21
```

In this example, if the value of the integer variable ITEM is 1, statement L25 will be executed next. If ITEM is equal to 2, statement L10 is executed next, and so on.

<u>Example 2</u>

```
DECLARE LARRAY(3) LABEL INITIAL (L10, L20, L30)
    .
    .
    .
GO TO LARRAY(Y)
```

In this case LARRAY is the name of a LABEL array. If the value of the integer variable Y is 1, statement L10 will be executed next. If Y is equal to 2, statement L20 will be executed and if Y is equal to 3, statement L30 is executed.

ADDITIONAL CONTROL STATEMENTS

IF Statement

General Form

IF a THEN S1 [ELSE S2]

Where: a is a logical expression.

and S1, S2 represent executable statements. If it is desired to execute more than one statement if the condition is or is not met, the form

```
IF a THEN DO
    A1
    A2
    .
    .
    .
    An
    END
ELSE DO
    B1
    B2
    .
    .
    .
    Bn
    END
```

may be used.

Other examples:

```
IF INADD(2) LT O THEN INADD(1) = "FF"

IF I LT 32 THEN GO TO BYTE1
```

<u>Do Statement</u>

General Form

1. DO

   .

   .

   .

   END

   The sequence of statements is performed only once.

2. DO i = m_1 TO m_2 [BY m_3]

   <u>i</u> is a nonsubscripted integer variable of data representation BINARY.

   m_1, m_2 and m_3 are either unsigned integer constants or unsigned nonsubscripted integer variables with a BINARY data representation. If the clause BY m_3 is omitted, it is assumed to be present with m_3 = 1.

   The DO statement is a command to execute at least <u>once</u> the statements that physically follow the DO statement, up to a END statement. These statements are called the range of the DO. The first time the statements in the range of the DO are executed, <u>i</u> is initialized to the value $\underline{m_1}$; each succeeding time <u>i</u> is increased by the value $\underline{m_3}$.

When, at the end of the iteration, $\underline{i}$ is equal to or greater than $\underline{m}_2$, control passes to the statement following the END statement.

If $\underline{m}_2$ is equal to $\underline{m}_1$, the statements in the range of the DO are executed once. Upon completion of the DO, the DO variable contains the value $\underline{m}_2$. All four variables, $\underline{i}$, $\underline{m}_1$, $\underline{m}_2$, and $\underline{m}_3$ must be either BINARY(1) or BINARY(2) in a DO definition.

3. DO WHILE Boolean expression

   If the Boolean expression after the word WHILE is true the sequence of statements down to the END statement is executed as often as the Boolean expression is true. When the expression is false an exit is made from the loop. The Boolean expression must contain variables which are altered during the execution of the DO loop and the result of such alteration must eventually change the value of the Boolean expression from TRUE to FALSE. If this does not occur no regular exit from the loop is possible. However an IF statement in the loop might in the usual way cause a transfer of control outside the loop.

4. DO i = m_1 TO m_2 [BY m_3] WHILE Boolean expression

   This form is a combination of forms 2 and 3. For example:

   DO I = 1 TO 5 WHILE Y LT 4

   The loop would be executed up to 5 times, depending upon the truth of the Boolean expression.

NOTE:

The Boolean expression must start with a variable; starting with a constant will cause an error.

There are several ways in which looping (repetitively executing the same statements) may be accomplished when using the MPL language. For example, assume that a manufacturer carries 100 different machine parts in stock. Periodically, he may find it necessary to compute the amount of each different part presently available. This amount may be calculated by subtracting the number of each item used, OUT(I), from the previous stock on hand, STOCK(I).

Example 1:

```
          .
          .
          .
    I=O
L10:STOCK(I) = STOCK(I) - OUT(I)
    I = I + 1
    IF I LT 100 THEN GO TO L10
L30:C = D + E
          .
          .
          .
```

Explanation:

The first, third, and fourth statements required to control the previously shown loop could be replaced by a single DO statement as shown in example 2.

Example 2:

```
          .
          .
          .
      DO I = 1 TO 100
  L25:STOCK(I) = STOCK(I) - OUT(I)
      END
      C = D + E
          .
          .
          .
```

Explanation:

In example 2, the DO variable, I, is set to the initial value of 1. Before the second execution of statement L25, I is increased by the increment, 1, and statement L25 is again executed. After 100 executions of the DO loop, I equals 100. Since I is now equal to the test value, 100, control passes out of the DO loop and the fourth statement is executed next. Note that the DO variable I is now 100.

Example 3:

```
          .
          .
          .
      DO I = 1 TO 9 BY 2
      J = I + K
  L25:ARRAY(J) = BRAY(J)
      END
      C = D + E
          .
          .
          .
```

Explanation:

In example 3, statement L25 is at the end of the range of the DO loop. The DO variable, I, is set to the initial value of 1. Before the second execution of the DO loop, I is increased by the increment, 2, and the second and third statements are executed a second time. After the fifth execution of the DO loop, I equals 9. Since I is now equal to the test value, 9, control passes out of the DO loop and the fifth statement is executed next. Note that the DO variable, I, is now 9.

Programming Considerations in Using a DO Loop

1. The indexing parameters of a DO statement $(\underline{i}, \underline{m}_1, \underline{m}_2, \underline{m}_3)$ should not be changed by a statement <u>within</u> the range of the DO loop.
2. There may be other DO statements within the range of a DO statement. All statements in the range of the inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DO's and may be nested nine deep.

<u>Example 1</u>:

```
DO I = 1 TO 4
C(I) = D(I) IAND 2
                                            Range of
DO J = 2 TO 5
                          Range of          Outer DO
E(J) = C(I)
                          Inner DO
END

END
```

<u>Example 2</u>:

```
DO INDEX = L TO M

N = INDEX + K
                                            Range of
DO J = 1 TO 100 BY 2
                          Range of          Outer DO
TABLE(J) = SUM(J,N)-1
                          Inner DO
END

D(N) = C(N)

END
```

3. A transfer out of the range of any DO loop is permissible at any time.

4. The extended range of a DO is defined as those statements in the program unit containing the DO statement that are executed between the transfer out of the innermost DO of DO's and the transfer back into the range of this innermost DO. The following restrictions apply:

   . Transfer into the range of a DO is permitted only if such a transfer is from the extended range of the DO.

   . No DO statements are permitted in the extended range of the DO.

   . The indexing parameters ($\underline{i}$, $\underline{m}_1$, $\underline{m}_2$, $\underline{m}_3$) cannot be changed in the extended range of the DO.

   Note that a statement that is the end of the range of more than one DO statement is within the innermost DO loop.

5. The indexing parameters ($\underline{i}$, $\underline{m}_1$, $\underline{m}_2$, $\underline{m}_3$) may be changed by statements outside the range of the DO statement. No transfer may be made into the range of the DO statement from outside the DO statement.

6. The use of, and return from, a subprogram from within any DO loop in a nest of DO's is permitted.

POINTERS AND THE 'DO' STATEMENT

Consider a table of student data containing student name, student ID, and eight class entries each containing semester hours and class number. The DECLARE statement to allow for 100 such entries would be:

```
DECLARE
01 STUDNT(100),
   02 NAME CHAR(10),
   02 ID DEC(3),
   02 CLASES(8),
      03 HOURS DEC(2,1),
      03 CNUM DEC(3)
```

The hours can be summed stepping through all entries in the table using the following statements:

```
DO ZZ = 'STUDNT' TO 'STUDNT+99*53' BY 53
DO J = 1 TO 8
SUM = SUM + HOURS(J):ZZ
END
END
```

If it were desired to have the entries available in some order, each entry in the table can be linked to the next.

```
     DECLARE
       01 STUDNT (100),
          02 LINK BINARY(2),
          02 NAME CHAR(10),
          02 ID DEC(3),
          02 CLASES(8),
             03 HOURS DEC(2,1),
             03 CNUM DEC(2,1)
```

The previous example would then be:

```
                   ZZ = initial entry
     LOOP:DO J = 1 TO 8
          SUM = SUM + HOURS(J):ZZ
          END
          ZZ = LINK:ZZ
          IF ZZ NE Ø THEN GO TO LOOP
```

<u>END Statement</u>

The END statement terminates a DO loop or a procedure (see below). An optional label may follow which then must be the same as the label of the beginning of the DO loop, or the label of the procedure.

For example:

```
                             XX: DO
                                 .
                                 .
                                 .
                                 END XX
```

MAIN PROCEDURE

A main procedure (usually referred to as a mainline program in other languages) is identified by the statement

PROCEDURE OPTIONS (MAIN)

or

PROCEDURE OPTIONS (MAIN, stack name)

In the first example, the compiler will assume that the program is wholly in RAM memory and will allocate temporary storage and stack. Additionally, the compiler will generate jumps around in line DECLARE statements.

When a stack name is given, the compiler assumes a mixture of RAM and ROM memories. The programmer must allocate a temporary variable called T and a stack using a DECLARE statement. The compiler will not generate jumps around in line DECLARE's as it is assumed that the programmer will place DECLARE's in RAM and procedures in ROM.

A main program should be terminated with a 'branch to self' instruction. This will stop the program and also keep it from running right on into any sub programs following the main program. An END statement will not generate a terminating instruction. Use $ BRA * or LABEL: GO TO LABEL before the END statement.

Subscripts and mathematical expressions are evaluated by subroutines called by the compiler and appended to the end of the user's program. A label ZFFF is attached to the END statement following these routines. The user can find the first byte address following the end of his program by referring to the label.

SUBROUTINE PROCEDURES

It is sometimes desirable to write a program which, at various points, requires the same computation to be performed with different data for each calculation. It would simplify the writing of that program if the statements required to perform the desired computation could be written only once and then could be referred to freely, with each subsequent reference having the same effect as though these instructions were written at the point in the program where the reference was made.

For example, to take the cube root of a number, a program must be written with this object in mind. If a general program were written to take the cube root of any number, it would be desirable to be able to combine that program (or subprogram) with other programs where cube root calculations are required.

The MPL language provides for the above situation through the use of subroutine procedures. A subroutine procedure is set up as follows:

label: PROCEDURE (a_1, a_2, a_3, ... a_n)

A subroutine procedure label is a statement label as defined above and a_1, a_2, a_3 ... is the parameter list of arguments associated with the subroutine procedure.

The SUBROUTINE is referenced by a CALL statement, which consists of the word CALL followed by the label of the subroutine procedure and its parenthesized arguments.

For example:

```
ABC: PROCEDURE (P,Y,Z)
          .
          .
          .
          .
        RETURN
        END
```

is called by the statement:

```
CALL ABC (R,S,T)
```

where R, S and T are the arguments in the parameter list corresponding to the dummy arguments P, Y, Z in the subroutine procedure. Of course there need not be any arguments at all. In this connection it should be noted that if the main procedure and the subroutine procedure(s) are compiled together, then all variables in both the main and the subroutine procedures are effectively common to all procedures. On the other hand if subroutine procedures are compiled separately, the variable names and statement labels within them do not relate to any other procedures. Constants used as arguments default to BINARY(2).

The subroutine procedure may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement. The subroutine procedure name (label) must not appear in any other statement in the procedure.

Subroutines compiled by themselves must contain a $T RMB 40 or DCL T(40) statement for temporary workspace used by routines called when the subroutine is compiled.

A main program calling a subroutine compiled separately must have an EQU telling the main program where the subroutine starts.

The dummy arguments ($\underline{a}_1$, $\underline{a}_2$, $\underline{a}_3$, ...,$\underline{a}_n$) may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. Additional information about dummy arguments is in the section "Arguments in a Subroutine Procedure".

Example:

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the subroutine procedure is illustrated in the following example. The object of the subprogram is to "copy" one array directly into another.

Calling Program

```
DECLARE
01 W(100) BINARY(1),
01 Y(100) BINARY(1)
CALL COPY (W,Y,100)
   .
   .
   .
```

SUBROUTINE Procedure

```
COPY:PROCEDURE(C,D,N)
     DECLARE
     01 C(100) BINARY(1),
     01 D(100) BINARY(1),
     01 N BINARY(2),
     01 NN BINARY(1),
     01 I BINARY(1)
     NN=N
     DO I = 1 TO NN
     D(I) = C(I)
     END
     RETURN
     END
```

The same names for dummy arguments may not be used in subsequent subroutines in the same compilation. Once a name is defined, it remains defined for the remainder of the program. Thus all variables are common and need not be redefined in each subroutine.

ARGUMENTS IN A SUBROUTINE PROCEDURE

The dummy arguments of a subroutine procedure appear after the word PROCEDURE and are enclosed in parentheses. They are, in effect, replaced at the time of execution by the actual arguments supplied in the CALL statement of the calling program. The dummy arguments must correspond in number, order, and type to the actual arguments. For example, if the actual argument is BINARY, then the dummy argument must be BINARY. If a dummy argument is an array, the corresponding actual argument must be an array. The size of the dummy array must not exceed the size of the actual array.

The actual arguments can be:

- Any type of constant
- Any type of nonsubscripted variable
- An array name

For each dummy argument in the procedure, an appropriate DECLARE specification statement must appear in the procedure.

If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a nonsubscripted variable name or an array name. A constant should not be specified as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram. Dummy arguments may not be used for:

. subscripts

. arguments in another subroutine call

. index of a computed GO TO statement

. parameters or index of DO loops

An alternative method of using a parameter list which is more efficient in terms of object code generation may be used if there are three or less arguments in the parameter list and the arguments themselves are of a certain length. The general form in this case is:

$$\text{label: PROCEDURE} < a_1, a_2, a_3 >$$

and the corresponding CALL statement would be:

$$\text{CALL label} < b_1, b_2, b_3 >$$

Note especially the use of the '<' and '>' delimiters. a_1, a_2, b_1, and b_2 <u>must</u> be one byte in length and a_3 and b_3 <u>must</u> be two bytes in length. Of course the same method may be used if there are only one or two arguments in the parameter list, but in those cases all arguments must be one byte in length.

Examples:

CALL XYZ <3,XX,J>

CALL DEF < Z,VAL:YY >

CALL GHI <,,PTR(3)>

Example of a main program calling a subroutine when each has been compiled as a separate program. It should be noted using this technique causes redundant coding when the same type of operations are performed in each program. For instance, all routines called from the library for computing subscripts will be called for each program using subscripts. When compiled together, these routines are called once.

```
!COMPILE MAIN PROGRAM SEPARATELY AND CALL
!SUBROUTINE COMPILED SEPARATELY
   ORIGIN "200"
   DECLARE
   01 W(100) BIN(1),
   01 Y(100) BIN(1)
   DCL I
!SUBROUTINE 'COPY'  HAS BEEN COMPILED AND STARTS AT
!HEX ADDRESS 2A
$COPY  EQU $2A
   PROCEDURE  OPTIONS(MAIN)
   DO I = 1 TO 10
   W(I)=I
   END
   I=10
   CALL COPY(W,Y,I)
STOP:GO TO STOP
$  OPT NOL
    END
```

Subprogram

```
!COMPILE SUBROUTINE CALLED FROM SEPARATELY
!COMPILED MAIN PROGRAM.
COPY:   PROCEDURE(C,D,N)
! NOTE*** THE DECLARE T(40) STATEMENT IS
!         REQUIRED FOR ROUTINES CALLED WHEN
!         THIS SUBROUTINE IS COMPILED.
!         VARIABLE 'T' IS INCLUDED WITH
!         MAIN PROGRAMS BY  PROCEDURE OPTIONS(MAIN)
!         STATEMENT, THEREFORE THE USER DOES NOT SUPPLY IT.
   DECLARE T(40)
   DECLARE
     01 C(100)  BINARY(1),
     01 D(100)  BINARY(1),
     01 N  BINARY(1),
     01 I  BINARY(1)
   DCL NN
  NN=N
  DO I = 1 TO NN
     D(I) = C(I)
  END
  RETURN
!THE FOLLOWING OPTION WILL PROHIBIT THE
!CROSS-ASSEMBLER FROM PRINTING FROM HERE ON.
$    OPT   NOLIST
  END
```

OPERATION

This section describes how to access the compiler on the XEROX Sigma 9 timesharing system and the GE MARK III timesharing system.

INVOKING THE COMPILER

The compiler is invoked by typing

MTSS T/S	GE MARK III T/S
!M68MPL.MPU	OPT BIG
	RUN M68MPL

The compiler will request filenames by typing:

```
ENTER INPUT FILENAME

? input file name

ENTER OUTPUT FILENAME

? output file name (this will be the input file to the
                    cross assembler)
```

The compiler is also available in GE MARK III background.

Invoke as follows:

RUN M68MPLI - This invokes the program that prepares and submits JCL file for background processing.

SOURCE FILE NAME:? enter source file name here.

PRIORITY:? enter H for high, press carriage return for normal, enter L for low which is overnight processing.

ASSEMBLE MPL OUTPUT - YES/NO? enter YES if you wish to compile and assemble both during this run. Enter NO if you wish to compile only. The MPL output file is saved for later retrieval. No MPL output file is saved if you answer YES. An assembly listing and object tape file is saved instead.

```
CONTROL FILE NAME:  M68SNNNN  \  These names are returned
                               | to you.  Keep them, they
JOB ID=JJJJ                    > are required to retrieve
                               / output.
```

Allow anywhere from 10-90 minutes for this to be executed in background. Time required depends on how fast the background processor is running.

Check background progress of job by entering: BST JJJJ

When this entry returns DONE to you, proceed with retrieval program as follows:

```
RUN M68MPLO    starts retrieval program.

JOB ID: ?JJJJ enter the 4-character job#.

COMPILER OUTPUT FILE:?   Press carriage return if you answered
                         YES for both compile and assembly.
                         Enter the name of a file you want the
                         MPL output to be saved in if you
                         compiled only, answered NO.

LIST FILE:?   enter the name of a file you want assembly listing
              retrieved to, press carriage return if you don't
              want to retrieve it.

OBJECT FILE:?  enter name of object tape file, press carriage
               return if you don't want to save it.

CONTROL FILE TO DELETE:?  enter M68SNNNN control file name
                          returned when job was initiated, press
                          carriage return if you don't want to
                          delete it.

PURGE JOB - YES/NO?  YES purges JOB JJJJ, NO saves it.  It will
                     be purged automatically after 36 hours.
                     BE sure to retrieve output before that
                     time.
```

SOURCE PROGRAM ERRORS

Source program errors are identified by printing the source line in error followed by an error message.

```
***ERROR XXX
```

The error number XXX is defined on the following pages.

500 - illegal character

501 - syntax error (or compiler expects a different symbol type)

502 - "SYM" overflow in 'STUFF' (too many symbols in the program)

503 - parse stack overflow (statement is too complex)

504 - this symbol has already been declared as a variable

505 - the compiler expects this symbol to be a declared variable

506 - illegal character scan

507 - token overflow - scan (symbol over 47 characters long)

508 - missing level (level numbers must be in sequence)

509 - name duplicated

510 - decimal location too big

511 - value too big for data representation

512 - DEFINED or BIT items may not have initial values

513 - undefined error message number

514 - undefined error message number

515 - level number cannot be greater than 5

516 - statement label is a variable

517 - index name not found

518 - index representation is not BINARY

519 - index size is not 1

520 - variable cannot be pointed, dummy or subscripted

521 - "TEMP" (variable name T) is not large enough for this
expression

522 - variable name not found or "T" is not declared and there
is no main procedure

523 - too many operands in this expression

524 - variable name has too many occurrences

525 - pointer name not found

526 - pointer not level 01 or is dummy

527 - pointer is array

528 - pointer size is not 2 bytes

529 - pointer data representation is not BINARY

530 - subscript name not found

531 - too many subscripts

532 - subscript not level Ø1

533 - subscript is an array

534 - subscript is not 1 byte

535 - subscript data representation is not BINARY

536 - too many operands

537 - constant ± 127

538 - constant not allowed as first operand

539 - the number of subscripts used do not agree with the number of subscripts in the declare statement

540 - DO operand data representation is not BINARY and/or size is 2

541 - too many nested IF's and/or DO's

542 - compiler error

543 - IF/DO overlap

544 - DO loops are nested more than nine deep

545 - increment of DO is not 1 byte in length

546 - initial or final value size is not same as index

547 - simple variable required

548 - shift operand is not constant

549 - number of bits shifted is not consistent with field size or zero

550 - constant not allowed greater than 1 byte

551 - illegal operation for this picture type

552 - mixed mode not allowed

553 - A=B+C not allowed when mode changes

554 - compiler error. A op B = C has mixed mode

555 - operand for bit operation is not constant Ø or 1

556 - illegal bit operation

557 - STUFF error

558 - illegal action call

559 - action number out of range

560 - picture or initialize conflict between levels; only elementary items may have a picture or be initialized

561 - bit string spans more than one byte

562 - warning code can be deleted; the last statements store into the same location; example: J=1; J=2;

563 - warning code can be deleted; the last statement loaded from and stored into the same location; examples: J=J; Z(5)=Z(5)

564 - the keyword following the variable name is not a scale (picture) attribute; example: binary, decimal or character

565 - the array has more than three dimensions

566 - arrays with more than one dimension can only be dimension as elementary items

567 - more items initialized than the space declared

568 - BASED variables can only be declared at level one (01) and can not be initialized

SOURCE PROGRAM LISTING

```
$ OPT MF=MFTEST,SYMBOL
 DCL I(5) INIT(5,10,7,-3,22)
 DCL J,S,L
 PROC OPTIONS(MAIN)
 L=0
 S="7F"
  DO J = 1 TO 5
!FIND THE LARGEST NUMBER IN THE ARRAY
      IF I(J) LE L THEN GO TO S1
      L = I(J)
!FIND THE SMALLEST NUMBER IN THE ARRAY
S1:   IF I(J) GE S THEN GO TO S5
      S=I(J)
S5: END
STOP:  GO TO STOP
 END
```

ASSEMBLY LISTING

```
                         NAM     LARO
                 ***     COMPILED WITH PL/I VERSION 0.5A
                         OPT     MF=MFTEST,SYMBOL
                 *00002   DCL I(5) INIT(5,10,7,-3,22)
0000 05          I       FCB     5
0001 0A                  FCB     10
0002 07                  FCB     7
0003 FD                  FCB     -3
0004 16                  FCB     22
                 *00003   DCL J,S,L
0005 0001        J       RMB     1
0006 0001        S       RMB     1
0007 0001        L       RMB     1
                 *00004   PROC OPTIONS(MAIN)
0008 0028        T       RMB     40
0030 8E 002F     Z000    LDS     #T+39
                 *00005   L=0
0033 7F 0007             CLR     L
                 *00006   S="7F"
0036 86 7F               LDA A   #127
0038 97 06               STA A   S
                 *00007    DO$J = 1 TO 5
003A 86 01               LDA A   #1
                 *00008 !FIND THE LARGEST NUMBER IN THE ARRAY
003C 97 05       Z001    STA A   J
                 *00009          IF I(J) LE L THEN GO TO S1
003E CE 0000             LDX     #I
0041 D6 05               LDA B   J
0043 BD 0085             JSR     ZF1F
0046 A6 00               LDA A   0,X
0048 91 07               CMP A   L
004A 2E 03               BGT     Z003
004C 7E 005B     Z002    JMP     S1
                 *00010          L = I(J)
                 *00011 !FIND THE SMALLEST NUMBER IN THE ARRAY
004F CE 0000     Z003    LDX     #I
0052 D6 05               LDA B   J
0054 BD 0085             JSR     ZF1F
0057 A6 00               LDA A   0,X
0059 97 07               STA A   L
                 *00012 S1:      IF I(J) GE S THEN GO TO S5
005B CE 0000     S1      LDX     #I
005E D6 05               LDA B   J
0060 BD 0085             JSR     ZF1F
0063 A6 00               LDA A   0,X
0065 91 06               CMP A   S
0067 2D 03               BLT     Z005
0069 7E 0078     Z004    JMP     S5
                 *00013          S=I(J)
006C CE 0000     Z005    LDX     #I
006F D6 05               LDA B   J
0071 BD 0085             JSR     ZF1F
```

```
0074 A6 00              LDA A   0,X
0076 97 06              STA A   S
                 *00014 S5:  END
0078 96 05       S5     LDA A   J
007A 81 05              CMP A   #5
007C 2C 04              BGE     Z006
007E 4C                 INC A
007F 7E 003C            JMP     Z001
                 *00015 STOP:   GO TO STOP
     0082        Z006   EQU     *
0082 7E 0082     STOP   JMP     STOP
                 *00016  END
0085 26 02       ZF1F   BNE     ZF1F2     *** ADD TO INDEX ***
0087 09                 DEX
0088 39                 RTS
0089 5A          ZF1F2  DEC B
008A DF 0E       ZF5F   STX     T+6
008C DB 0F              ADD B   T+7
008E D7 0F              STA B   T+7
0090 24 03              BCC     ZF1F1
0092 7C 000E            INC     T+6
0095 DE 0E       ZF1F1  LDX     T+6
0097 39                 RTS
                        END


SYMBOL TABLE


I       0000 J       0005 L       0007 S       0006 S1      005B
S5      0078 STOP    0082 T       0008 Z000    0030 Z001    003C
Z002    004C Z003    004F Z004    0069 Z005    006C Z006    0082
ZF1F    0085 ZF1F1   0095 ZF1F2   0089 ZF5F    008A
```