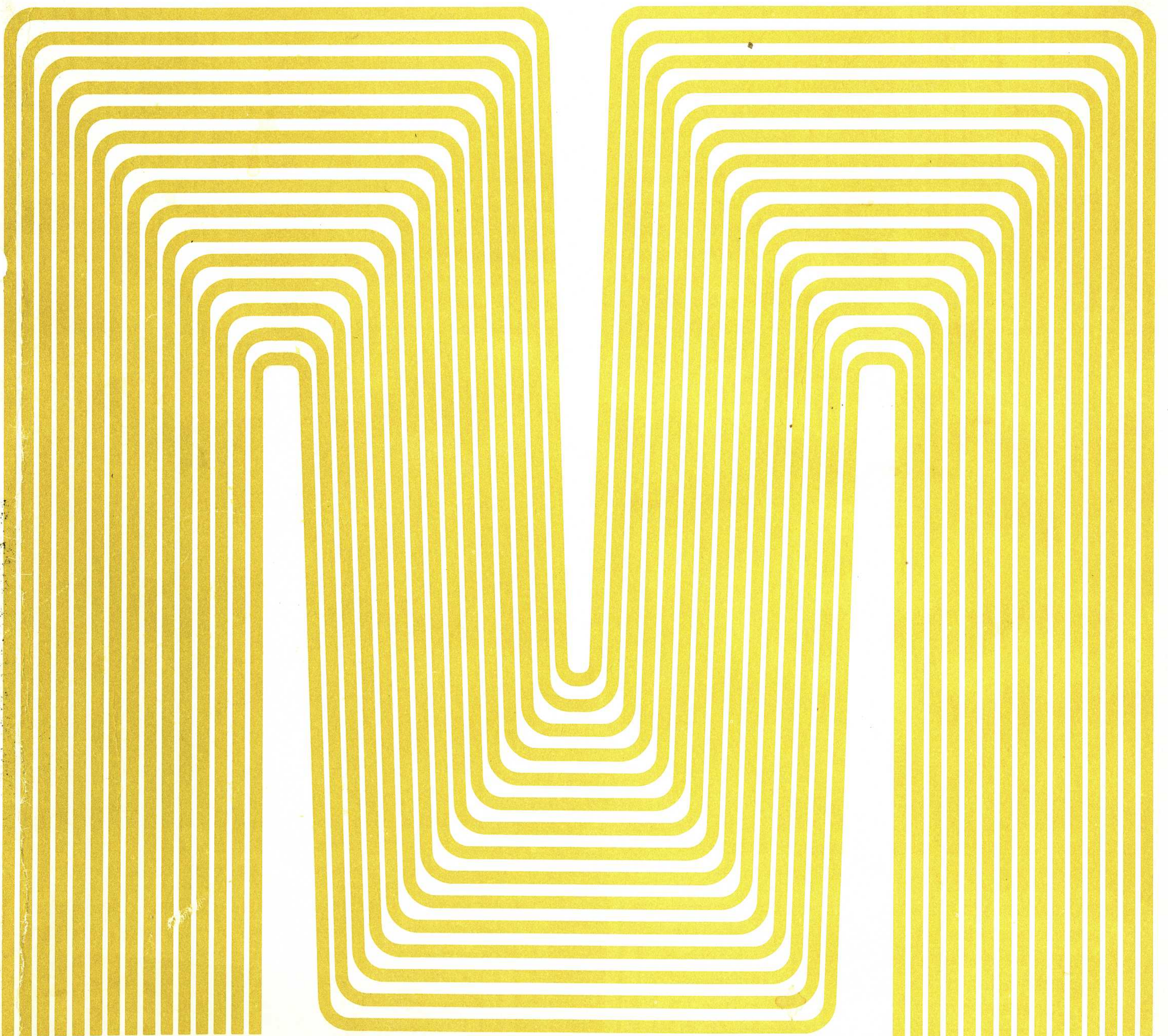
Microdata

Microdata 32/S Computer (MPL)



$5.00

Microdata 32/S Programming Language Reference Manual (MPL)

PUMPL-2

November 1973

© 1973 Microdata Corporation
TM Trademark of Microdata Corporation
Printed in U. S. A.

™ Microdata

Microdata Corporation
17481 Red Hill Avenue
Irvine, California 92705

TABLE OF CONTENTS

Page

SECTION 1 INTRODUCTION

1.1 Summary of the Language 1-1

1.1.1 Block Structure 1-1

1.1.2 Data Description.............................. 1-2

1.1.3 Storage Allocation 1-2

1.1.4 Data Organization.............................. 1-2

1.1.5 Input/Output 1-2

1.1.6 Expressions 1-3

1.2 Syntax Notations 1-3

1.2.1 Notation Variables 1-4

1.2.2 Notation Constants 1-4

1.2.3 Vertical Stroke.............................. 1-4

1.2.4 Braces 1-4

1.2.5 Brackets.............................. 1-5

1.2.6 Ellipsis.............................. 1-5

1.2.7 The Definition Symbol.............................. 1-5

SECTION 2 ELEMENTS OF THE LANGUAGE

2.1 Basic Lanaguage Structure 2-1

2.1.1 Character Set.............................. 2-1

2.1.2 Delimiters 2-2

2.1.3 Operators.............................. 2-3

2.1.4 Separators and Other Delimiters.............................. 2-4

2.1.5 Data Character Set 2-4

2.1.6 Identifiers 2-5

2.1.7 Keywords.............................. 2-5

2.1.7.1 Built-In Function Names.............................. 2-6

TABLE OF CONTENTS (Cont)

		Page
SECTION 2	(Continued)	
2.1.8	The Use of Blanks	2-7
2.1.9	Comments	2-7
2.2	Basic Program Structure	2-7
2.2.1	Simple Statements	2-8
2.2.2	The IF Statement	2-8
2.2.3	Label List	2-8
2.2.4	Groups	2-9
2.2.5	Blocks	2-10
2.2.6	Programs	2-13
SECTION 3	DATA ELEMENTS	
3.1	Data Organization	3-1
3.1.1	Scalar Items	3-1
3.1.1.1	Constants	3-1
3.1.1.2	Scalar Variables	3-1
3.1.2	Arrays	3-1
3.2	Referencing of Data	3-2
3.2.1	Simple Reference	3-2
3.2.2	Subscripted Reference	3-2
3.2.3	Indirect Reference	3-3
3.2.4	Subfield Reference	3-4
3.3	Data Types	3-5
3.3.1	Decimal Numbers	3-5
3.3.2	Bit Strings	3-6
3.3.3	Character Strings	3-7

TABLE OF CONTENTS (Cont)

		Page
SECTION 4	DATA MANIPULATION	
4.1	Expressions	4-1
4.1.1	Simple Expressions	4-1
4.1.1.1	Arithmetic Operations	4-1
4.1.1.2	Comparison Operations	4-3
4.1.1.3	Logical Operations	4-3
4.1.1.4	Shift Operations	4-4
4.1.1.5	The Assignment Operation	4-5
4.1.2	Conditional Expressions	4-5
4.2	Evaluation of Expressions	4-6
SECTION 5	DATA DESCRIPTION	
5.1	Attributes	5-1
5.2	Declarations	5-1
5.2.1	The Declare Statement	5-2
5.2.2	Factoring of Attributes	5-3
5.2.3	Multiple Declarations	5-4
5.2.4	Procedure Labels	5-4
5.2.5	Scope of Declarations	5-4
5.2.5.1	Scope of External Names	5-4
5.2.5.2	Basic Rules for the Use of Identifiers	5-8
5.2.6	The Attributes	5-8
5.2.6.1	The Type Attribute	5-8
5.2.6.2	The Procedure Attribute	5-9
5.2.6.3	The Micro Attribute	5-10
5.2.6.4	The Dimension Attribute	5-11
5.2.6.5	The Size Attribute	5-12
5.2.6.6	The Area Attribute	5-13
5.2.6.7	The Literally Attribute	5-15

TABLE OF CONTENTS (Cont)

		Page
SECTION 6	PROCEDURES, FUNCTIONS, AND SUBROUTINES	
6.1	Parameters	6-1
6.2	Procedure References	6-1
6.2.1	Function Reference	6-2
6.2.2	Built-In Functions	6-3
6.2.3	Subroutine References	6-5
6.3	Procedure Reference Examples	6-5
6.4	Arguments	6-7
6.4.1	Passing Arguments to a Procedure	6-8
SECTION 7	DYNAMIC PROGRAM STRUCTURE	
7.1	Program Control	7-1
7.2	Activation and Termination of Blocks	7-1
7.2.1	Dynamic Descendents	7-1
7.3	Allocation of Data	7-2
7.3.1	Definitions and Rules	7-3
7.3.2	Storage Classes	7-3
7.3.2.1	The Static Storage Class	7-3
7.3.2.2	The Automatic Storage Class	7-4
SECTION 8	STATEMENTS	
8.1	Relationship of Statements	8-1
8.1.1	Assignment Statements	8-1
8.1.2	Control Statements	8-1
8.1.3	Data Declaration Statement	8-1
8.1.4	Block Statements	8-1
8.2	Sequence of Control	8-2

TABLE OF CONTENTS (Cont)

		Page
SECTION 8	(Continued)	
8.3	Alphabetic List of Statements	8-3
8.3.1	The Assignment Statement	8-3
8.3.2	The Begin Statement	8-5
8.3.3	The Call Statement	8-6
8.3.4	The Declare Statement	8-7
8.3.5	The Do Statement	8-7
8.3.6	The End Statement	8-12
8.3.7	The EOF Statement	8-13
8.3.8	The GO TO Statement	8-13
8.3.9	The If Statement	8-15
8.3.10	The Null Statement	8-17
8.3.11	The Procedure Statement	8-17
8.3.12	The Repeat Statement	8-19
8.3.13	The Return Statement	8-20
APPENDIX A	SYNTAX OF THE LANGUAGE	
	Level 1 Syntax	A-2
	Level 2 Syntax	A-4
	Syntax Cross Reference	A-8
APPENDIX B	COMPILIER TOGGLES	
APPENDIX C	SAMPLE MPL PROGRAM	

Section 1.
INTRODUCTION

1.1 SUMMARY OF THE LANGUAGE

This document describes the Microdata 32/S Programming Language (MPL). The MPL language is used to write programs for the 32/S system.

A program, called the MPL compiler, translates MPL statements into 32/S machine instructions, assigns storage locations, and performs other functions required to produce an executable machine language program.

MPL is designed to be the primary implementation language for the 32/S computer. Although MPL is a high-level programming language, it is not machine independent. In fact the 32/S machine and MPL were designed symbiotically. Full access to the resources of the 32/S computer is provided through appropriate language constructs. Each construct of MPL is directly mirrored in the 32/S architecture. For this reason the compiler of MPL can generate execution code that is as efficient as that generated by assembly language programs.

In the immediately following subsections, the language is summarized briefly. Full details of the language are given in Sections 2 through 8.

1.1.1 BLOCK STRUCTURE

MPL statements are organized into sections called <u>blocks</u>. A program may consist of one or more blocks. Blocks may be separate from one another with no statements in common, or they may be <u>nested</u>. A block is said to be nested if the statements comprising the block are contained within another block.

Blocks serve two functions. (1) They provide for the automatic allocation of storage. Storage for data declared in a block is automatically allocated when

the block is entered and freed for use by other blocks when a block is terminated. (2) They provide a means of using the same name for different purposes in different blocks without ambiguity.

Certain blocks called procedures may be invoked from different places in the program, and will return control to the point from which they were invoked.

1.1.2 DATA DESCRIPTION

All data used in an MPL program are described as having certain attributes. For example numeric data have a size attribute such as BYTE or WORD. The programmer must declare all the attributes for a variable before the variable is referenced.

1.1.3 STORAGE ALLOCATION

The storage for data in a program may be assigned statically or dynamically. Static storage is assigned when a program begins execution and remains allocated during the entire program execution. Dynamic storage is allocated on block entry and freed upon block exit.

1.1.4 DATA ORGANIZATION

The MPL language supports two types of data organization; scalars and arrays of one dimension.

1.1.5 INPUT/OUTPUT

Input and output communication with system peripheral devices is accomplished by assigning variable names to the various registers of a device controller. The contents of these registers may then be accessed by name in a manner identical to the accessing of any other variable. Facilities are provided to enable the programmer to write interrupt routines to service the interrupts generated by input/output devices.

1.1.6 EXPRESSIONS

Expressions are used in MPL to specify computations to be performed. Two types of expressions may be written in MPL.

The first type of expression, called a simple expression, is similar to that of algebra. For example: A+2*(B+C)

Specifies multiplying the sum of B and C by 2 and adding the result to A. Data with different size-attributes may be used in the same expression. In the example above, A may be a BYTE variable, B a WORD variable, and C a DOUBLE variable.

The second type of expression is the conditional expression. For example:

```
IF A > B THEN A+1 ELSE B-5
```

The value of this expression is A+1 if the value of A is greater then the value of B. Otherwise the value of the expression is B-5.

MPL allows the use of a conditional expression anywhere that a simple expression is allowed.

1.2 SYNTAX NOTATION

In this manual a uniform system of notation is used to describe the manner of writing MPL statements or parts of statements. This system of notation is not part of MPL. It is a standard notation that may be used to describe the syntax of any phrase-structured programming language. A system of notation such as this is commonly called a metalanguage.

The following paragraphs describe the notation.

1.2.1 NOTATION VARIABLES

A notation variable is used to name a general class of elements in the programming language. A notation variable is written in lower case letters. Multiple words in the notation variable are connected by hyphens; no spaces may exist in a notation variable. Some examples of notation variables are:

1. item-list
2. goto-statement

1.2.2 NOTATION CONSTANTS

Notation constants specify the literal occurrence of the characters in the language element being described. For example:

    DECLARE identifier BYTE;

indicates the literal occurrence of the word DECLARE followed by the notation variable "identifier" followed by the literal occurrence of the word BYTE followed by the literal occurrence of the semicolon (;).

1.2.3 VERTICAL STROKE

The vertical stroke | indicates that a choice of alternatives is to be made. For example:

    BYTE | WORD | DOUBLE

indicates that a choice is to be made from one of the three notation constants BYTE, WORD, or DOUBLE.

1.2.4 BRACES

Braces { } are used to group notations constants and variables into a syntatical unit. When braces are used for this grouping, the presence of the syntactical unit is required. For example:

    identifier {BYTE | WORD}

indicates that the notation variable "identifier" must be followed by the literal occurrence of either the word BYTE or the word WORD.

The notation:

{identifier BYTE | identifier WORD}

has exactly the same meaning as the previous example.

1.2.5 BRACKETS

Brackets [ ] are also used to group notation constants and notation variables into syntactial units. When brackets are used, the syntactial unit is optional. For example:

identifier DOUBLE [BASED]

indicates the literal occurrence of the word DECLARE followed by the notation variable. "identifier" followed by the literal occurrence of the word BYTE followed by the literal occurrence of the semicolon (;).

Brackets may be used to enclose a syntactical unit containing alternatives. When used in this fashion, the brackets indicate that any one alternative may be used or that none of them may be used. For example:

[alphabetic-character | digit]

indicates a choice of either the notation variable "alphabetic-character" or the notation variable "digit" or neither of them.

1.2.6 ELLIPSIS

Ellipsis ... following a syntactical unit indicates that the unit may be repeated. If the syntactial unit contains alternatives, a different choice of alternative may be used for each repetition. For example:

[alphabetic-character | digit]...

indicates that any number of "alphabetic-character"s or "digit"s may appear in any order.

1.2.7 THE DEFINITION SYMBOL

The definition symbol ::= is used to define a notation variable. The symbol may be interpreted to mean "may be composed of".

For example:

identifier ::= alphabetic-character [alphabetic-character |digit] ...

defines the notation variable "identifier". The example indicates that an "identifier" may be composed of an "alphabetic-character" followed optionally by any sequence of "alphabetic-character" and/or "digit", or by none at all.

Appendix A uses the syntax notation to present a formal syntatical description of MPL. In other parts of the manual the notation is used to describe the language elements in an informal way.

Section 2.
ELEMENTS OF THE LANGUAGE

2.1 BASIC LANGUAGE STRUCTURE

MPL programs consist of a collection of statements. A statement is composed of characters and is always terminated with the special character semicolon. Statements may be written in free-field format and are independent of any physical record boundaries.

The programmer may define record margins so that programs can be entered through fixed-length records such as cards. Characters in the margin are not considered to be part of the program and can be used for arbitrary purposes such as card deck identification.

2.1.1 CHARACTER SET

The character set is composed of digits, special characters, and English language alphabetic characters. The corresponding English language upper and lower case letters are considered to be identical. In this manual only the upper case form of the letters are used.

There are 28 alphabetic-character symbols defined as follows:

alphabetic-character ::=
A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W
X | Y | Z | _ | #

There are 10 digit symbols defined as follows:

digit ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

An alphameric-character is defined as

alphameric-character::=
alphabetic-character | digit

There are 20 special character symbols. Their names and graphic representations are given in table 1 below.

Table 1. Special Characters

NAME	GRAPHIC
Blank	
Equal or Assignment symbol	=
Plus	+
Minus	-
Asterisk or Multiply Symbol	*
Slash or Divide Symbol	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Single Quote	'
Double Quote	"
Currency Symbol	$
Commercial-at Sign	@
Semicolon	;
Colon	:
Not Symbol	¬
And Symbol	&
Or Symbol	|
Greater-Than Symbol	>
Less-Than Symbol	<
Question Mark	?
For keyboards without the symbols | and ¬ the following substitutions are made	
Not symbol	^
Or symbol	!

2.1.2 DELIMITERS

The special characters are used to from delimiters. A delimiter is an operator or a separator.

2.1.3 OPERATORS

The operators are used in expressions to indicate computations to be performed. The expression operators are shown in table 2 below.

Table 2. Expression Operators

GRAPHIC	USAGE
+	denoting addition or (prefix) plus
-	denoting subtraction or (prefix) negation
*	denoting multiplication
/	denoting division
>	denoting greater-than
>=	denoting greater-than-or-equal-to
=	denoting equal-to
¬=	denoting not equal to
<=	denoting less-than-or-equal-to
<	denoting less-than
¬	denoting prefix not (one's complement)
|	denoting logical inclusive Or
&	denoting logical And
:=	denoting assigned-to

In addition the following identifiers are reserved for use as operators in expressions.

Table 3. Identifiers Used As Expression Operators

IDENTIFIER	USE
XOR	denoting logical Exclusive Or
MULD	denoting double-word multiply
DIVD	denoting double-word divide
MOD	denoting modulo
SRA	denoting shift-right arithmetic

Table 3. Identifiers Used As Expression Operators (continued)

IDENTIFIER	USE
SRL	denoting shift-right logical
SLC	denoting shift-left circular
SLL	denoting shift-left logical

2.1.4 SEPARATORS AND OTHER DELIMITERS

The following special characters are used to separate and delimit elements of the language.

Table 4. Separator Characters

GRAPHIC	USE
( )	used in expressions, for enclosing subscripts, and for specifying information associated with various keywords.
,	Separates elements of a list
;	terminates a statement
=	used in the assignment statement and DO statement
:=	used in the assignment statement
+=	used in the assignment statement
:	used to terminate a label and in field selection.
$	used to specify field selection
'	used to enclose character strings
"	used to enclose bit strings
@	used to modify a storage reference
?	Used to terminate scanning of a physical record

2.1.5 DATA CHARACTER SET

The MPL source language statements are written in the character set defined in the preceding section. However, the characters that can be processed as data are not limited. Data characters may include all 256 8-bit combinations. The collating sequence for data characters is ASCII-67.

2.1.6 IDENTIFIERS

Identifiers are strings of alphameric characters, the first of which is alphabetic. Identifiers may consist of from 1 to 255 characters. The definition of a an identifier is:

identifier ::=

alphabetic-character [alphameric-character] ...

Identifiers in MPL are used for the following:

Scalar variable names

Array names

Statement labels

Procedure names

Keywords

Examples:

X

VARA

RATE_OF_PAY

#2152

X2

TTY_STATUS

___#7

2.1.7 KEYWORDS

A keyword is an identifier which is a part of the language. Some keywords are reserved words and may not be used except in their intended structural use. The following is a list of the reserved keywords.

BASED	LITERALLY
BEGIN	MAIN
BIT	MICRO
BY	MOD
BYTE	MULD
CALL	POINTER
CASE	PROCEDURE
CONSTANT	PRTNUM
DECLARE	REPEAT
DIVD	RETURN
DO	SLC
DOUBLE	SLL
ELSE	SRA
END	SRL
EXTERNAL	THEN
FOREVER	TIMES
GO	TO
GOTO	WHILE
IF	WORD
INTERRUPT	XOR

2.1.7.1 BUILT-IN FUNCTION NAMES. Built-in function names are keywords that name functions accessible to the programmer. Built-in function names are <u>not</u> reserved and may be used as variable identifiers or labels. The use of a built-in function name as an identifier overrides the built-in function itself in the scope where identifier is known (see Section 5 "Scope of Declarations"). Table 10 (in Section 6) lists the built-in function names.

2.1.8 THE USE OF BLANKS

Identifiers, constants, and composite operators (e.g., >=) may not contain blanks.

Identifiers and/or constants may not be immediately adjacent. They must be separated by a delimiter, one or more blanks, or a comment.

Examples:

```
CALL SUB          is not equivalent to CALLSUB
A TO 10 BY 3      is not equivalent AT010BY3
A=5               is equivalent to A = 5
```

2.1.9 COMMENTS

General format:

```
/*comment-string*/
```

Comments are used for documentation of the source program and have no effect on execution. However, the character $ appearing in a comment is used to control the compilation process. See Appendix 2 for details.

A comment may appear wherever a blank is allowed, except in a character-string. The comment-string in a comment may not contain the character pair */ in that order.

Example:

```
FACTORIAL:  /* PROCEDURE TO COMPUTE X! */
            PROCEDURE (X);
              .
              .
              .
            END;
```

2.2 BASIC PROGRAM STRUCTURE

An MPL program is composed of basic program elements called statements.

Statements are grouped into larger elements the group and the block. There are two types of statements: the simple statement and the if-statement.

2.2.1 SIMPLE STATEMENTS

General format:

simple-statement::=

[label-list] [[statement-identifier] statement-body] ;

The statement-identifier is a keyword indicating the kind of statement. If no statement-identifier appears the statement is an assignment-statement. If only the terminating semicolon appears the statement is a null-statement.

Examples:

```
LABEL:  DO I = 1 TO 5;      /* DO is the "statement-identifier" */
L1:L2:  A = B + C;          /* assignment-statement with two labels */
        ;                   /* null-statement  */
```

2.2.2 THE IF STATEMENT

General format:

IF-statement::=

IF expression THEN unit-1 [ELSE unit-2]

The if-statement is a compound statement that contains other statements within it.

Each unit of an if-statement has a terminal semicolon. The semicolon of the final unit also terminates the if-statement. The if-statement itself is not otherwise terminated by a semicolon.

Example:

```
IF A = B THEN C += 2; ELSE C=5;
```

2.2.3 LABEL LIST

General format:

label-list::=

{identifier:} ...

Statements may be preceded by a label-list. The identifiers in the list are called labels and any one of them may be used to refer to the statement.

The label-list of a procedure-statement is a special case. For this statement the label-list is mandatory and may only contain a single identifier. The label of a procedure statement is called the entry-name of the procedure.

2.2.4 GROUPS

A group is a collection of one or more statements and is used to control the sequence of execution of a program. There are three forms of group. The first, called the do-group, is defined by:

```
do-group::=
 [label-list]  do-statement
               statement...
               END [identifier] ;
```

If an identifier follows END, it must correspond to an identifier in the label-list of the do-statement.

The second form of group, called the repeat-group, is defined by:

```
repeat-group::=
 [label-list]  repeat-statement
               statement...
               END [identifier] ;
```

If an identifier follows END, it must correspond to an identifier in the label-list of the repeat-statement.

The third form of group is a single statement as follows:

```
 [label-list]  statement
```

The statement-identifier of the single statement group may not be DO, END, PROCEDURE, BEGIN, or DECLARE.

Example:

```
ALPHA:  DO;
          IF A = B THEN
            DO;
              X=Y;
              P=Q;
            END;
        END ALPHA;
```

This example contains two do-groups. The first do-group contains the second do-group within it. In the example every statement except the DO and END statements is a single statement group.

2.2.5 BLOCKS

A block is a collection of statements that define the program region--or scope--throughout which identifiers are known, and for which storage is allocated to them. Blocks are also used to control the sequence of execution.

There are two kinds of blocks, begin-blocks and procedures.

A begin-block has the form:

```
begin-block::=
  [label-list]  begin-statement
                statement...
                END [identifier] ;
```

If an identifier follows END, it must correspond to an identifier in the label-list of the begin-statement.

A procedure has the form:

```
procedure::=
  identifier:   procedure-statement
                statement...
                END [identifier] ;
```

If an identifier follows END, it must correspond to the identifier of the procedure-statement.

The begin-statement and procedure-statement in the above forms are called heading-statements.

Although begin-blocks and procedures have the same role in delimiting scope of names and allocation of storage, they differ in the way in which they are activated. A begin-block, like a single statement, is activated by normal sequential flow of control, and can appear wherever a single statement can appear. A procedure may only be activated by a CALL statement or by a function reference. Normal sequential flow of control skips over procedures.

Since a procedure can be activated only by a reference to it, every procedure must have an entry-name. The identifier required on the procedure-statement serves as the entry-name.

Any block, A, may include another block, B, within it. However, partial overlap is not possible. Block B must be completely included within block A. The inclusion of one block within another is called nesting. Such nesting may occur to a maximum of 16 nesting levels.

A procedure-block that is not included in any other block is called an external-procedure.

A procedure nested within a block is called an internal-procedure.

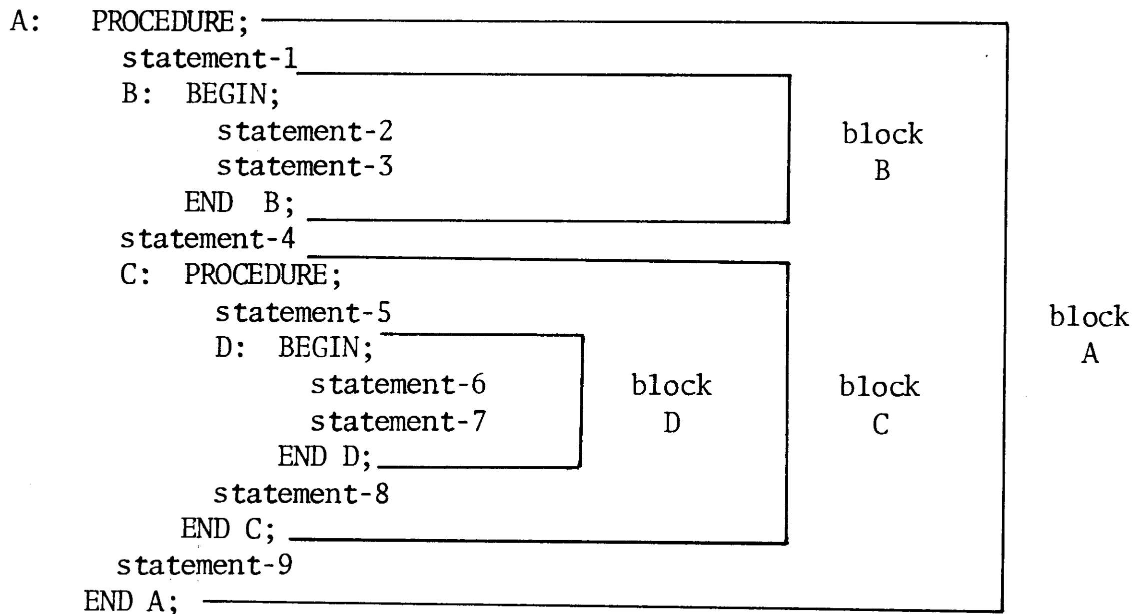
Begin blocks must be nested within another block. Therefore, the only form of external block is the procedure.

All of the text of a block except the label on the block heading-statement is said to be contained in the block.

The part of the text of a block B that is contained in the block B, but is not contained within any other block nested inside of B, is said to be internal to the block B.

The notion internal to is vital to the understanding of the definition of scope and to the understanding of allocation of storage.

Example:



```
A:   PROCEDURE;
       statement-1
       B: BEGIN;
            statement-2
            statement-3
          END  B;
       statement-4
       C: PROCEDURE;
            statement-5
            D: BEGIN;
                 statement-6
                 statement-7
               END D;
            statement-8
          END C;
       statement-9
     END A;
```


In the example, statement-1 through statement-9 represent simple statements.

As shown by the brackets to the right of the example, block A contains blocks B and C, and block C in turn contains block D.

Block A is an external-procedure (it is not contained within any other block). The entry-name is A and is an external name.

Blocks B and D are begin-blocks.

Block C is an internal-procedure (it is contained in block A).

The text internal to block A is:

```
PROCEDURE;
  statement-1
  B:
  statement-4
  C:
  statement-9
  END A;
```

The text internal to block B is:

```
BEGIN:
  statement-2
  statement-3
END B;
```

The text internal to block C is:

```
PROCEDURE;
  statement-5
  D:
  statement-8
  END C;
```

The text internal to block D is:

```
BEGIN;
  statement-6
  statement-7
  END D;
```

2.2.6 PROGRAMS

A program is composed of one or more external-procedures. Thus a program is a set of procedure blocks each of which may have other procedures contained in them.

Section 3
DATA ELEMENTS

Information manipulated by MPL programs during execution is called data. Data may be integers, characters, arbitrary collections of bits, or pointers to other items of data.

3.1 DATA ORGANIZATION

Data may be either scalars (i.e., single items) or arrays (i.e., collections of single items)

3.1.1 SCALAR ITEMS

A scalar item may be a constant or the value of a scalar variable.

3.1.1.1 CONSTANTS. A constant is a data item that denotes itself. That is, its representation in a program is both its name and its value; thus, the value of a constant cannot change during execution of a program.

3.1.1.2 SCALAR VARIABLES. A scalar variable denotes a data item. This data item is called the value of the variable. The identifier used in a program to reference the data item is called the name of the variable. A variable may take on more than one value during the execution of a program. The set of possible values of a variable is called the range of the variable.

3.1.2 ARRAYS

An array is an ordered collection of scalar data items all of which have the same declaration. The number of scalar elements in the array is specified by the use of a dimension attribute in the declaration for the name of the array. The elements of the array are numbered starting with zero. The maximum element number corresponds to the declared dimension.

Example:

```
DECLARE X(3) WORD;
```

This statement declares X to be an array containing four scalar elements. The number, 3, appearing in the example is the dimension. Each element is a word. The elements of the array X can be conceptualized as a collection of data items referenced as follows:

```
X(0)
X(1)
X(2)
X(3)
```

The number in parentheses following the array name identifies the particular element being referenced.

3.2 REFERENCING OF DATA

This portion of the manual describes the rules for refering to data items. Reference is made to data items via a simple reference, a subscripted reference or an indirect reference. Additionally any subfield of a variable declared with WORD precision may be referenced via a field selection modifier.

3.2.1 SIMPLE REFERENCE

A simple reference is an identifier (see section 2). A simple reference may refer to a scalar or to an array. If a simple reference is prefixed by the commercial at sign @, then the reference is to the storage location of the variable and not to the value.

Example:

ABC is a reference to the value of the data item.

@ABC is a reference to the storage location of the data item.

3.2.2 SUBSCRIPTED REFERENCE

A subscripted-reference is used to refer to a particular element of an array.

The general form of a subscripted reference is

```
subscripted-reference::=
identifier (expression)
```

The value of the expression within the parenthesis specifies the particular element of the array.

Example:

Assume that I has a value of 3, and that the array X is declared as

```
DECLARE X(5) WORD;
```

then

```
X(0)   references element zero of the array
X(I+2) references element five of the array
```

3.2.3 INDIRECT REFERENCE

A commercial at sign, @, following a variable indicates that the value of the variable is an address of some other variable. The commercial at sign may follow only variables declared with the POINTER TO attribute (see "The Area Attribute", Section 5).

The general form of an indirect reference is:

```
indirect-reference::=
    variable @ [(expression)]
```

When the (expression) option is used, it indicates an index on the indirect reference (post indexing).

The use of the indirect reference will be illustrated by the following example.

Consider the following declarations and assignments.

```
DECLARE  X(1) POINTER TO WORD;
DECLARE  A WORD;
DECLARE  B(2) WORD;
```

```
DECLARE  Y POINTER TO WORD
X (0) = @A;
X (1) = @B(0);
Y     = @B(0);
```

Then, subsequent to the above statements, and within their scope:

```
 X(0)@     is a reference to the value of A
 X(0)      is a reference to the value of X(0)
           (i.e., it is a reference to the storage location of A)
Y@(1)      is a reference to the value of B(1)
X(1)@(2)   is a reference to the value of B(2)
```

3.2.4 SUBFIELD REFERENCE

If a data item has been declared WORD, then the reference (simple, subscripted, or indirect) may be qualified with a field-select.

The general form of a field-select is:

```
field-select::=
  $(expression:expression)
```

Subfield qualification is accomplished by writing the field-select after the reference to the data item.

The first expression of a field-select specifies the number of bits to be referenced. The second expression specifies the right most bit position of the field. Bits within a word are numbered in descending order from left to right starting with 15 and ending with 0.

Examples:

```
X$(1:0)    references the least significant bit of the WORD X.
A$(4:12)   references a field consisting of the 4 most significant bits of
           the WORD A.
```

3.3 DATA TYPES

There is no data type for variables in MPL. The type of operations to be performed on the data is determined by the operators of the language, not by the operands. For example if the variable A and B appear in the expression:

    A + B

then the values will be considered to be binary integers and will be combined by the rules of algebraic addition. On the other hand if the same values appear in the expression

    A & B

then the values will be treated as bit strings and will be combined bit by bit according to the rules for the operator Logical And.

Although there is no data type associated with variables, there is a type associated with the representation of constant data. It should be noted that different representations of a constant may have the same value. For example the two constants:

    255
    "FF"

represent the same value and using either representation in a source program will have an identical effect.

The following paragraphs describe the various representations of constants.

3.3.1 DECIMAL NUMBERS

General format:

```
decimal-number::=
        digit...
```

A decimal-number is treated as an integer. The precision is self-defining. The possible precisions are 4, 8, 16 and 32 bits.

Examples:

12	precision used is 4 bits
200	precision used is 8 bits
15972	precision used is 16 bits
87962	precision used is 32 bits

3.3.2 BIT STRINGS

General format:

```
bit-string::=
        "{[(size)]  legal-digit... }..."
size::=
        1|2|3|4
```

The size determines the number of bits that will be generated from each of the legal-digits. The legal-digit must correspond with size as shown in the table below.

Table 6. Legal Digits for Bit Strings

SIZE	LEGAL DIGIT
1	0|1
2	0|1|2|3
3	0|1|2|3|4|5|6|7
4	0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

when size option is omitted, a default of 4 is used.

Examples:

bit string	binary value	precision
"32"	00110010	8 bits
"(3)32"	00011010	8 bits
"(1)12"	(invalid)	
"FA(1)1010(2)21"	1111101010101001	16 bits

3.3.3 CHARACTER STRINGS

General format::=

'[character] ...'

The character symbols that may appear include any characters available at the keyboard used to prepare the source program. When a single quote character is required in the character-string it must be represented by two consecutive single quotes.

Example:

'THIS IS A '' character-string'''

represents the characters:

THIS IS A 'character-string'

Note that lower-case characters are permitted in character-string. Also note the use of consecutive single quotes within the character-string.

Section 4
DATA MANIPULATION

4.1 EXPRESSIONS

An expression describes an algorithm for computing a value. Expressions are of two types: simple and conditional. Each operation performed in evaluating an expression is carried out with precision WORD or DOUBLE. The precision used is determined by the precision of the operands. If the operands are WORD or BYTE, then the precision used is WORD. If both operands are precision DOUBLE, then the precision used is DOUBLE. If one operand is precision DOUBLE and the other is either WORD or BYTE, then the lesser precision operand is converted to DOUBLE and the precision used is DOUBLE.

4.1.1 SIMPLE EXPRESSIONS

Simple expressions have a form similar to the formulas of algebra. A simple expression consists of a sequence of one or more operands separated by infix operators. Additionally an operand may be preceded by prefix operators. The operands of an expression may be data references or constants. Additionally an expression enclosed in parenthesis may be used as an operand. The following are examples of valid MPL expressions:

```
 A
-A
 A+B
 A+B*C
 A>3&C<D
 (A+B) * (C-D)
 A+-B
```

4.1.1.1 ARITHMETIC OPERATIONS. Arithmetic operations treat the operands as signed integers. BYTE (8 bit) precision operands are treated as 8 bit positive values and are converted to WORD (16 bit) precision by extending the precision

with zeros to the left. WORD precision variables are converted to DOUBLE precision by extending the most significant bit (sign bit) to the left. Negative values are represented in two's complement binary notation.

The following infix operators are used to indicate arithmetic operations.

+	indicates addition. The precision is determined by the operands.
-	indicates subtraction. The second operand is subtracted from the first. The precision is determined by the operands.
*	indicates multiplication. The precision is determined by the operands.
/	indicates division. The first operand is divided by the second. The precision is determined by the operands.
MOD	indicates remainder after division of the first operand by the second. The precision is WORD. The sign is the sign of the divisor.
MULD	indicates double-precision multiplication. The result of this operation is precision DOUBLE even though the operands are of precision WORD or BYTE.
DIVD	indicates double precision division. The first operand is divided by the second. The result of this operation is precision WORD. It is used when a DOUBLE precision operand is divided by a WORD or BYTE precision operand and the desired result is WORD.

The following are the prefix operators used to indicate arithmetic operations.

+	no effect on the value of the operand.
-	negation. The algebraic sign of the operand is changed.

4.1.1.2 COMPARISON OPERATIONS. The comparison operations are all infix operations. Comparison operations treat the operands as signed numbers in two's complement binary notation. The result of the operation is the value 1 if the relation is true. The result of the operation is the value 0 if the relation is false.

The following are the operators are used to indicate comparison operations.

- `>` indicates greater than
- `>=` indicates greater than or equal to
- `=` indicates equal to
- `¬=` indicates not equal to
- `<=` indicates less than or equal to
- `<` indicates less than

Examples:

5 > 3 has the value 1

2 > 2 has the value 0

3 >= 3 has the value 1

4.1.1.3 LOGICAL OPERATIONS. Logical operations operate on the operands on a bit by bit basis. The result in each bit position is determined by the values of the corresponding bit positions of the operands according to the following table.

Table 7. Logical Operations

A	B	NOT A	NOT B	A AND B	A OR B	A EXCLUSIVE OR B
1	1	0	0	1	1	0
1	0	0	1	0	1	1
0	1	1	0	0	1	1
0	0	1	1	0	0	0

The following infix operators are used to indicate logical operations:

	indicates logical or
&	indicates logical and
XOR	indicates logical exclusive or

The following prefix operator indicates a logical operation

¬ indicates logical not

Examples:

assume

A has the value "(1)00010111"

B has the value "(1)11111111"

C has the value "(1)10100000"

then

¬A	has the value "(1)11101000"
B&C	has the value "(1)10100000"
A|¬C	has the value "(1)01011111"
A XOR B	has the value "(1)11101000"

4.1.1.4 SHIFT OPERATIONS. Shift operations cause the first operand to be shifted the number of bit positions equal to the value of the second operand.

The following infix operators are used to specify shift operations.

SRA indicates shift-right-arithmetic. Bit shifted off the right are lost. Bits positions vacated on the left are filled with the sign of the original value.

SRL Indicates shift-right-logical. Bits shifted off the right are lost. Bit positions vacated on the left are filled with zeros.

SLL indicates shift-left-logical. Bits shifted off the left are lost. Bit positions vacated on the right are filled with zeros.

SLC indicates shift-left-circular. Each bit shifted off on the left fills the bit position vacated on the right.

Examples:

-25 SRA 2	has the value -7
"FFFF" SRA 7	has the value "FFFF"
"FA" SLL 1	has the value "1F4"
"8000" SLC 2	has the value "0002"

4.1.1.5 THE ASSIGNMENT OPERATION. The assignment operation is used to create the side effect of storing the value of the expression that appears to the right of the assignment operator. The assignment operator is:

:=

This operation can only be used in an expression in the form:

storage-reference := expression

That is, the operand to the left of the assignment operator must be a storage-reference. An expression or a constant may not appear to the left of the assignment operator. The assignment may only be made to WORD variables.

Examples:

X:=3+5	X is assigned the value 8. The value of the expression is 8
X:=Y:=7-2	X and Y are each assigned the value 5. The value of the expression is 5.
(X:=7)+X+2	X is assigned the value 7. The value of the expression is 16.

4.1.2 CONDITIONAL EXPRESSIONS

General form:

conditional-expression::=

IF expr-a THEN expr-b ELSE expr-c

where "expr-a", "expr-b" and "expr-c" are arbitrary expressions including the possibility of conditional expressions.

Conditional expressions are interpreted as follows. If the value of "expr-a" is an odd number (least significant bit is a 1) then the value of the "condition-expression" is the same as the value of "expr-b". If the value of "expr-a" is an even number (least significant bit is a zero) then the value of the "conditional-expression" is the same as the value of "expr-c". In the evaluation of a conditional expression only one of the, expr-b expr-c, expressions is evaluated.

Normally "expr-a" in the "conditional-expression" will be an expression with comparison operators. However, this need not be the case.

Examples:

```
IF A > B THEN C*D+2 ELSE C/D
IF A THEN B ELSE A
IF A THEN A ELSE B
```

Note that if the range of values for A and B is restricted to 0 and 1, then the value of the second example is identical to the value of the expression:

```
A & B
```

and that the value of the third expression is identical to the value of the expression:

```
A | B
```

4.2 EVALUATION OF EXPRESSIONS

Operations within an expression are assigned a priority as follows:

unary :	+,-,¬	highest
multiplication and shift:	*,/,MOD,DIVD,MULD,SRA,SRL,SLL,SLC	↓
addition:	+,-	
comparison:	>,>=,=,¬=,<=,<	
and	&	
or	|, XOR	
assignment	:=	lowest

operations within an expression are performed in the order of decreasing priority. For example, in the expression A+B*C multiplication of B by C is performed first and then the result is added to A. Consecutive assignment operations are performed in right to left order. All other infix operations of the same priority are performed in left to right order. Consecutive prefix operations are performed in right to left order.

If an expression is enclosed in parentheses, it is treated as a single operand. The parenthesized expression is evaluated before its associated operation is performed. For example, in the expression

    (A + B) * (C + D)

B will be added to A, D will be added to C and the first result will be multiplied by the second. Thus the use of parentheses may modify the normal priority of the operators.

The operands of an expression are always evaluated in left to right order. This is true regardless of the order in which the operands themselves are combined with operators. For example if A, B, and C represent operands to be evaluated (e.g. expressions in parentheses or function references), then the expression:

    A + B * C

is evaluated in the following steps:

1. A is evaluated
2. B is evaluated
3. C is evaluated
4. The multiplication of B by C is performed
5. The result of the multiplication is added to A

The expression:

    A * B + C

is evaluated in the following steps:

1. A is evaluated
2. B is evaluated

3. The multiplication of A by B is performed
4. C is evaluated
5. C is added to the result of the multiplication.

This strict left to right evaluation of operands guarantees the programmer can control side effects.

Section 5
DATA DESCRIPTION

5.1 ATTRIBUTES

An identifier in an MPL program may represent one of several types of objects. It may represent a data variable, a procedure name or a statement label. Those properties that characterize the object represented by the identifier, and other properties of the identifier (such as precision and accessing method) make up a set of attributes of the identifier.

When an identifier is used in a program, the attributes of the identifier must be known.

Examples of Attributes:

EXTERNAL - This attribute defines an identifier to have a certain special scope.

CONSTANT (5) - This attribute defines an identifier to be the name of a constant data item with the value of 5.

DOUBLE - This attribute defines an identifier to have a precision of 32 bits.

5.2 DECLARATIONS

An identifier is established as the name of some object and the attributes of the identifier are specified by means of a declaration.

If a declaration of an identifier is internal to a certain block, then the identifier is said to be declared in that block.

In a given program an identifier may represent more than one object. In this case each different object represented by the identifier is said to be a different use of the identifier. For example an identifier may represent a data item with precision BYTE in one part of the program and the same identifier

may represent a statement label in another part of the program. These two parts of the program, of course, cannot overlap.

Each use of an identifier is established by a separate declaration. References to different uses of the same identifier are distinguished by the rules of scope (See "scope of declarations" in this section).

Declarations are made by the use of the "declare-statement" or by the appearance of an identifier as a label of a statement.

5.2.1 THE DECLARE STATEMENT

Function:

The declare-statement is a non executable statement used to establish an identifier and to specify the attributes of the identifier.

General Format:

```
declare-statement::=
        DECLARE item-spec [, item-spec] ...;
item-spec::=
        {item type-attribute}|
        {item size-attribute [area-attribute]}|
        {identifier LITERALLY string}
item::=
    identifier [(constant)]
```

Syntax Rules:

1. Any number of identifiers may be declared in a single declare-statement and declarations must be separated by commas.

2. Attributes must follow the items to which they refer. (Note the above general form does not show the factoring of attributes which is allowed and explained later).
3. A label is not allowed on a declaration-statement.

General Rules:

1. All of the attributes of a given identifier must be declared in a single declare-statement.
2. Attributes of an EXTERNAL name declared in separate blocks and compilations must be identical.
3. The declaration-statements within any block must follow immediately after the block heading statement and before any other statements in the block.

Example:

```
DECLARE JOE BYTE, JIM WORD, SAM (15) BIT (4);
```

JOE is declared to be a data-item with a precision of 8 bits. JIM is declared to be a data-item with a precision of 16 bits. SAM is declared to be an array of 16 data-items each item with a precision of 4 bits.

5.2.2 FACTORING OF ATTRIBUTES

When several data-items have the same attributes (other than the dimension attribute), then the attributes can be factored to eliminate repeated specification of the same attribute for many identifiers. The factoring is accomplished by enclosing the items in parentheses and following them with the attributes that apply to all the items. The items within the parentheses are separated by commas.

Example:

```
DECLARE (A,B) WORD, (D,E) BYTE EXTERNAL;
```

This declaration is equivalent to the following:

```
DECLARE A WORD, B WORD, D BYTE EXTERNAL, E BYTE EXTERNAL;
```

5.2.3 MULTIPLE DECLARATIONS

More than one declaration of the same identifier, internal to the same block constitutes a multiple declaration of that identifier. Multiple declarations are in error with the exception that the declaration of an identifier as a PROCEDURE in a declare-statement does not constitute a multiple declaration if the same identifier appears subsequently as the label of a procedure-statement.

5.2.4 PROCEDURE LABELS

A label on a procedure statement declares the identifier as an entry-name. If the procedure is the outermost block, the EXTERNAL attribute is also implicitly declared for the label.

5.2.5 SCOPE OF DECLARATIONS

When a declaration of an identifier is made, there is a certain well defined region of the program over which this declaration is applicable. This region is called the scope of the declaration. Outside the scope of the declaration, the identifier is said to be unknown or undefined.

The scope of a declaration of an identifier is defined by the block, B, in which the identifier is declared but excluding any blocks contained in B where the same identifier is declared again.

5.2.5.1 SCOPE OF EXTERNAL NAMES. In general, declarations of the same identifier made in different blocks represent different distinct objects with non overlapping scopes. It is possible to declare the same identifier in more than one block such that each declaration represents the same object. This is done by using the EXTERNAL attribute.

When the same data item is declared EXTERNAL in more than one block, each declaration represents the same object.

If an identifier is declared with the type-attribute EXTERNAL PROCEDURE, then the identifier represents the external procedure whose label is the same as the identifier being declared.

The following examples illustrate scope of declarations. Tables 8 and 9 explain the scope and use of each name for example 1.

Example 1:

```
A:  PROCEDURE;                                  /*LINE 1 */
    DECLARE (X, Z) WORD;                        /*LINE 2 */
    B:  PROCEDURE (Y);                          /*LINE 3 */
        DECLARE Y BYTE;                         /*LINE 4 */
        C:  BEGIN;                              /*LINE 5 */
            DECLARE (A, X) DOUBLE;              /*LINE 6 */
              .
              .
            Y: Z=A;                             /*LINE 7 */
            END C;
        END B;
    D:  PROCEDURE;                              /*LINE 8 */
        DECLARE X WORD;                         /*LINE 9 */
        Y = Z + X;                              /*LINE 10, WITH ERROR */
          .
          .
        END D;
          .
          .
END A,
```

Table 8. Scope and Use of Names of Example 1

LINE	NAME	USE	SCOPE (BY BLOCK LABELS)
1	A	external procedure	all of A except C
2	X	WORD data-item	all of A except C and D
2	Z	Word data-item	all of A
3	B	internal procedure	all of A
4	Y	BYTE data-item	all of B except C
5	C	Statement label	all of B
6	A	DOUBLE data-item	all of C
6	X	DOUBLE data-item	all of C
7	Y	Statement label	all of C
8	D	internal procedure	all of A
9	X	WORD data-item	all of D

Table 9. Items Referenced By Example 1

LINE OF REFERENCE	NAME	ITEM REFERENCED
7	Z	WORD data-item declared on line 2
7	A	DOUBLE data-item declared on line
10	Y	Invalid reference. Y is not known in block D
10	Z	WORD data-item declared on line 2
10	X	WORD data-item declared on line 9

Example 2:

```
A:  MAIN PROCEDURE;
    DECLARE X WORD EXTERNAL;                          /*LINE 1 */
       .
       .
       .
    B:  PROCEDURE;
        DECLARE X BYTE;                               /*LINE 2 */
           .
           .
           .
        C:  BEGIN;
            DECLARE X WORD EXTERNAL;                  /*LINE 3 */
               .
               .
               .
            END C;
        END B;
    END A;
D:  PROCEDURE;
    DECLARE X DOUBLE;                                 /*LINE 4 */
       .
       .
       .
    E:  PROCEDURE;
        DECLARE X WORD EXTERNAL;                      /*LINE 5 */
           .
           .
           .
        END E;
    END D;
```

In example 2 there are five declarations of the identifier X.

The declaration of line 2 declares X as a BYTE data-item. Its scope is all of block B except block C.

The declaration of line 4 declares X a DOUBLE data-item. This item is distinct from that of line 2. Its scope is all of block D except block E.

Declarations in lines 1, 3, and 5 all declare X to be the _same_ WORD data-item. Its scope is all of the program except the scopes of declarations in lines 2 and 4.

5.2.5.2 BASIC RULE FOR THE USE OF IDENTIFIERS. The fact that an identifier is unknown outside its scope suggests the following basic rule on the use of identifiers.

> All appearences of an identifier which are intended to represent a given object in a program must lie within the scope of that identifier.

The most important implication of the above rule is on the limitation of transfer of control by the statement, GOTO LAB, where LAB is a statement-label.

The statement, GOTO LAB, internal to a block B, can cause transfer of control to another statement (having label LAB internal to B) or to a statement in a block that contains B, and to no other statement. In particular it cannot transfer control to any statement internal to a block contained in B.

5.2.6 THE ATTRIBUTES

Attributes are used to give characteristics to their associated identifiers. The attributes are divided into the following classes:

- type-attribute
- dimension-attribute
- size-attribute
- area-attribute
- literally-attribute

5.2.6.1 THE TYPE ATTRIBUTES. The type attributes are PROCEDURE and MICRO. Identifiers are declared to be procedure-names or micro-procedure-names if they appear in a declaration with a type-attribute. All identifiers appearing in a declaration without the type-attribute are declared to be data-items.

5.2.6.2 THE PROCEDURE-ATTRIBUTE.

Function:

The procedure-attribute specifies that the identifier is a entry-name.

General Format:

```
procedure-attribute::=
     [EXTERNAL] PROCEDURE [BYTE|WORD|DOUBLE]
```

General Rules:

Identifiers must be declared with the procedure-attribute in two cases

1. A reference is made to an external procedure other than the one containing the reference by either a call-statement or a function reference. In this case the EXTERNAL option is required in the declaration.
2. A reference is made to an internal-procedure by either a call-statement or a function reference appearing in the scope of the declaration and the reference occurs prior to the procedure itself.

The BYTE, WORD, or DOUBLE option is required with the procedure attribute if the procedure is to be referenced as a function. In this case the option specifies the precision of the value returned by the function.

Example:

```
A: PROCEDURE;
   DECLARE B PROCEDURE, C PROCEDURE,
           D EXTERNAL PROCEDURE;
   CALL D;
   CALL B;
   B: PROCEDURE;
          .
          .
          .
      CALL D;
      END B;
```

```
        C:  PROCEDURE;
                 .
                 .
                 .
            END C;
        CALL C;
        END A;
    D:  PROCEDURE;
          .
          .
          .
        END D;
```

In the example the declaration "B PROCEDURE" is required because the "CALL B;" statement occurs prior to the occurence of the procedure-block B. The declaration "C PROCEDURE" is not required (however it is allowed) because the "CALL C" statement occurs after the procedure-block C. The declaration "D EXTERNAL PROCEDURE" is required because the external-procedure D is referenced from block A which is not in the scope of block D.

5.2.6.3 THE MICRO ATTRIBUTE.

Function:

The micro-attribute allows the programmer to access micro-coded procedures of the 32/S that are not accessed by any of the normal MPL constructs.

General Format:

```
micro-attribute::=
     (constant) MICRO [BYTE | WORD | DOUBLE]
```

General Rules:

The constant represents the address in control storage where the microcoded process begins.

The BYTE, WORD, or DOUBLE option is required if the process leaves a value in the stack of the 32/S. If this option is not given it is assumed that the process does not place a value in the stack.

Example:

```
DECLARE ABC ("3F2") MICRO WORD;
VALUE = ABC (I,J);
```

The first statement in the example declares the identifier ABC to be a micro-coded procedure. The WORD option specifies that the procedure will leave a word result on the top of the 32/S stack. The second statement invokes the procedure. The value of the variables I and J are placed on the 32/S stack before the procedure is invoked.

5.2.6.4 THE DIMENSION ATTRIBUTE

Function:

The dimension attribute declares an identifier to be an array name and specifies the upper bound of the array.

General Format:

```
dimension-attribute::=
    (constant)
```

General Rule:

The constant specifies the upper bound of the array. The upper bound is the largest value of a subscript that may be used to reference an element of the array. The lower bound of an array is always zero; therefore, the number of elements in an array in one greater than the value of the upper bound.

Example:

```
DECLARE A(10) WORD, B(5) BYTE;
```

In the example, A is a WORD array of 11 elements (upper bound = 10). B is a BYTE array of 6 words (upper bound = 5).

5.2.6.5 THE SIZE ATTRIBUTE

Function:

The size-attribute is required for the declaration of all data-item identifiers. The size-attribute specifies the precision of the data item. The size-attribute is also used as an option of the type-attribute (see "Type Attribute" in this section).

General Format:

```
size attribute::=
        BYTE | WORD | DOUBLE | BIT ({1|2|4})|
        POINTER TO {WORD | BYTE | DOUBLE}
```

General Rules:

1. The size-attribute must be given with the declaration of all data-item identifiers.
2. The BIT option is only allowed if the dimension-attribute is also given.
3. The precision implied by the various size-attribute options are:

BYTE	8 bits of precision
WORD	16 bits of precision
DOUBLE	32 bits of precision
BIT (1)	1 bit of precision
BIT (2)	2 bits of precision
BIT (4)	4 bits of precision
POINTER TO	16 bits of precision

Example:

```
DECLARE A(7) BIT (4), B BYTE, C WORD,
        D POINTER TO BYTE;
           .
           .
           .
        C = D;
        B = D@;
```

In the example A is an array of 8 data-items each with a precision of 4 bits, B is a data-item with a precision of 8 bits, C is a data-item with a precision of 16 bits, and D is a item with a precision of 16 bits. The fact that D is declared a POINTER TO BYTE indicates that it may be used for indirect referencing.

The assignment statement C=D; of the example assigns the value of the 16 bit item D to the 16-bit item C. The assignment statement B=D@; assumes that the value of D is the location of an 8 bit item. That 8-bit item is assigned to the 8-bit item, B.

5.2.6.6 THE AREA ATTRIBUTE

Function:

The area-attribute is used to declare identifiers as occupying an area of storage outside the area implied by the normal dynamic storage allocation rules. (See storage allocation in section 7).

General format:

```
area-attribute::=
   EXTERNAL|BASED {constant|identifier}|
   CONSTANT {string | (constant [, constant]...)}
```

General Rules:

1. The EXTERNAL attribute declares a identifier to be identical to the item of the same name declared EXTERNAL in another block. (see "Scope of External Names" in this section).

2. The BASED attribute is used to assign absolute storage locations to identifiers. The value of the {constant|identifier} is multiplied by 4 to generate the base for an absolute address. The BASED attribute can only be used in programs that operate in executive mode.
3. The CONSTANT attribute allows the programmer to assign names to constants.
4. The string option may only be used with the BYTE size-attribute. The string option assumes the dimension-attribute. If the dimension-attribute is not given, the upper bound is set equal to the number of characters in the string. The first value of the array is set equal to the number of characters of the string.

Example 1.

```
DECLARE TTY (7) BYTE BASED "F800";
```

This example declares the BYTE array TTY to be based at the absolute location "F800" *4. If a reference is made to the item TTY (6), the actual address to be referenced is computed by multiplying "F800" by 4 and then adjusting the result by the index value of 6.

Example 2.

```
DECLARE WORK_AREA (1000) WORD BASED STACK;
```

In this example the identifier STACK must have been previously declared as a word variable. If a reference is made to WORK-AREA (1), the actual address to be referenced is computed by multiplying the value of the data-item STACK by 4. The result is adjusted by the subscript 1, to form the absolute address for the reference.

Example 3.

```
DECLARE TEXT CONSTANT 'ERROR MESSAGE';
```

This example generates an array of byte constants. TEXT (0) is the constant 13 (the number of bytes in the string). TEXT (1) is the constant "45" (the ASCII

Section 6
PROCEDURES, FUNCTIONS, AND SUBROUTINES

6.1 PARAMETERS

The procedure-statement that heads a given procedure may specify a parameter-list (see section 8 for the syntax and details of the procedure-statement).

Parameters are identifiers and may represent scalar variable names or array names used in the procedure. The appearance of an identifier in the parameter-list declares the identifier as a parameter. This declaration causes the scope of the parameters to be the procedure-block. Identifiers that appear as parameters must also appear in a declare-statement. The declare-statement must assign a size-attribute and optionally, a dimension attribute to the identifier.

Example:

```
ABC:  PROCEDURE (X,Y,Z);
 DECLARE (A,Z) WORD, (X,Y(6)) BYTE;
```

In this example there are four variables declared in the scope of the block ABC. Three of the variables X, Y, and Z are declared to be parameters by virtue of their appearance in the parameter-list of the procedure-statement. The fourth variable, A, is not a parameter.

The use of the dimension attribute for the parameter, Y, merly identifies Y as an array. No storage for the array is allocated by the declaration. The storage is allocated by the calling procedure.

6.2 PROCEDURE REFERENCES

The label appearing on a procedure is called the procedure entry-name. At any point in a program where an entry name for a given procedure is known, the

procedure may be invoked by a procedure-reference. The procedure reference has the form:

entry-name [(argument [,argument]...)]

The number of arguments in the procedure-reference (possibly zero) should be equal to the number of parameters in the parameter-list of the procedure being invoked.

When a procedure-reference invokes a procedure, each argument specified in the reference is associated with a formal parameter in the corresponding position of the parameter-list of the denoted procedure. Control is then passed to the procedure. The manner of associating arguments with parameters is discussed under the heading "arguments in a procedure reference" in this section.

There are two distinct uses of procedures, determined by the way in which they are referenced.

1. A procedure reference may appear as an operand in an expression. In this case the reference is said to be a function reference and the procedure is said to be invoked as a function.
2. The procedure may appear after the keyword CALL in a call-statement. In this case the reference is said to be a subroutine reference and the procedure is said to be invoked as a subroutine.

Ordinarily a given procedure will be used exclusively as a function or exclusively as a procedure. However, it is not mandatory that this be the case.

6.2.1 FUNCTION REFERENCE

When a function reference appears in an expression, the function procedure is invoked. The procedure is then executed, using the arguments (if any) that were specified in the function-reference. The result of this execution is the required value of the function which is passed with return of control back to

the point of invocation. This value is then used in place of the function reference, as an operand, and the evaluation of the expression continues.

The procedure invoked by the function-reference normally will terminate with the execution of a statement of the form:

RETURN expression ;

It is the value of the expression appearing in the return-statement that is returned as the function value.

If the invoked function terminates with a goto-statement, the evaluation of the expression that invoked the function will not be completed (imbedded assignments that occurred before the invocation will be performed), and control will pass to the point specified by the goto-statement.

If the invoked function terminates with an end-statement, evaluation of the expression containing the function reference continues, however, the value of the function in this case is undefined. This type of termination of a function is normally an error.

6.2.2 BUILT IN FUNCTIONS

Besides functions written by the programmer, a function reference may invoke one of several built-in functions.

The built-in functions are an intrinsic part of MPL. Entry-names for built-in functions are not declared by the programmer. However, if the programmer declares an identifier that is identical to a built-in function name, the normal scoping rules apply and the built-in function cannot be invoked in the scope where the programmer's identifier is known.

Each built-in function has a specific number of arguments and returns a value of a specified precision. The table below summarizes the built-in functions.

Table 10. Built-In Functions

FUNCTION NAME	ARGUMENT SIZE	RETURN SIZE	DESCRIPTION
SUPERVISOR	WORD	null	Generates a supervisor call instruction
OVERFLOW	None	WORD	Returns the state or the arithmetic overflow indicator 1 is on. 0 is off.
RESUME	WORD	null	Resume the process whose stack base is the argument. Can be executed only in executive mode.
HIGH	DOUBLE	WORD	Returns the most significant half of the argument.
LOW	DOUBLE	WORD	Returns the least significant part of the argument.
ABS	WORD	WORD	Returns the absolute value of the argument.
XIM	WORD	WORD	Places the argument into the interrupt mask register. Returns the previous contents of the interrupt mask register
CARRY	none	WORD	Returns the status of the carry indicator. One (1) indicates on, 0 indicates off.
SWITCHES	none	WORD	Returns the value in the configuration switch register.
NOP	none	null	Generates a NOP instruction.
PNOP	none	null	Generates a PNOP instruction.
WAIT	none	null	Generates a WAIT instruction.
TRAP	none	null	Generates a TRAP instruction.
PRTNUM	procedure-name	WORD	Returns the procedure reference table entry number of the argument.

6.2.3 SUBROUTINE REFERENCES

When a procedure is invoked as a subroutine by the execution of a call-statement, the arguments (if any) are associated with the formal parameters and control passes to the called subroutine.

Unlike the function, subroutines do not return a value to the point of invocation. Subroutines may terminate in the following ways:

1. Control reaches a return-statement. When this occurs the expression in the return-statement (if present) is evaluated. The value of the expression is lost. Control then passes to the first statement following the invoking call-statement.
2. Control reaches a goto-statement. In this case control passes to the point specified by the goto-statement.
3. Control reaches an end-statement. In this control passes to the first statement following the invoking call-statement.

6.3 PROCEDURE REFERENCE EXAMPLES

Example of a function reference:

```
COMP: PROCEDURE;
        DECLARE (P,Q,R,V) WORD;
        POLY: PROCEDURE (C,X);
              DECLARE (C,X) WORD;
              RETURN (C + X * ( 1 + X * (2 + X)));

              END POLY;
               .
               .
               .
        S1: P = Q * POLY (R,V);
               .
               .
               .
        END COMP;
```

In this example, the external-procedure COMP contains the function POLY which is invoked when the expression appearing in the statement labeled S1 is being evaluated. When the procedure POLY is invoked the values of the arguments R and V will be substituted for the parameters C and X respectively.

Example of a subroutine reference:

```
COMP:  PROCEDURE;
       DECLARE (P,Q,R,V, TEMP) WORD;
       POLY:  PROCEDURE(C,X);
                DECLARE (C,X) WORD;
                TEMP= C+X*(1+X*(2+X));
                END POLY;
       S1:  CALL POLY (R,V);
                :
       S2:  P=Q * TEMP;
                :
       END COMP;
```

In this example, the effect is the same as in the previous example. The subroutine procedure POLY is invoked by the call-statement labeled S1. POLY computes the polynominal and assigns it to the variable TEMP. Then control passes to the statement labeled S2. This statement then uses the value placed in TEMP to compute the final result. Thus the value of the polynominal is communicated thru the variable TEMP. This is possible because the name TEMP is known to both the procedures COMP and POLY, and by the rules of scope TEMP represents the same object to both procedures.

In some cases the invoked procedure cannot share a variable with the invoking procedure. For example it may be that both procedures are external and then

by definition it is not possible to share a variable (see scope of declarations). Another more general way of returning values from subroutines in such cases is shown in the following example:

```
COMP:  PROCEDURE;
       DECLARE (P,Q,R,V, TEMP) WORD;
       DECLARE POLY EXTERNAL PROCEDURE;
  S1:  CALL POLY (R,V,@TEMP);
          .
          .
          .
  S2:  P = Q * TEMP;
          .
          .
          .
       END COMP;
POLY:  PROCEDURE (C,V, PTR);
       DECLARE (C,V) WORD;
       DECLARE PTR POINTER TO WORD;
       PTR@ = C+X*(1+X*(2+X));
       END POLY;
```

In this example the call to POLY contains an additional argument, namely the location of the variable TEMP. POLY is declared with one additional parameter, PTR, which is a POINTER TO WORD. Then the value is returned indirectly via the PTR parameter.

6.4 ARGUMENTS IN A PROCEDURE REFERENCE

In general, an argument in a procedure-reference may be any one of the following:

1. A simple variable or subscripted variable
2. An expression.
3. An variable array name. An array name used as an argument is equivalent to a pointer to element zero of the array.

A constant array name may not be used as an argument in a procedure-reference.

The attribute of each argument in a procedure reference should match the attributes of the corresponding parameters.

For example, assume the procedure ABC in a program defined by:

```
ABC: PROCEDURE (A,X,Y,Z);
     DECLARE (X,Z) WORD, Y DOUBLE, A (0) BYTE;
       .
       .
       .
     END ABC;
```

This implies that the first parameter is used as a BYTE array, the second and fourth parameters are used as WORD scalars, and the third parameter is a DOUBLE scalar. If the subroutine ABC is invoked by the statement:

```
CALL ABC (P, B + 2, C-5, W);
```

It is assumed that:

1. P is the name of a byte array.
2. The expression B + 2 has precision WORD.
3. The expression C-5 has precision DOUBLE.
4. The variable W has precision WORD.

6.4.1 PASSING ARGUMENTS TO A PROCEDURE

When a procedure is invoked by a procedure reference and each argument is associated with its corresponding parameter, the arguments are said to be <u>passed</u> to the invoked procedure.

The mechanism used for passing arguments in MPL is referred to as <u>pass by value</u>. Passing by value is accomplished in the following way. The called procedure has a location of appropriate precision allocated for each parameter in the parameter-list. When a procedure-reference is encountered each argument is <u>evaluated</u>. The result of the evaluation is, in effect, stored in the location reserved for the corresponding parameter. This value is used in the same way as any variable declared in the called procedure. In other words a

parameter is a variable local to the called procedure which, upon entry, is initialized to the value of the corresponding argument.

In general a called subroutine cannot affect the value of a variable passed to it as an argument. For example, in the program:

```
A:  PROCEDURE;
    DECLARE (X,Y) WORD;
    B:  PROCEDURE (T);
        DECLARE T WORD;
        T = T + T;
        END B;
    S1:  X = 5 ;
    S2:  CALL B (X);
    S3:  Y = X;
         .
         .
         .
        END A;
```

The statement labeled S1 assigns the value 5 to the variable X. The subroutine B is passed the value 5 as an initial value for T. The subroutine then adds T to itself. However, this has no affect on the value of X. Therefore, when the statement S3 is executed, the value of the variable X is still 5.

Section 7
DYNAMIC PROGRAM STRUCTURE

7.1 PROGRAM CONTROL

Execution of a program is initialized by an operating system, which invokes the initial procedure. The initial procedure must be a main-procedure. When the program is being executed there is a control that determines the order of execution of the statements. For a discussion of the order of execution see "sequence of control" section 8.

7.2 ACTIVATION AND TERMINATION OF BLOCKS

A begin-block is said to be activated when control passes through the begin-statement for the block. A procedure-block is activated when it is invoked by a procedure-reference.

A block is active if it has been activated and has not yet terminated. The following rules describe the ways that a block may be terminated:

1. A begin-block is terminated when control passes through the end-statement of the block.
2. A procedure-block is terminated when control passes through a return-statement or the end-statement of the block.
3. Either type of block is terminated by execution of a goto-statement that transfers control to a point not contained in the block. The go-to statement may terminate more than one block (see the "goto-statement in section 8).

7.2.1 DYNAMIC DESCENDANTS

If a block B is active another block B1 may be activated from a point internal to block B while B still remains active. The following rules describe the case in which this will occur.

1. B1 is a procedure-block immediately contained in B (the label of B1 is internal to B) and is reached through a procedure-reference.
2. B1 is a begin-block internal to B and is reached through normal flow of control.
3. B1 is a procedure-block not contained in B and is reached through a procedure-reference. B1 in this case, may be identical to B, i.e., B is called recursively. However, it is still regarded as a dynamically different block.

In any of the above cases, while B1 is active it is said to be an _immediate dynamic descendant_ of B.

Block B1 may itself have an immediate dynamic descendant B2, etc., so that a chain of blocks (B,B1,B2...) is created, where all the blocks in the chain are active. In this chain each of the blocks B1, B2, etc., is a _dynamic descendant of B_.

It is important to note that the termination of a given block may imply the termination of other blocks and that these other blocks need not be contained in the given block.

7.3 ALLOCATION OF DATA

The simple static process of data allocation implied by many programming languages--the assignment of a distinct storage region for each distinct variable used in a source program--may be wasteful. Multiple use of storage for different data during program execution can reduce the total storage requirements of the program. MPL provides automatic allocation and release of storage during program execution, in order to minimize the use of storage.

7.3.1 DEFINITIONS AND RULES

Storage is said to be allocated for a variable when a region of storage is associated with it. Allocation of storage takes place dynamically, during program execution.

Storage that has been allocated for a variable may subsequently be released. Thus the storage is freed for possible use in a later allocation. If storage has been allocated and has not been substantially released, the variable is in the allocated state.

When a variable appears in an executable statement of a program, the appearance is called a reference.

> At any point where a variable is referenced, it must be in the allocated state.

Violation of the above rule is a program error. However, the error may not be detected.

7.3.2 STORAGE CLASSES

Every variable in a program has a storage class which specifies the manner of storage allocation.

There are two storage classes, static and automatic.

7.3.2.1 THE STATIC STORAGE CLASS. The storage for variables declared in the main-program is in the static class. Storage for these variables is allocated when the program begins execution and is never released during execution.

Variables declared EXTERNAL are also of the static class.

7.3.2.2 THE AUTOMATIC CLASS. Variables declared in any block except the main-program (or variables declared EXTERNAL) are in the automatic storage class. Whenever a block B is activated, storage for all variables declared in the block (including parameters) is allocated. The variables remain in the allocated state until termination of the block. At the time of termination storage for the variables is released. Thus the time interval during which the variable is in the allocated state includes the interval when the variable is known (see "Scope of Declarations").

Termination of a block by means of a goto-statement may imply termination of other blocks and, consequently, the simultaneous release of storage for all variables declared in these blocks.

If a block B is a procedure and is referenced from a statement contained in B or from a statement contained in a dynamic descendant of B, then the procedure B is said to be invoked recursively. Each recursive activation of a procedure causes the previous allocation to be "pushed-down" (assignments of values in the previous allocation are retained) and a new allocation for the variables declared in the procedure is made. On each return from the procedure the most recent allocation is released. Each invocation of the procedure is called a new generation of the procedure. References to data items declared internal to a procedure always reference the most recent generation of the procedure.

Once a block has terminated, the values assigned to the variables that were released by the termination becomes undefined. If the block is subsequently reactivated, the storage for variables is reallocated. However, the values assigned in the previous activation are not known in the current activation.

Example:

```
A: PROCEDURE;
B: BEGIN;
     DECLARE X(1000) WORD;
     CALL PROCESS (X);
     END B;
C: BEGIN;
     DECLARE Y(1000) WORD;
     CALL PROCESS (Y);
     END C;
   END A;
```

In this example the arrays X and Y are declared in separate begin-blocks. Since both blocks B and C cannot both be active at the same time, the storage for the array X and Y will not be allocated at the same time. Thus 1001 words are required for both arrays. This contrasts with the following example where X and Y are declared in the same block thus requiring 2002 words.

```
A: PROCEDURE;
     DECLARE (X(1000), Y(1000))WORD;
     CALL PROCESS (X);
     CALL PROCESS (Y);
     END A;
```

Section 8
STATEMENTS

This section gives a description of each statement in the language. The statement are described in alphabetical order.

8.1 RELATIONSHIP OF STATEMENTS

Statements may be classed into the following four logical groups:

1. Assignment
2. Control
3. Data Declaration
4. Block statements

8.1.1 ASSIGNMENT STATEMENTS

There are two types of assignment statements; assignment by replacement and assignment by addition. The assignment statements are used to evaluate expressions and to assign values to scalars and array elements.

8.1.2 CONTROL STATEMENTS

The control statements affect the normal sequential flow of control through a program. The control statements are GOTO, IF, DO, CALL, RETURN, and REPEAT.

8.1.3 DATA DECLARATION STATEMENT

The data declaration statement, DECLARE, specifies attributes of identifiers. This statement is described in section 5.

8.1.4 BLOCK STATEMENTS

The block statements are used to delimit procedure-blocks and begin-blocks. The block statements are BEGIN, PROCEDURE, EOF, and END.

8.2 SEQUENCE OF CONTROL

Within a block, control normally passes sequentially from statement to statement. If an internal procedure is encountered, control passes to the statement following the end of the procedure. Control passes to the statement following an if-statement when control reaches the end of the "THEN unit-1" (The "ELSE unit-2" is skipped in this case). This occurs if control reaches one of the statements of unit-1 as the result of a goto-statement that references a label of a unit-1 statement. Sequential operation is modified by the following statements: CALL, END, GOTO, PROCEDURE, and RETURN.

A call-statement passes control to the specified procedure.

A goto-statement causes control to transfer to the statement with the specified label.

A procedure-statement heads a procedure. Procedures are independent blocks and may be placed anywhere within an external procedure, consistent with the identifier scopes desired by the programmer. However, a procedure may be invoked only by a procedure-reference in a call-statement or an expression. Thus control passes around a nested procedure from the statement before the procedure statement to the statement following the end-statement of the procedure.

The RETURN statement returns control from a procedure to the invoking procedure.

The following conditions may modify the sequential statement execution:

1. A function reference in any expression causes control to pass to the specified procedure.
2. The flow of control through an if-statement and do-group may or may not be sequential.

The following program segment illustrates the sequence of control:

```
A:  PROCEDURE;
B:     X=Y+Z;
C:     CALL G;
D:     IF 2>1 THEN
E:     P=Q; ELSE
F:     P=R;
G:     PROCEDURE;
H:     S=T&P;
I.     RETURN;
J:     END G;
K:     GOTO N;
        .
        .
        .
N:  END A;
```

The statements are executed in the order A, B, C, G, H, I, D, E, K, N.

8.3 ALPHABETIC LIST OF STATEMENTS

8.3.1 THE ASSIGNMENT STATEMENT

Function:

The assignment statement evaluates an expression and to assigns the value to a variable.

General Format:

```
assignment statement::=
    storage-reference assign-operator expression;
assign-operator::=
    = | := | +=
```

Syntax Rules:

1. The assignment-statement is recognized by the absence of a statement-identifier keyword as the first identifier.
2. The storage-reference may be of any precision. Variables of WORD precision may contain a field-reference option.

The storage-reference may also specify an indirect reference.

General Rules:

1. If the storage-reference contains a subscript, the subscript expression is evaluated first. The expression to the right of the assign-operator is then evaluated.
2. If the assign-operator is either = or := then the value of the expression replaces the value previously assigned to the storage-reference. If a field-reference option is specified with the storage reference, then only the contents of that field is changed by the assignment.
3. If the assign-operator is += then the value of the expression is added to the value of the storage-reference. In this case the storage-reference must be of WORD precision and may not contain a field-reference option.
4. If an indirect storage-reference is used, the variable pointed to by the reference must be currently allocated.
5. If the storage-reference is of precision BYTE or BIT(n), then the expression is truncated on the left before the assignment is made. If the storage-reference is of precision WORD, BYTE, or BIT(n), and expression is of precision DOUBLE, then the expression is truncated on the left before the assignment is made. If the expression is of precision WORD and the storage-reference is of precision DOUBLE, then the expression is sign extended to the left before the assignment is made.

Example 1:

```
X=3+2;
```

This statement assigns the value 5 to the variable X.

Example 2:

```
W$(4:0) = "F";
```

This statement assigns the value "F" to bits 3 through 0 of W. Bits 15 through 4 of W are not changed.

Example 3:

```
PTR@ = X;
```

This statement assigns the value of X to the variable pointed to by the variable PTR.

Example 4:

```
X+=-1;
```

This statement causes the variable X to be decremented by 1.

8.3.2 THE BEGIN STATEMENT

Function:

The begin-statement is the heading of a begin-block.

General Format:

```
begin-statement::=
        BEGIN;
```

General Rules:

1. A begin-statement is used in conjunction with an end-statement to delimit a begin-block. See Section 2 for a discussion of blocks.
2. Declarations appearing in a begin-block must immediately follow the begin-statement with no intervening statements.

Example:

```
BEGIN;
DECLARE X(100) WORD;
   .
   .
   .
END;
```

8.3.3 THE CALL STATEMENT

Function:

The call statement invokes a procedure.

General Format:

```
call-statement::=
    CALL entry-name[(argument[, argument]...)];
```

Syntax Rules:

1. The entry-name represents the label on the procedure to be invoked.
2. Each argument may be any of the following: an expression, a scalar-constant name, an array-name or a pointer. A pointer is a variable preceded by the commercial at sign (e.g., @X is a pointer to X).
3. An array-name used as an argument is equivalent to a pointer to the array. For example if X is an array name then the following two statements are equivalent:

```
CALL (X);
CALL (@X(0));
```

Example:

```
CALL   SUB (A,@B, 'XYZ', X+3*Y);
```

8.3.4 THE DECLARE STATEMENT

Function:

The declare-statement is used to specify attributes for identifiers.

General format:

```
    declare-statement::=
      DECLARE item-spec [, item-spec]... ;
    item-spec::=
      item type-attribute |
      item size-attribute [area-attribute] |
      identifier LITERALLY string
```

General rules:

See Section 5 for a description of the declare-statement.

8.3.5 THE DO STATEMENT

Function:

The do-statement delimits the start of a do-group (see "Groups" in section 2) and may specify iteration of statements within a group or may specify the selection of one of the statements within the group. The end of the go-group is delimited by an end-statement.

General format:

There are 5 forms of the do statement.

```
    option 1:
       do-statement::=
               DO;
    option 2:
       do-statement ::=
           DO WHILE expression;
```

option 3:

```
do-statement::=
DO variable = expression-1 TO expression-2
          [BY expression-3];
```

option 4:

```
do-statement::=
    DO CASE expression;
```

option 5:

```
do-statement::=
  DO FOREVER;
```

Syntax rules:

1. The "variable" in option 3 must be a simple-variable. Indirect references or array-variables may not be used.
2. IF "BY expression-3" is omitted in option 3, expression-3 is assumed to have a value of one (1).

General rules:

1. In option 1 the do-statement delimits the start of the do-group. The statements in the range of this form of do-group are executed according to the normal sequence of control.
2. In option 2 the do-statement delimits the start of a do-group and also specifies an iteration as indicated below.

```
LABEL:  DO WHILE expression;
   L1:  statement-1
          .
          .
          .
        statement-n
        END LABEL;
NEXT:   statement
```

The effect of the above is exactly equivalent to the following expansion:

```
LABEL:  DO;
L1:     IF expression THEN
        DO;
          statement-1
            .
            .
            .
          statement-n
          GOTO L1;
         END;
        END LABEL;
NEXT:   statement
```

3. In option 3 the do-statement delimits the start of a do-group and specifies a controlled iteration as indicated below:

```
LABEL:  DO variable = expression-1
           TO expression-2
           By expression-3;
   L1:  statement-1
           .
           .
           .
        statement-n
      END LABEL;
 NEXT:  Statement
```

The effect of the above is equivalent to the expansion shown below where T1, T2, and T3 are temporary variables created by the compiler.

```
LABEL:  BEGIN;
        DECLARE (T1, T2, T3) WORD;
        T1 = expression-1;
        T2 = expression-2;
        T3 = expression-3;
        variable = T1;
```

```
L1:  IF (T3>=0) & (variable <=T2)|
        (T3<0)  & (variable >=T2) THEN
       DO;
         statement 1
          .
          .
          .
         statement n
         variable += T3;
         GOTO L1;
       END;
     END LABEL;
```

4. In option 4 the do-statement delimits the start of a do-group and selects a single group within the do-case-group for execution as specified below:

```
LABEL:  DO CASE expression;
          group-0
          group-1
            .
            .
            .
          group-n
        END;
NEXT:   statement
```

The above is equivalent to the expansion shown below where T1 is a temporary variable created by the compiler.

```
     LABEL BEGIN;
             DECLARE T1 WORD;
             IF expression =0 THEN
                 DO;
                  group-0
                  GOTO NEXT;
                 END;
             IF expression =1 THEN
                 DO;
                  group-1
                  GOTO NEXT;
```

```
              END;
                .
                .
                .
            IF expression = n THEN
               DO;
                 group-n
                 GOTO NEXT;
               END;
            END LABEL;
   NEXT:    statement
```

In option 4 if the value of the expression is negative or if the value is greater than n, the result is undefined. This is considered a program error. In option 4, group-0, group-1, etc., may be do-groups, repeat-groups, begin-blocks, or they may be simple statements.

5. In option 5 the do-statement delimits the start of a do-group and indicates an indefinite iteration as indicated below:

```
   LABEL:  DO FOREVER;
             statement 1
                .
                .
                .
             statement n
           END LABEL;
```

The above is equivalent to the following expansion:

```
   LABEL:  DO;
             statement 1
                .
                .
                .
             statement n
             GOTO LABEL;
             END LABEL;
```

Once the range of a do-forever-group has been entered execution of the group can only be terminated by a goto-statement that references a label outside the group, or by a return-statement.

Examples:

```
DO WHILE A > B;
DO 1 = 3 TO 9 BY 2;
DO WHILE   TAX-DEDUCT < GROSS & TAX-RATE > 0;
DO;
DO CASE I;
DO FOREVER;
```

8.3.6 THE END STATEMENT

Function:

The end-statement terminates blocks and groups.

General format:

```
end-statement::=
   END [label] ;
```

General rules:

1. The end-statement terminates that group or block headed by the nearest preceding do-statement, procedure-statement, begin-statement or repeat-statement for which there is no closer corresponding end-statement.
2. If a label follows END, it must correspond to a label on the group or block heading being terminated.
3. An end-statement can terminate only one group or block.
4. If control reaches an end-statement which terminates a procedure, it is treated as a return-statement.

8.3.7 THE EOF STATEMENT

Function:

The eof-statement is an optional statement that may appear after the end-statement that terminates an external procedure.

General format:

    eof-statement::=
            EOF

General rule:

The eof-statement is used to control the compiler. The statement forces the compiler to end the compilation of the current external procedure. The statement as no effect if the source program is properly constructed i.e., one end-statement for each block heading statement. The use of the eof-statement is suggested as a debugging aid.

8.3.8 THE GOTO STATEMENT

Function:

The goto statement causes control to be transferred to the statement referenced.

General format:

    goto-statement::=
    {GOTO | GO TO} label;

General rules:

1. The label may not be the label of a procedure statement.
2. A goto-statement may not transfer control into a group that specifies iteration.

3. A goto-statement that transfers control from a block B to a dynamically encompassing block A has the effect of terminating block B as well as all the other blocks that are dynamic descendents of the most recent activation of block A. Variables allocated in these blocks are freed in the same way as if the blocks had terminated normally.
4. When a goto-statement transfers control out of a procedure that was invoked by a function reference, the evaluation of the corresponding expression is discontinued and control passes to the specified statement.

Example 1

```
        GO TO L2;
            .
            .
            .
    L2: statement
```

Example 2:

```
    A:  BEGIN;
        statement
    B:  BEGIN;
          DECLARE X(100) BYTE;
          .
          .
          .
          GOTO C;
           .
           .
           .
          END B;
    C:  statement
          .
          .
          .
         END A;
```

The goto-statement in the second example passes control to a point outside of block B. Therefore, it has the effect of terminating block B and of freeing the storage allocated to the array X.

8.3.9 THE IF STATEMENT

Function:

The if-statement causes program flow to depend on the value of an expression.

General Format:

if-statement::=

    IF expression THEN unit-1 [ELSE unit-2]

Syntax rules:

1. Each "unit" is either a group or a begin-block (recall that a simple statement is a special case of a group), either of which is terminated by a semicolon.
2. The if-statement itself is not terminated by a semicolon. Instead the semicolon that terminates unit-1 (or unit-2) serves to terminate the if-statement.
3. Each unit may contain labels.

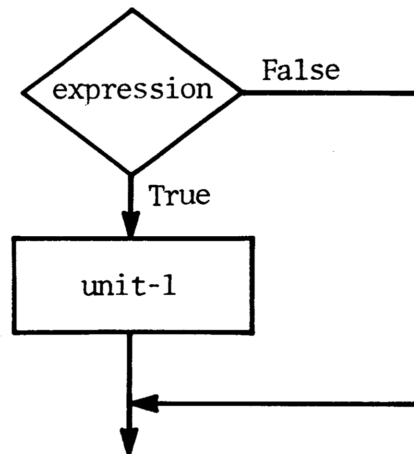
General Rules:

1. The expression following IF is evaluated. If the result of the expression evaluation is an odd number (least significant bit is 1), the expression is said to be <u>true</u>. If the result of the expression evaluation is an even number (least significant bit is 0) the expression is said to be <u>false</u>.
2. If the expression is true "unit-1" is executed and then control passes to the next statement ("unit-2" is skipped).
3. If the expression is false "unit-1" is skipped and control passes to unit-2. If "unit-2" is not given control passes to the following statement.
4. IF statements may be nested. That is either "unit-1" or "unit-2" may themselves be if-statements. When if-statements are nested the "ELSE unit-2" portion of a statement is always associated with the innermost unmatched "THEN unit-1". For this reason a null-statement may be required to specify a desired sequence of control.

The following two flowcharts illustrate the sequence of control for if-statements with and without the "ELSE unit-2" option.

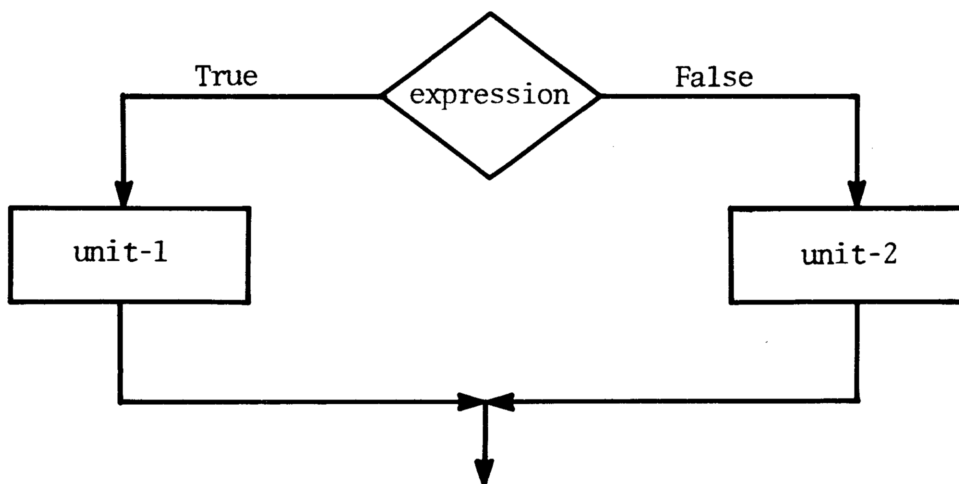
The first flowchart illustrates the case of:

IF expression THEN unit-1



The second flowchart illustrates the case of:

IF expression THEN unit-1 ELSE unit-2



Examples:

1.
```
IF SCAN_STACK=EMPTY THEN CALL GET_INPUT;
```

2.
```
IF X>Y|Y>Z THEN
   IF Z=W THEN
     IF W<P THEN Y=1;
     ELSE Y=2;
   ELSE;
ELSE Y=3;
```

8.3.10 THE NULL STATEMENT

Function:

The null statement causes no operation and does not modify the sequence of control.

General format:

```
null-statement::=
          ;
```

Example:

```
IF A>B THEN GOTO LABEL; ELSE;
```

The semicolon following ELSE is a null statement.

8.3.11 THE PROCEDURE STATEMENT

Function:

The procedure-statement has the following functions:

1. It heads a procedure block.
2. Defines the entry-name for the procedure.
3. Declares certain variables as having attribute "parameter".
4. Specifies the precision of the value to be returned if the procedure is to be invoked as a function.
5. Defines any special attributes of the procedure.

General format:

The procedure-statement has three formats.

Option 1

```
procedure-statement::=
entry-name: MAIN PROCEDURE;
```

Option 2

```
procedure-statement::=
entry-name: INTERRUPT PROCEDURE (parameter);
```

Option 3

```
procedure-statement::=
entry-name: PROCEDURE
          [(parameter [, parameter] ...)]
          [WORD|BYTE|DOUBLE] ;
```

General rules:

1. The procedure-statement is used in conjunction with an end-statement to delimit a procedure-block. See Section 2 for a discussion of blocks.
2. Any declarations appearing in a procedure block must immediately follow the procedure-statement without any intervening statements.
3. If parameters appear in the procedure statement, they must also appear in declaration statements that specify precision-attributes for them.
4. Option 1 specifies a procedure to be a main-procedure. The entry-name is the starting point for program execution. There may be only one main procedure in a program and the main program may not be called recursively.
5. Option 2 specifies a procedure to be an interrupt-procedure. A interrupt-procedure must have exactly one parameter. Interrupt procedures must be external procedures or they must be internal to an external procedure. An interrupt procedure is a special form of procedure that can be invoked outside the normal sequence of control. See the "32/S Reference Manual" for a discussion of interrupt-procedures.

6. The option BYTE, WORD or DOUBLE must be specified on a procedure that returns a value unless the option was specified in a previous declaration of the entry-name in a declare-statement.

Example

```
B: PROCEDURE;
   DECLARE (C,X,Y) WORD;
F: PROCEDURE (B,C) WORD;
   DECLARE (B,C) WORD;
      .
      .
      .
      RETURN B*C+5;
   END F;
L1: C=F(X+2,F(Y,X-1));
  END B;
```

The option WORD in the procedure-statement, F, specifies that when F is invoked a function it is to return a value with precision WORD. The statement,L1, invokes the function twice.

8.3.12 THE REPEAT STATEMENT

Function:

The repeat-statement delimits the start of a repeat-group and specifies repeated execution of the group.

General format:

```
repeat-statement::=
     REPEAT  expression  TIMES;
```

General rules:

The repeat statement delimits the start of a repeat-group. It also specifies an itteration as indicated below.

```
LABEL:  REPEAT  expression  TIMES;
         statement 1
           .
           .
           .
         statement n
         END LABEL;
```

The effect of the above is exactly equivalent to the expansion shown below where T1 is a temporary created by the compiler.

```
LABEL:  BEGIN;
        DECLARE T1 WORD;
        DO T1=1 TO expression;
         statement 1
           .
           .
           .
         statement n
        END;
       END LABEL;
```

Example:

```
REPEAT  IF X>Y THEN 5 ELSE 7  TIMES;
```

8.3.13 THE RETURN STATEMENT

Function:

The return statement terminates execution of a procedure and returns control to the point of invocation.

General format:

```
return-statement::=
     RETURN [ expression ] ;
```

1. The expression option must be used if the procedure is invoked as a function. If this option is used with a subroutine type of invocation the expression is evaluated but no value is returned.
2. Any number of return-statements may appear in a procedure-block.

Example:

```
A:  PROCEDURE;
    DECLARE (X,Y,Z W) WORD;
B:  PROCEDURE (P,X) WORD;
    DECLARE (P,X) WORD;
      Z+=1;
      IF P THEN RETURN; ELSE RETURN (X*X);
      END B;
      CALL B(0,X);
      W=Z;
      Y= B(1,X);
       .
       .
       .
      END A;
```

In this example procedure B is invoked once as a subroutine and once as a function. The first argument is a 0 to indicate that the invocation was in subroutine mode and the argument is 1 to indicate the invocation was in function mode. The procedure, B, contains both types of return-statements. The procedure tests the parameter P to determine the type of return to be executed. The "RETURN expression" form is executed if and only if the invocation was as a function. Otherwise the "RETURN" form is executed.

Appendix A
SYNTAX OF MPL

This appendix gives the syntax of MPL using the notation of section 1. The syntax is specified at two levels. Two levels of syntax are used in order to reduce the number of rules that would be required in a single level of syntax.

Level-one syntax defines MPL source text in terms of delimiters and non-delimiters.

The delimiters are defined as:

- blank
- comment
- special characters

The non-delimiters are defined as:

- identifiers
- constants

In the level-one syntax MPL source text is defined as a string of delimiters and non-delimiters. Furthermore, it is specified that between any two non-delimiters there must be one or more delimiters. Since blanks and comments are delimiters, this syntax specifies where blanks or comments may appear in an MPL program.

Level-2 syntax is the major part of the syntactical description of MPL. It defines an MPL program in terms of the programming character set and the following notation variables of the level-1 syntax:

- identifier
- decimal-number
- string
- bit-string

The level-2 syntax does not describe where blanks and comments may appear in a MPL program, the spaces shown in the rules are provided to aid readability.

The following notation is used to avoid ambiguity:

bl is used to represent the blank character

⊥ The vertical stroke is underlined whenever the language character is intended. When not underlined the vertical stroke represents the alternative operator of the notation.

extra-lingual-character

this notation variable is used to represent any characters available at the users input terminal that are not part of the MPL character set.

In the syntax presentation each rule is numbered. The notation variable being defined is shown on the numbered line. The definition is given on the following unnumbered line(s).

LEVEL 1 SYNTAX

1. source-text::=
   nondelimiter [tail-one] | delimiter [tail-one | tail-two]
2. tail-one::=
   delimiter [tail-one | tail-two]
3. tail-two::=
   nondelimiter [tail-one]
4. nondelimiter::=
   identifier | decimal-number | bit-string | string
5. identifier::=
   alphabetic-character [alphameric-character] ...
6. alphabetic-character::=
   A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|#|_
7. alphameric-character::=
   alphabetic-character | digit
8. digit::=
   0|1|2|3|4|5|6|7|8|9

9. decimal-number::=

   digit...

10. bit-string::=

    "{[(legal-size)] legal-digit...} ..."

11. legal-size::=

    1|2|3|4

12. legal-digit::=

    digit |A|B|C|D|E|F

13. string::=

    'string-character...'

14. string-character::=

    bl|=|+|-|*|/|(|)|,|''|"|;|:|&|⊥|¬|>|<|@ alphameric-character|
    extra-lingual-character

15. delimiter::=

    composite-operator | delimiting-character | bl | comment

16. composite-operator

    >= |¬= | <= | := | +=

17. delimiting-character::=

    =|+|-|*|/|(|)|,|;|: |¬|&|⊥|>|<|@

18. comment::=

    /* [comment-character-string]*/

19. comment-character-string::=

    {comment-character | /} [comment-character-string]|
    *... [comment-character [comment-character-string]]

20. comment-character::=

    bl|=|+|-|(|)|,|'|"|;|:|¬|&|⊥|>|<|@|
    alphameric-character|extra-lingual-character

LEVEL 2 SYNTAX

21. program::=

    external-procedure...

22. external-procedure::=

    entry-name: external-procedure-head

    [declare-statement]...[block-sentence] ... end-statement [EOF]

23. entry-name::=

    identifier

24. external-procedure-head::=

    MAIN PROCEDURE; | INTERRUPT PROCEDURE(parameter);|

    PROCEDURE [(parameter-list)] [simple-size] ;

25. parameter::=

    identifier

26. parameter-list::=

    identifier [,identifier]...

27. simple-size::=

    BYTE|WORD|DOUBLE

28. declare-statement::=

    DECLARE declaration-element [, declaration-element] ...;

29. declaration-element::=

    item-specification | identifier LITERALLY string

30. item-specification::=

    entry-name type-attribute|

    item-list size-attribute [area-attribute]

31. type-attribute::=

    {[EXTERNAL] PROCEDURE | (constant)MICRO} [simple-size]

32. Item-list::=

    identifier [(constant)] | (identifier[(constant)][,identifier[(constant)]]...)

33. size-attribute::=

    simple-size | POINTER TO simple-size | BIT (constant)

34. area-attribute::=

EXTERNAL | CONSTANT {string | ([+|-] constant [, [+|-]constant]...)}|

BASED {constant | identifier}

35. block-sentence::=

procedure | executable-unit

36. procedure::=

entry-name: procedure-head

[declare-statement]... [block-sentence]... end-statement

37. procedure-head::=

INTERRUPT PROCEDURE (parameter); |

PROCEDURE [(parameter-list)] [simple-size];

38. end-statement::=

[label-list] END [label | entry-name];

39. label-list::=

{label:} ...

40. label::=

identifier

41. executable-unit::=

unconditional-executable-unit | if-statement

42. unconditional-executable-unit::=

block|group|command

43. block::=

[label-list] BEGIN; [declare-statement]...

[block-sentence] ... end-statement

44. group::=

[label-list] group-heading [block-sentence]...end-statement

45. group-heading::=

REPEAT expression TIMES; | DO [do-specification] ;

46. do-specification::=

identifier=expression TO expression [BY expression]|

CASE expression | WHILE expression | FOREVER

47. command::=

[label-list] statement

48. statement::=

assignment-statement | call-statement

goto-statement | null-statement | return-statement

49. assignment-statement::=

storage-reference {= | := | +=} expression;

50. call-statement::=

CALL procedure-reference;

51. procedure-reference::=

entry-name [(expression [,expression] ...)]

52. goto-statement::=

{GOTO | GO TO} label;

53. null-statement::=

;

54. return-statement::=

RETURN [expression] ;

55. if-statement::=

if-clause executable-unit |

if-clause balanced-executable-unit ELSE executable-unit

56. if-clause::=

[label-list] IF expression THEN

57. balanced-executable-unit::=

unconditional-executable unit |

if-clause balanced-executable-unit ELSE balanced-executable-unit

58. expression::=

conditional-expression | simple-expression |

storage-reference:= expression

59. conditional-expression::=

IF expression THEN expression ELSE expression

60. simple-expression::=

    logical-term [{⊥ | XOR} logical-term]...

61. logical-term::=

    logical-factor [& logical-factor]...

62. logical-factor::=

    numeric-expression [comparison-operator numeric-expression]...

63. numeric-expression::=

    numeric-term [{+ | -} numeric-term]...

64. numeric-term::=

    numeric-factor [multiply-operator numeric-factor]...

65. numeric-factor::=

    numeric-primary | unary-operator numeric-factor

66. comparison-operator::=

    < | <= | = | = | >= | >

67. multiply-operator::=

    * | / | MOD | MULD | DIVD | shift-operator

68. unary-operator::=

    + | - | ‾

69. shift-operator

    SRA | SRL | SLL | SLC

70. numeric-primary::=

    constant | string | @ variable | procedure-reference |
    storage-reference | PRTNUM (entry-name) | (expression)

71. constant::=

    decimal-number | bit-string

72. variable::=

    identifier [(expression)]

73. storage-reference::=

reference [$ (expression:expression)]

74. reference::=

variable [@ [(expression)]]

SYNTAX CROSS REFERENCE

The list below is a cross reference of the notation variables and reserved key words that appear in both the level 1 and level 2 syntax. Each notation variable is proceeded with the rule number in which it is defined. Following each notation variable and reserved key word is the list of rule numbers where they are referenced.

6 alphabetic-character

5,7

7 alphanumeric-character

5,14,20

34 area-attribute

30

49 assignment-statement

48

57 balanced-executable-unit

55,57

BASED

34

BEGIN

43

BIT

33

10 bit-string

4,71

43 block
42

35 block-sentence
22,36,43,44

BY
46

BYTE
27

CALL
50

50 call-statement
48

CASE
46

47 Command
42

18 comment
15

20 comment-character
29

19 comment-character-string
18,19

66 comparison-operator
62

16 composite-operator
15

59 conditional-expression
58

CONSTANT
34

71 constant
31,32,33,34,70

9 decimal-number
4,71

29 declaration-element
28

DECLARE
28

28 declare-statement
22,36,43

29 delimiter
1,2

17 delimiting-character
15

8 digit
7,9

DIVD
67

DO
45

46 do-specification
45

DOUBLE
27

ELSE
55,57,59

END
38

EOF
22

IF

56,59

56 If-clause

55,57

55 if-statement

41

INTERRUPT

24,37

32 item-list

30

30 item-specification

29

40 label

38,39,52

39 label-list

38,43,44,47,56

LEAVE

12 legal-digit

10

11 legal-size

10

LITERALLY

29

62 logical-factor

61

61 logical-term

60

MAIN

24

MICRO

31

38 end-statement

22,36,43,44

23 entry-name

22,30,36,38,51,70

41 executable-unit

35,55

58 expression

45,46,49,51,54,56,58,59,70,72,73,74

extra-lingual-character

14,20

EXTERNAL

34

22 external-procedure

21

24 external-procedure-head

22

FOREVER

46

GO

52

GOTO

52

52 goto-statement

48

44 group

42

45 group-heading

44

5 identifier

4,23,25,26,29,32,34,40,46,72

MOD
67
MULD
67
67 multiply-operator
64
4 non-delimiter
1,3
53 null-statement
48
63 numeric-expression
62
65 numeric-factor
64,65
70 numeric-primary
65
64 numeric-term
63
25 parameter
24,37
26 parameter-list
24,37
POINTER
33
PROCEDURE
24,31,37
36 procedure
35
37 procedure-head
36
51 procedure-reference
50,70
21 program
PRTNUM
70
74 reference
73
REPEAT
45
RETURN
54
54 return-statement
48
69 shift-operator
67
60 simple-expression
58
27 simple-size
24,31,33,37
33 size-attribute
30
SLC
69
SLL
69
1 source-text
SRA
69
SRL
69

48 statement

47

73 storage-reference

49,70

13 string

4,29,34,70

14 string-character

13

2 tail-one

1,2

3 tail-two

1,2,3

THEN

56,59

TIMES

45

TO

33,46,52

31 type-attribute

30

68 unary-operator

65

42 unconditional-executable-unit

41,57

72 variable

70,74

WHILE

46

WORD

27

XOR

60

Appendix B
COMPILER TOGGLES

MPL provides some control over the compilation process through the use a mechanism called compiler toggles. Each compiler toggle is represented by a single character appearing in a comment immediately following a dollar sign within a comment. For example in the comment:

```
/* $C */
```

The character C is a toggle.

A toggle may have a value of true or false.

Each time a toggle appears its truth value is changed. The setting of the toggle causes the compiler to take a specified action. For example the toggle, C, causes the compiled operation codes to be listed. Therefore in the program segment:

```
/* $C        LINE 1 */
A = B;    /* LINE 2 */
C = B;    /* LINE 3 */
/* $C        LINE 4 */
E = F;    /* LINE 5 */
```

The code for lines 2 and 3 will be listed but the code for line 5 will not be listed.

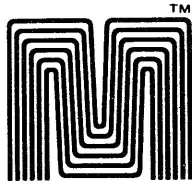
The table below gives the function of the toggles.

TOGGLE	FUNCTION	INITIAL STATE
A	Prints Symbol Table	False
C	Prints Operation Codes	False
D	Prints Symbol Table at Procedure End	False
E	Prints Allocation Map at Block End	False
F	Full Symbol Table Rather Than Current Block	False
H	Print Hexadecimal Load Records	False
L	Lists Source Program	True
M	Lists Without Numbering	False
U	Page Eject Immediately (This toggle resets after the page is ejected)	False
|	Sets right hand margin. The portion of the input record from this and all succeeding records starting from the column containing the vertical stroke will be ignored. The subsequent appearance of a vertical stroke in column that is recognized cause the right margin to be set at column 80	Column 80
O	Outputs loader text to a separate file	False

Appendix C
SAMPLE MPL PROGRAM

```
 /*****************************************************************/
 /*            PROCEDURE TO PRINT ALL PRIMES LESS THEN 1000        */
 /*****************************************************************/

PRIME:
  MAIN PROCEDURE;
    DECLARE
      PRINT    EXTERNAL PROCEDURE,
      P(200)   WORD,                        /* THE ARRAY OF PRIMES */
      TEST     WORD,                        /* VALUE BEING TESTED */
      (I,J)    WORD;                        /* SUBSCRIPTS */
    DO I=1 TO 3;
      CALL PRINT(I,P(I):=I);
    END;
    TEST=5;
    DO WHILE TEST<1000;
      J=3;
      DO WHILE TEST/P(J)>=P(J);
        IF TEST MOD P(J)=0 THEN GOTO PR01;
        J+=1;
      END;
      CALL PRINT(I,P(I):=TEST);
      I+=1;
PR01:
      TEST+=2;
    END;
  END PRIME;


 /*********************************************/
 /*    PRINT TWO DECIMAL NUMBERS              */
 /*********************************************/

PRINT:
  PROCEDURE(L,M);
    DECLARE
      ASCII(5) WORD,       /* THE ASCII FORM OF A NUMBER */
      (L,M)    WORD,       /* NUMBERS TO PRINT */
      I        WORD;       /* LOOP INDEX */


DECCONV: /* THIS ROUTINE CONVERTS BINARY VALUE TO DECIMAL */

      PROCEDURE(K);
        DECLARE K WORD;
        DO I=4 TO 0 BY -1;
          ASCII(I)=K MOD 10+'0';
          K=K/10;
        END;
        DO I=0 TO 3;
          IF ASCII(I)¬='0' THEN RETURN;
          ASCII(I)=' ';
        END;
      END DECCONV;


TYPE: /* THIS ROUTINE TYPES CHARACTERS ON THE TTY */

      PROCEDURE(N);
        DECLARE
          N      WORD,
          TTY(7) WORD BASED "E000";
        TTY(1)="22";                          /* START THE TTY */
        DO I=0 TO N-1;
          DO WHILE (TTY(0)&"20")=0; END;      /* WAIT FOR READY */
          TTY(2)=ASCII(I);                    /* SEND OUT A CHARACTER */
        END;
        TTY(1)=4;                             /* STOP THE TTY */
      END TYPE;


 /* NEXT STATEMENT IS START OF PROCEDURE 'PRINT' */

    ASCII(0)="0D";
    ASCII(1)="0A";
    CALL TYPE(2);
    CALL DECCONV(L);
    CALL TYPE(5);
    CALL DECCONV(M);
    CALL TYPE(5);
  END PRINT;
```

Microdata



Microdata Corporation
17481 Red Hill Avenue
Irvine, California 92705
(714) 540-6730 TWX: 910-595-1764