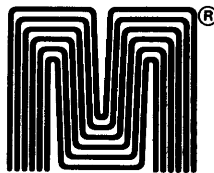
MPL
REFERENCE MANUAL

MPL REFERENCE MANUAL

SMPL-1

September 1975

32/S PROGRAMMING LANGUAGE



© 1976 Microdata Corporation
TM Trademark of Microdata Corporation
Printed in U.S.A.
98800 76 1022A

Microdata Corporation
17481 Red Hill Avenue, Irvine, California 92714
Post Office Box 19501, Irvine, California 92713
Telephone: 714/540-6730 · TWX: 910-595-1764

TABLE OF CONTENTS

Section		Page
1.0	INTRODUCTION	1
1.1	Summary of the Language	1
1.1.1	Block Structure	1
1.1.2	Data Description	2
1.1.3	Storage Allocation	2
1.1.4	Data Organization	2
1.1.5	Input/Output	2
1.1.6	Expressions	2
1.2	Syntax Notation	3
1.2.1	Notation Variables	3
1.2.2	Notation Constants	3
1.2.3	Vertical Stroke	4
1.2.4	Braces	4
1.2.5	Brackets	4
1.2.6	Ellipsis	5
1.2.7	The Definition Symbol	5
2.0	ELEMENTS OF THE LANGUAGE	6
2.1	Basic Language Structure	6
2.1.1	Character Set	6
2.1.2	Delimiters	8
2.1.3	Operators	8
2.1.4	Separators and Other Delimiters	10
2.1.5	Data Character Set	10
2.1.6	Identifiers	11
2.1.7	Keywords	11
2.1.8	Built-In Function Names	12
2.1.9	Use of Blanks	12
2.1.10	Comments	13
2.2	Basic Program Structure	13
2.2.1	Simple Statements	13
2.2.2	The IF Statement	14
2.2.3	Label List	14
2.2.4	Groups	14
2.2.5	Blocks	15
2.2.6	Programs	18
3.0	DATA ELEMENTS	19
3.1	Data Organization	19
3.1.1	Scalar Items	19
3.1.1.1	Constants	19

TABLE OF CONTENTS (con't)

Section		Page
3.1.1.2	Scalar Variables	19
3.1.2	Arrays	19
3.2	Referencing of Data	20
3.2.1	Simple Reference	20
3.2.2	Subscripted Reference	20
3.2.3	Indirect Reference	21
3.2.4	Subfield Reference	22
3.3	Data Types	23
3.3.1	Decimal Numbers	23
3.3.2	Bit Strings	24
3.3.3	Strings	25
4.0	DATA MANIPULATION	26
4.1	Expressions	26
4.1.1	Simple Expressions	26
4.1.1.1	Arithmetic Operations	27
4.1.1.2	Arithmetic Comparison Operations	28
4.1.1.3	Logical Comparison Operations	28
4.1.1.4	Logical Operations	29
4.1.1.5	Shift Operations	30
4.1.1.6	The Assignment Operation	30
4.1.2	Conditional Expressions	31
4.2	Evaluation of Expressions	32
5.0	DATA DESCRIPTION	34
5.1	Attributes	34
5.2	Declarations	34
5.2.1	The Declare Statement	35
5.2.2	Factoring of Attributes	36
5.2.3	Multiple Declarations	36
5.2.4	Procedure Labels	37
5.2.5	Scope of Declarations	37
5.2.5.1	Scope of External Names	38
5.2.5.2	Basic Rule For The Use Of Identifiers	40
5.2.6	Declare Elements	41
5.2.6.1	Literally Declare Element	41
5.2.6.2	Procedure Declare Element	42
5.2.6.3	Data Item Declare Element	44
5.2.6.4	Size Attribute	44
5.2.6.5	Area Attribute	45
6.0	PROCEDURES, FUNCTIONS, AND SUBROUTINES	48
6.1	Parameters	48
6.2	Procedure References	48
6.2.1	Function Reference	49
6.2.2	Built-In Functions	50

TABLE OF CONTENTS (con't)

Section		Page
6.2.3	Subroutine References	52
6.3	Procedure Reference Examples	53
6.4	Arguments in a Procedure Reference	54
7.0	DYNAMIC PROGRAM STRUCTURE	57
7.1	Program Control	57
7.2	Activation and Termaination of Blocks	57
7.3	Allocation of Data	58
7.3.1	Definitions and Rules	58
7.3.2	Storage Classes	58
7.3.2.1	The Static Storage Class	59
7.3.2.2	The Automatic Class	59
8.0	STATEMENTS	61
8.1	Relationship of Statements	61
8.1.1	Assignment Statements	61
8.1.2	Control Statements	61
8.1.3	Data Declaration Statement	61
8.1.4	Block Statements	61
8.2	Sequence of Control	62
8.3	Alphabetic List of Statements	63
8.3.1	The Assignment Statement	63
8.3.2	The Begin Statement	65
8.3.3	The Call Statement	65
8.3.4	The Declare Statement	66
8.3.5	The Do Statement	66
8.3.6	The End Statement	70
8.3.7	The EOF Statement	71
8.3.8	The GOTO Statement	71
8.3.9	The If Statement	73
8.3.10	The Null Statement	75
8.3.11	The Procedure Statement	75
8.3.12	The Repeat Statement	77
8.3.13	The Return Statement	78
APPENDIX A	SYNTAX OF MPL	80
APPENDIX B	SYNTAX CROSS REFERENCE	87
APPENDIX C	COMPILER TOGGLES	91
APPENDIX D	32/S STANDARD SYMBOL-CODE CORRESPONDENCES	93
APPENDIX E	SAMPLE MPL PROGRAM	110

1.0 INTRODUCTION

1.1 SUMMARY OF THE LANGUAGE

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

The remainder of this section briefly summarizes the MPL language and describes the syntax notation used in the remainder of this manual. Sections 2 through 8 describe the full details of the MPL language.

1.1.1 Block Structure

MPL statements are organized into sections called blocks. 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 nested. A block is said to be nested when all 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 the 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 before a program begins execution and remains allocated during the entire program execution. Dynamic storage is allocated on block entry and freed on block exit.

1.1.4 Data Organization

The MPL 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 memory addresses of the various registers of an I/O 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 I/O 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, the expression:

```
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 this 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 an underscore and no spaces may exist in a notation variable. Some examples of notation variables are:

```
item_list
goto_statement
```

1.2.2 Notation Constants

Notation constants specify the literal occurrence of the characters in the language element being described and are written in upper case letters. 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 notation constants and notation variables into a syntactical unit. When braces are used for this grouping, the presence of the syntactical unit is required. For example:

indentifier {BYTE | WORD}

indicates that the notation variable "identifier" must be followed by the occurrence of either the notation constant BYTE or the notation constant 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 syntactical units. When brackets are used, the syntactical unit is optional. For example:

identifier DOUBLE [BASED]

indicates that the notation variable "identifier" is followed by the occurrence of the notation constant DOUBLE and optionally followed by the occurrence of the notation constant BASED.

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 indicate that the unit may be repeated. If the syntactical 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"s and/or "digit"s, or by none at all.

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

2.0 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 form format and are independent of any physical record boundaries.

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 when used in identifiers (and keywords). 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
```

The names of the special character symbols and their graphic representations follow in table 1.

<table>
<tr><th>NAME</th><th>GRAPHIC</th></tr>
<tr><td>Blank</td><td></td></tr>
<tr><td>Equal or assignment symbol</td><td>=</td></tr>
<tr><td>Plus</td><td>+</td></tr>
<tr><td>Minus</td><td>-</td></tr>
<tr><td>Asterisk or multiply symbol</td><td>*</td></tr>
<tr><td>Slash or divide symbol</td><td>/</td></tr>
<tr><td>Left parenthesis</td><td>(</td></tr>
<tr><td>Right parenthesis</td><td>)</td></tr>
<tr><td>Comma</td><td>,</td></tr>
<tr><td>Single Quote</td><td>'</td></tr>
<tr><td>Double Quote</td><td>"</td></tr>
<tr><td>Currency symbol</td><td>$</td></tr>
<tr><td>Commercial-at sign</td><td>@</td></tr>
<tr><td>Semicolon</td><td>;</td></tr>
<tr><td>Colon</td><td>:</td></tr>
<tr><td>Not symbol</td><td>¬</td></tr>
<tr><td>And symbol</td><td>&</td></tr>
<tr><td>Or symbol</td><td>| or !</td></tr>
<tr><td>Greater-than symbol</td><td>></td></tr>
<tr><td>Less-than symbol</td><td><</td></tr>
<tr><td>Question mark</td><td>?</td></tr>
</table>

Table 1: Special Characters

2.1.2 Delimiters

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

2.1.3 Operators

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

Table 2: Expression Operators

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

In addition the following identifiers are keywords which 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
SRL	denoting shift right logical
SLC	denoting shift left circular
SLL	denoting shift left logical
LLT	denoting logical less-than
LLE	denoting logical less-than-or-equal-to
LEQ	denoting logical equal-to
LNE	denoting logical not-equal-to
LGE	denoting logical greater-than-or-equal-to
LGT	denoting logical greater-than

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 for enclosing subscripts, subexpressions, 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 and DO 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
/*	Used to specify the beginning of a comment
*/	Used to specify the ending of a comment

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 data characters are fully described in Appendix D.

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 an identifier is:

```
identifier ::=

     alphabetic_character [alphameric_character]...
```

Identifiers in MPL are used for the following:

Scalar variable names
Array variable names
Statement labels
Procedure names
Keywords
Literally names

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. All keywords are reserved and may not be used except in their intended structural use. The following is a list of the keywords:

```
BASED       FOREVER      MULD
BEGIN       GO           POINTER
BIT         GOTO         PROC
BY          IF           PROCEDURE
BYTE        INTERRUPT    PRTNUM
CALL        INIT         REPEAT
CASE        INITIAL      RETURN
CONSTANT    LEQ          SLC
DCL         LGE          SLL
DECLARE     LGT          SRA
DIVD        LIT          SRL
DO          LITERALLY    STATIC
DOUBLE      LLE          THEN
ELSE        LLT          TIMES
END         LNE          TO
EOF         MAIN         WHILE
EXT         MICRO        WORD
EXTERNAL    MOD          XOR
```

2.1.8 Built-In Function Names

Built-in function names are identifiers that name functions accessible to the programmer. Built-in function names are not reserved and may be used as variabe names, statement labels, or procedure names. 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). Table 10 (in Section 6) lists the built-in function names.

2.1.9 Use of Blanks

Blanks are defined to be a string of blank characters and/or comments (see 2.1.10). Blanks are not allowed within identifiers, decimal numbers, or multiple character operators:

```
A B C represent the three identifiers A, B, and C.
1 2 3 represent the three decimal numbers 1, 2, and 3.
> =   represent the two separate operators > and =.
```

Blanks within a character string are recognized as legitimate characters:

```
'A B C' represent the five characters A B C.
'/*B*/' represent the five characters /*B*/.
```

Blanks are required after an identifier and before an otherwise adjacent identifier or decimal number:

```
ABC represent the single identifier ABC.
A23 represent the single identifier A23.
```

Otherwise blanks are optional:

```
IF'X'->ALPHA THEN B="5";
```

is equivalent to:

```
IF 'X' /**/ -> ALPHA/**/THEN B ="(/**/4)5";

CALL SUB        is not equivalent to CALLSUB.
A TO 10 BY 3    is not equivalent ATO10BY3.
A=5             is equivalent to A = 5.
```

2.1.10 Comments

General format:

```
/*comment_string*/
```

Comments are used for documentation of the source program and have no effect on execution. However, the characters $, &, and % appearing in a comment are used to control the compilation process. See Appendix C 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 a semicolon character (;) or the character pair */ in that order. For example:

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

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. For 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. For 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 the names of the identifiers are known and for which storage is allocated to the identifiers. Blocks are also used to control the sequence of execution.

There are two kinds of blocks: begin and procedure. 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.

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.

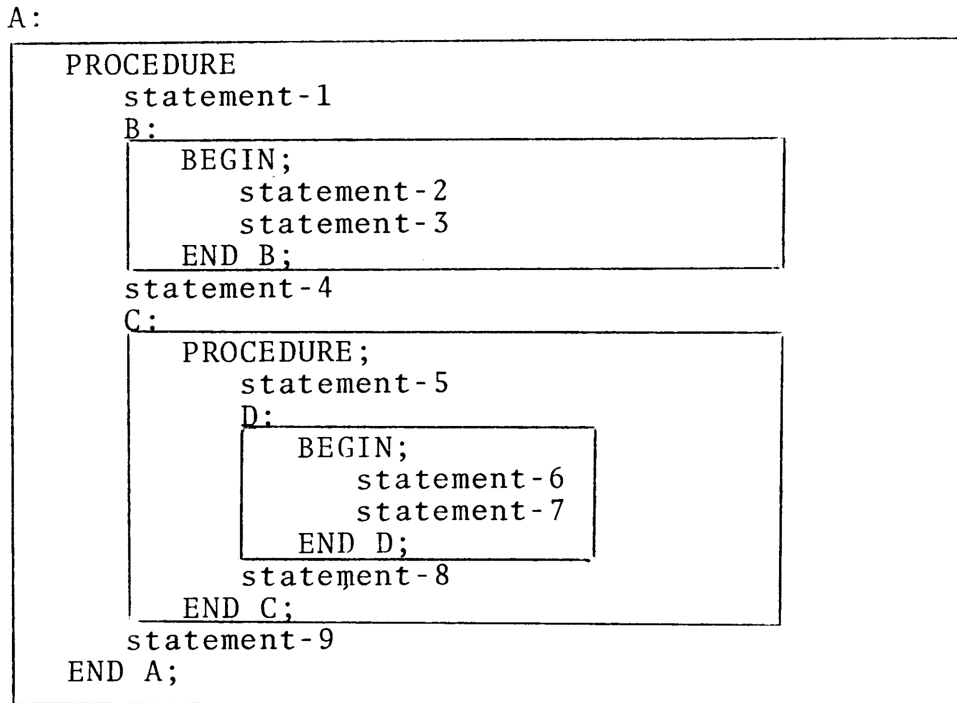
A procedure nested within a block is called an internal procedure.

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 block B, but is not contained within any other block nested inside of B, is said to be internal to block B.

The notation internal to is vital to the understanding of the definition of scope and to the understanding of allocation of storage. For example:



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

As shown by the boxes enclosing the statements, 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 main procedure and, optionally, additional external procedures. Thus a program is a set of procedure blocks each of which may have other procedures contained in them.

3.0 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 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. For 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 referencing a data item. 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 2.1.6). 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 parentheses 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 Section 5.2.6.5).

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, subcripted, 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.

In a two expression field select, the first expression specifies the number of bits to be referenced and 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. For example:

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

The one expression field select is primarily used to reference a single bit at the specified bit position. Example:

X$(0)	references the least significant bit of the WORD X
A$(15)	references the most significant bit of the WORD A

However, the one expression field select may be used to reference any field in a word. Bits 7 through 4 of the field select expression specify one less than the number of bits in the referenced field and bits 3 through 0 of the field select expression specify the least significant bit position of the referenced field. Example:

A$("3C")	references a field consisting of the four most significant bits of the WORD A
X$(64)	references a field consisting of the five least significant bits of the WORD X

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 whose precision depends upon the arithmetic value of the decimal number. Decimal numbers in the range of 0 through 32,767 have a precision of 16 bits. Decimal numbers in the range of 32,768 through 2,147,483,647 have a precision of 32 bits. Leading zeros have no effect on the arithmetic value

or precision of a decimal number. Examples:

```
00012       precision used is 16 bits
  200       precision used is 16 bits
15972       precision used is 16 bits
87962       precision used is 32 bits
```

3.3.2 Bit Strings

General format:

```
bit_string ::=
   "[ [(legal_size)] [legal_digit] ]..."

legal_size ::=
   1 | 2 | 3 | 4
```

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

Table 6: Legal Digits for Bit Strings

Legal 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 the legal size option is omitted, a default of 4 is used.

A bit is treated as an integer whose precision depends upon the arithmetic value and the total number of bits specified in the bit string (where the total number of bits is equal to the sum of all of legal digits times their associated legal size). Bit strings whose total number of bits specified is in the range of 0 through 16 have a precision of 16 bits. Bit strings whose total number of bits specified is in the range of 17 through 32 have a precision of 32 bits. Leading zeros are counted as legal digits and hence are significant in determining the precision of bit strings.

Examples:

bit string	binary value	precision
"32"	00110010	16 bits
"(3)32"	00011010	16 bits
"FA(1)1010(2)21"	1111101010101001	16 bits
"00001"	00000000000000000001	32 bits
"(1)12"	(invalid)	

3.3.3 Strings

General format:

```
    string ::=
        ' [non_quote_char]... ' [string]
```

The character symbols that may appear include any characters in Appendix C. However, when a single quote character is required within the string it is represented by two consecutive single quote characters. Example:

```
'THIS IS A ''characters_string'''
```

represents the characters:

```
THIS IS A 'character_string'
```

The length of a string may be from 0 to 255 characters.

A string which is used as a constant is treated as an integer whose precision depends upon the length of the string. The length of a string used as a constant may be from 0 to 4 characters. Strings used as a constant whose length is 0 through 2 have a precision of 16 bits. Strings used as a constant whose length is 3 or 4 have a precision of 32 bits. Strings used as a constant are right justified and left zero filled.

Examples:

string	precision
''	16 bits
'A'	16 bits
'AB'	16 bits
'ABC'	32 bits
'ABCD'	32 bits
'ABCDE'	(invalid)

4.0 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 (16 bits) or DOUBLE (32 bits). The precision used is determined by the precision of the operands. If the operands are WORD or BYTE, or BIT(n) 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 parentheses 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. BIT(n) operands are treated as 1,2, or 4 positive values and are converted to WORD (16 bit) precision by extending the value with zeros to the left. BYTE (8 bit) operands are treated as 8 bit positive values and are converted to a WORD (16 bit) value by extending the value with zeros to the left. WORD operands are converted to DOUBLE values 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 of the result is the sign of the dividend.
MULD	indicates double precision multiplication. The result of this operation is precision DOUBLE even though the operands are WORD, BYTE, or BIT(n).
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, BYTE, BIT(n) 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 sign of the operand is changed.

4.1.1.2 Arithmetic Comparison Operations

The arithmetic comparison operations are all infix operations. Arithmetic comparison operations treat the operands as signed numbers in two's complement binary notation. The result of the operation is the value one if the relation is true. The result of the operation is the value zero if the relation is false.

The following are the operators used to indicate arithmetic comparison operations:

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

The logical comparison operations are all infix operations. Logical comparison operations treat the operands as unsigned 16 bit numbers in binary notation. The result of the operation is the value one if the relation is true. The result of the operation is the value zero if the relation is false.

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

LGT	indicates logical greater-than
LGE	indicates logical greater-than-or-equal-to
LEQ	indicates logical equal-to
LNE	indicates logical not equal-to
LLE	indicates logical less-than-or-equal-to
LLT	indicates logical less-than

Examples:

```
5  LGT  3    has the value 1
2  LGT  2    has the value 0
3  LGT  3    has the value 1
-1 LGT  4    has the value 1
-2 LGT -1    has the value 0
```

4.1.1.4 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	¬A	¬B	A & B	A | B	A XOR 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:

```
| or !        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"
```

4.1.1.5 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 "01F4"
"8000" SLC 2    has the value "0002"
```

4.1.1.6 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:

```
:=
```

The 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.

Example:

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 of 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 or 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. For example:

```
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:                   >,>=,=,¬=,<=,<,¬>,¬<,            |
                              LLT,LLE,LEQ,LNE,LGE,LGT          |
and:                          &                                |
or:                           |  , XOR                         v
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 the parantheses 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.

5.0 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 are:

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

INITIAL(5) - This attribute defines an identifier to be the name of a data item with the initialized 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 5.2.5).

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 declare_element [, declare_element]... ;

declare_element ::=
    identifier LITERALLY string |
    entry_name type_attribute |
    item_list size_attribute [area_attribute]

type_attribute ::=
    [EXTERNAL] procedure [simple_size] |
    (constant) MICRO [simple_size]

item_list ::=
    identifier [(constant)] |
    (identifier [(constant)] [, identifier [(constant)]]... )

size_attribute ::=
    simple_size |
    POINTER TO simple_size |
    BIT (constant)

area_attribute ::=
    EXTERNAL |
    STATIC |
    CONSTANT string |
    CONSTANT ([+ | -)] constant [, [+ | -] constant]... ) |
    INITIAL string |
    INITIAL ([+ | -] constant [, [+ | -] constant]... ) |
    BASED constant |
    BASED identifier

simple_size ::=
    BYTE | WORD | DOUBLE
```

General Rules:

1. A label is not allowed on a declare statement.
2. The declaration statements within any block must follow immediately after the block heading statement and before any other statements in the block.
3. Any number of identifiers may be declared in a single declare statement and declarations must be separated by commas.
4. Attributes must follow the items to which they refer.
5. All of the attributes of a given identifier must be declared in a single declare statement.
6. Attributes of an EXTERNAL name declared in separate blocks and compilations must be identical.

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 size attribute and area attribute, then the attributes may be factored to eliminate repeated specification of the same attribute for many identifiers. The factoring is accomplished by enclosing the data items in parentheses and following them with the attributes that apply to all the data items. The data 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 (implicit or explicit) 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 entry name of a procedure statement. The multiple declarations are:

1. An identifier being declared more than once internal to the same block.
2. An identifier appearing as a label more than once internal to the same block.
3. An identifier being declared and appearing as a label internal to the same block.
4. Declaring a parameter with an area attribute.

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 6
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 line 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 appearances 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, which is 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 Declare Elements

The declare statement is made up of a list of declare elements. A declare element is the declaration of a LITERALLY item, a procedure, or a data item.

5.2.6.1 Literally Declare Element

The LITERALLY declare element is used to name any arbitrary string of characters.

General form:

```
declare_element ::=
    identifier LITERALLY string
```

The normal scope rules apply to the identifier declared with the LITERALLY declaration. The occurrence of the identifier in the subsequent text of a block will result in the substitution of the string for the identifier. A use for the LITERALLY declaration allows the symbolic representation of constants and the ease in changing their values. For example the program segment:

```
A:  PROCEDURE;
    DECLARE INPUT PROCEDURE WORD;
    DECLARE KONSTANT LITERALLY '1';
    DECLARE SYMBOL LITERALLY '2';
    DECLARE OPERATOR LITERALLY '3';
    IF INPUT = KONSTANT THEN
    DO;
        .
        .
        .
    END;
    IF INPUT = SYMBOL THEN
    DO;
        .
        .
        .
    END;
    IF INPUT = OPERATOR THEN
    DO;
        .
        .
        .
    END;
        .
        .
```

is equivalent to the program segment:

```
A:  PROCEDURE;
    DECLARE INPUT PROCEDURE WORD;
    IF INPUT = 1 THEN
    DO;
         :
    END;
    IF INPUT = 2 THEN
    DO;
         :
    END;
    IF INPUT = 3 THEN
    DO;
         :
    END;
         :
```

5.2.6.2 Procedure Declare Element

The procedure declare element is used to declare identifiers to be procedure names.

General form:

```
declare_element ::=
    entry_name type_attribute
type_attribute ::=
    [EXTERNAL] PROCEDURE [BYTE | WORD | DOUBLE]|
    (constant) MICRO [BYTE | WORD | DOUBLE]
```

Identifiers declared to be procedure names are used when:

1. 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.
2. 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.
3. A reference is made to a procedure which has been microcoded. The 'constant' represents the address in the control storage of the 32/S where the microcoded process begins.

The size specification of BYTE, WORD, or DOUBLE is required with the procedure declaration when the procedure is to be referenced as a function. In this case the size specifies the precision of the value returned by the function. Example:

```
A:  PROCEDURE;
    DECLARE B PROCEDURE:
         :
    CALL B;
         :
    B:  PROCEDURE;
    END B;
    CALL B;
END A;
```

In this example the declaration "B PROCEDURE" is required because the first "CALL B;" statement occurs prior to the occurrence of the procedure block B. The declaration "B PROCEDURE" is not required (however, it is allowed) for the second "CALL B;" statement because it occurs after the procedure block B. For example:

```
A:  PROCEDURE;
    DECLARE B EXTERNAL PROCEDURE;
         :
    CALL B;
END A;
```

In this example the declaration "B EXTERNAL PROCEDURE" is required because the external procedure B is referenced from block A which is not within the scope of block B. Example:

```
A:  PROCEDURE;
    DECLARE B("3F0") MICRO WORD;
         :
    VALUE = B(I,J);
         :
END A;
```

In this example the declaration "B("3F0") MICRO WORD" declares the identifier B to be a microcoded procedure.

The WORD option specifies that the procedure will leave a 16-bit result on the top of the 32/S stack. The assignment statement invokes the microcoded procedure. The value of the variables I and J are placed on the 32/S stack before the procedure is invoked.

5.2.6.3 Data Item Declare Element

The data item declare element is used to declare identifiers as data items.

General form:

```
declare_element ::=
    item_list size_attribute [area_attribute]
```

An item list can be a single identifier or a list of identifiers. When the item list is a list of identifiers, all of the identifiers in the list are declared to have the same size and area attributes.

The item list can specify scalar and/or array identifiers. An array identifier is specified with a constant which is the upper array bound. The upper array 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 is one greater than the value of the upper array bound. Example:

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

In the example, A is a WORD array of 11 (16-bit) elements (upper bound = 10). B is a BYTE array of 6 (8-bit) elements (upper bound = 5).

5.2.6.4 Size Attribute

The size attribute is required for the declaration of all data item identifiers and specifies the precision of the data item.

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 precision specified 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 this 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 data 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 assigns the value of the 16 bit data item D to the 16 bit data item C. The assignment statement B = D@, assumes that the value of D is the location of an 8 bit data item. That 8 bit data item is assigned to the 8 bit data item, B.

5.2.6.5 Area Attribute

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 or that the normal storage area is initialized to a specific value.

General format:

```
area_attribute ::=
    EXTERNAL |
    STATIC |
    CONSTANT string |
    CONSTANT ([+ | -] constant [, [+ | -] constant]... ) |
    INITIAL string |
    INITIAL ([+ | -] constant [, [+ | -] constant]... ) |
    BASED constant |
    BASED identifier
```

General rules:

1. The EXTERNAL area attribute declares an identifier to be identical to the data item of the same name declared EXTERNAL in another block. (See 5.2.5).

2. The STATIC area attibute declares that the space for a data item is to be allocated only once for all invocations of a block.

3. The CONSTANT area attribute declares values for the associated data items. The number of data items and the number of initial values must be the same. A data item declared with the CONSTANT area attribute cannot be used as the object of any type of assignment.

   The string option may only be used with the BYTE size attribute. The data item is assumed to be an array variable whose dimension is specified by the MPL compiler based upon the number of characters in the string. If the data item is not specified as an array variable, then the MPL compiler will assign the dimension upper bound to the number of characters in the string. If the data item is specified as an array variable then the dimension must be specified to be equal to the number of characters in the string. The first byte of the array is set to a value by the MPL compiler which is equal to the number of characters in the string.

4. The INITIAL area attribute declares initial values for the associated data items. The number of data items must be equal to or greater than the number of initial values.

   The string option may only be used with the BYTE size attribute. The data item is assumed to be an array variable whose dimension is specified by the MPL compiler based upon the number of characters in the string. If the data item is not specified as an array variable, then the MPL compiler will assign the dimension upper bound as the number of characters in the string. If the data item is specified as an array variable, then the dimension must be specified to be equal to or greater than the number of characters in the string. The first byte of the array is set to a value by the MPL compiler which is equal to the number of characters in the string.

5. The BASED area attribute is used to assign absolute storage locations to identifiers. The value of the constant or the identifier base is multiplied by four to generate the base for an absolute address whenever the based variable is referenced.

The BASED area attribute can only be used in programs that operate in executive mode.

6. The relationship between identifiers declared in an item list and constants or strings specified in a value list is that the space for the item list is allocated first. The value list is then allocated into the same space starting with the first item in the item list and corresponding to the remainder of the item list on a one-to-one basis.

Example 1.

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

This example declares the BYTE array TTY to be used 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 four 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. 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 four. The result is adjusted by the subscript, 1, to form the absolute address for the reference.

Example 3.

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

This example generates an array of bytes in the program space. TEXT(0) contains the value 13 (the number of bytes in the string). TEXT(1) contains the value "45" (the ASCII representation of the letter E). The remaining elements of the array contain values that represent the remaining characters of the string.

Example 4.

```
DECLARE (ONE, TWO, THREE) WORD INITIAL (1, 2, 3);
```

This example initially assigns to the identifiers ONE, TWO, and THREE the 16 bit values 1, 2, and 3 respectively.

6.0 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 which are 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 to the identifier. For 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 specification of a dimension for the parameter, Y, 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 statement 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. Neither the MPL compiler nor the 32/S instructions validate the number, correspondence, or size of arguments and parameters.

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 in Section 6.4.

There are two distinct ways by which a procedure may be 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 reference 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 subroutine. However, it is not mandatory that this be the case, and neither the MPL compiler nor the 32/S instructions check this.

6.2.1 Function Reference

When a function reference appears in an expression, the referenced procedure is invoked. The procedure is then executed using the arguments (if any) that were specified in the function reference. The result of executing this function procedure requires that a value 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 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 SUPV instruction.
OVERFLOW	none, WORD, or DBLE	WORD	Returns the state of the arithmetic overflow indicator. A value of one(1) indicates on and a value of zero(0) indicates off.
RESUME	WORD	null	Resume the process whose stack base is the argument. Can be executed only in executive mode.
HIGH	DBLE	WORD	Returns the most significant part of the argument.
LOW	DBLE	WORD	Returns the least significant part of the argument.
ABS	WORD or DBLE	WORD or DBLE	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, or DBLE	WORD	Returns the status of the carry indicator. A value of one(1) indicates on and a value of zero (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.

FUNCTION NAME	ARGUMENT SIZE	RETURN SIZE	DESCRIPTION
PRTNUM	procedure-name	WORD	For internal procedures, returns the PB relative address of the procedure name. For external procedures, returns a word consisting of the PLIBN in bits 15 through 8 and the PRTN in bits 7 through 0.
DBLE	¬DBLE	DBLE	Makes a double precision value from a non-double precision argument.
DBLE	¬DBLE,¬DBLE	DBLE	Makes a double precision value from two non-double precision arguments.
SSR	none	null	Stuffs stack registers.
ENVIRONMENT	WORD	WORD	A 'POINTER TO' WORD array pointing to the mark in the stack for the current environment.

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 which specifies that control is to be transferred to a point outside the scope of the called procedure. In this case control passes to the point specified by the goto statement.

3. Control reaches the end statement associated with the called procedure. In this case 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 it may not be practical to return a value through a shared variable. Another 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. 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 array variable name. An array name used as an argument is equivalent to a pointer to element zero of the array.

An argument in a procedure reference cannot be an identifier which has been declared with a CONSTANT or BASED area attribute.

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 is 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.

When a procedure in invoked by a procedure reference and each argument is associated with its corresponding parameter, the arguments are said to be passed to the invoked procedure. Scalar arguments in MPL are passed by value and array arguments are passed by reference.

Passing by value is accomplished in the following way. The called procedure has a location of appropriate precision allocated for each scalar parameter in the parameter list. When a procedure reference is encountered, each scalar argument is evaluated. The result of the evaluation is, in effect, stored in the location reserved for the corresponding scalar parameter. This value is used in the same way as any scalar variable declared in the called procedure. In other words, a scalar 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 effect the value of a scalar 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 effect on the value of X. Therefore, when the statement S3 is executed, the value of the variable X is still 5.

Passing by reference is accomplished in the following way. The called procedure has a location of WORD precision allocated for each array parameter in the parameter list. When a procedure reference is encountered, the address of each array argument is determined. The address is, in effect, stored in the location reserved for the corresponding array parameter. This value is used as an indirect address in the called procedure. In other words, an array parameter is a variable local to the called procedure which, upon entry, is initialized to the value corresponding to the address of the base of the array argument.

7.0 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 sequence that determines the order of execution of the statements. For a discussion of the sequence of execution see Section 8.2.

7.2 ACTIVATION AND TERMINATION OF BLOCKS

A begin block is 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 a return statement internal to the block or the end statement of the block.

2. A procedure block is terminated when control passes through a return statement internal to the block 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 goto statement may termanate more than one block (see Section 8.3).

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 subsequently 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 three storage classes: static, automatic, and based.

7.3.2.1 The Static Storage Class

The storage for variables declared EXTERNAL or STATIC is in the static class. Storage for these variables is allocated before the program begins execution and is never released during execution.

7.3.2.2 The Automatic Storage Class

Variables declared in any block that are not declared EXTERNAL, STATIC, BASED, or CONSTANT are in the automatic storage class. Whenever a block 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 Section 5.2.5).

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 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 become 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 be active at the same time, the storage for the array X and Y will not be allocated at the same time. Thus, only 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;
```

7.3.2.2 The Based Storage Class

The storage for variables declared BASED is located in absolute memory and not allocated by the MPL compiler.

8.0 STATEMENTS

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

8.1 RELATIONSHIP OF STATEMENTS

Statements may be classed into the following four groups:

1. Assignment statement
2. Control statement
3. Data declaration statements
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, 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 internal 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 assigns the value to a variable.

General format:

```
assignment_statement ::=
    storage_reference assignment_operator expression ;

assignment_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 assignment operator is then evaluated.

2. If the assignment 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 assignment 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 WORD precision and may not contain a field reference option.

4. If an indirect storage reference is used, the variable pointed to by the storage reference must be currently allocated.

5. If the storage reference is precision BYTE or BIT(n), then the expression is truncated on the left before the assignment is made. If the storage reference is precision WORD, BYTE, or BIT(n) and the expression is precision DOUBLE, then the expression is truncated on the left before the assignment is made. If the expression is precision WORD and the storage reference is 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 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 procedure_reference ;

procedure_reference ::=
    entry_name [( procedure_argument [, procedure_argument]... )]

procedure_argument ::=
    expression |
    array_name

array_name ::=
    identifier
```

Syntax rules:

1. The entry name represents the label on the procedure to be invoked.

2. Each argument may be an expression or an array name.

3. An array name used as an argument is equivalent to a pointer to element zero of the array. For example if X is an array name then the following two statements are equivalent:

```
CALL SUB(X);
CALL SUB(@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 declaration_element [, declaration_element]... ;
```

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 Section 2.2.4) 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 do group is delimited by an end statement. There are five forms of the do statement.

General format 1:

```
do_statement ::=
    DO ;
```

General rules 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.

Example of format 1:

```
:
IF A = B THEN
DO;
    :
END;
:
```

All of the statements between the DO and the END are executed if A = B.

General format 2:

```
do_statement ::=
    DO WHILE expression ;
```

General rules 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
```

General format 3:

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

General rules 3:

1. The "variable" must be a simple variable. Indirect references or array variables may not be used.

2. If the "BY expression-3" is omitted, expression-3 is assumed to have a value of one (1).

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;
```

General format 4:

```
do_statment ::=
    DO CASE expression ;
```

General rules 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 LABEL;
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;
    T1 = expression;
    IF T1 = 0 THEN
        DO;
            group-0
            GOTO NEXT;
        END;
    IF T1 = 1 THEN
        DO;
            group-1
            GOTO NEXT;
        END;
        :
        :
    IF T1 = n THEN
        DO;
            group-n
            GOTO NEXT;
        END;
END LABEL;
NEXT:  statement
```

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. However, the error may not be detected. The groups: group-0, group-1, etc., may be do groups, repeat groups, begin blocks, or they may be simple statements.

General format 5:

```
do_statement ::=
    DO FOREVER ;
```

General rules 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
              :
          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.

8.3.6 The End Statement

Function:

The end statement terminates blocks and groups.

General format:

```
end_statement ::=
    [label_list] END [label | entry_name] ;
```

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 before or after an external procedure.

General format:

```
eof_statement ::=
    EOF
```

General rule:

The eof statement is included in this MPL compiler in order to be compatable with the 360/OS based MPL cross compiler. The eof statement has no effect if the source program is properly constructed (i.e., one end statement for each block heading statement).

8.3.8 The GOTO statement

Function:

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

General format:

```
goto_statement ::=
    go_to label ;

go_to ::=
    GOTO |
    GO [TO]
```

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;
```

In the second example, the GOTO C statement 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

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

General format:

```
if_statement ::=
    if_clause executable_unit |
    if_clause balanced_executable_unit ELSE executable_unit

if_clause ::=
    [label_list] IF expression THEN
```

The executable unit and the balanced executable unit are 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. The if statement itself is not terminated by a semicolon. Instead the semicolon that terminates the executable unit or the balanced executable unit serves to terminate the if statement. Both the balanced executable unit and the executable unit may contain labels.

General rules:

1. The expression following the 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 true. If the result of the expression evaluation is an even number (least significant bit is 0) the expression is said to be false.

2. If the expression is true, the first executable unit is executed and then control passes to the next statement (the second executable unit is skipped).

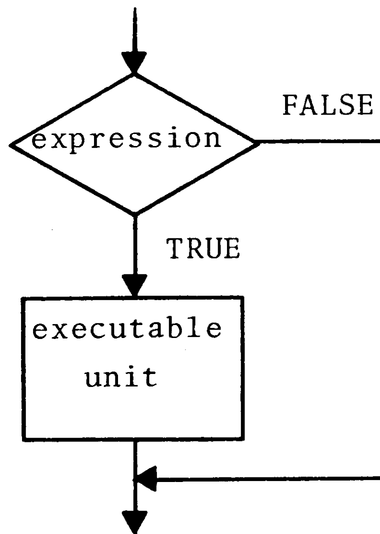
3. If the expression is false, the first executable unit is skipped and control passes to the second executable unit. If a second executable unit is not given, control passes to the next statement

4. If statements may be nested. That is, either the first or second executable may themselves be if statements. When if statements are nested the "ELSE second executable unit" portion of a statement is always associated with the innermost unmatched "THEN first executable unit". For this reason a null statement (see Section 8.3.10) 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 second executable unit" option.

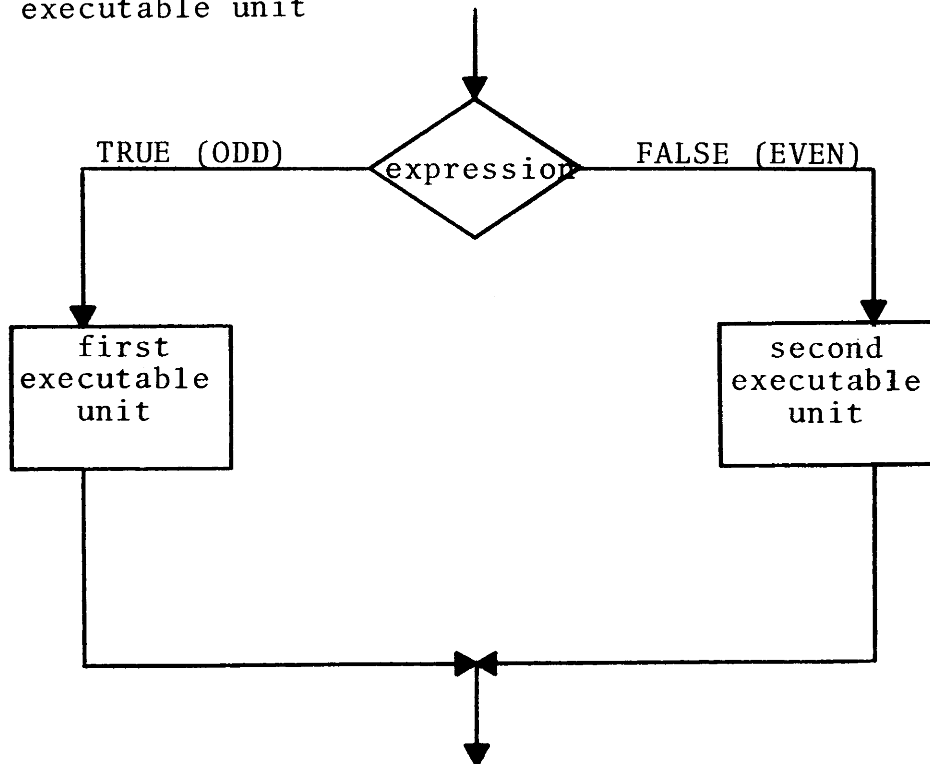
The first flowchart illustrates the case of:

IF expression THEN executable unit



The second flowchart illustrates the case of:

IF expression THEN first executable unit ELSE second executable unit



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 the ELSE is a null statement.

8.3.11 The Procedure Statement

The procedure statement has the following functions:

1. Heads a procedure block.
2. Defines the entry name for the procedure.
3. Declares certain variables as parameters of the procedure.
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 size 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. However, the error may not be detected.

5. Option 2 specifies a procedure to be an interrupt procedure. An 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.

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 as a function it is to return a value with precision WORD. The statement, L1, invokes the function twice.

8.3.12 The Repeat Statement

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 iteration as indicated below.

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

The effect of the above is equivalent to the expansion shown below where T1 is a temporary variable 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

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

General format:

```
return_statement ::=
    RETURN [expression] ;
```

General rules:

1. The expression option must be used if the procedure is declared as a function.
2. The expression option cannot be used if the procedure is not declared as a function.
3. Any number of return statements may appear in a procedure block.

Example:

```
A:  PROCEDURE;
    DECLARE (W,X,Y,Z) WORD;
    B:  PROCEDURE (I) WORD;
        DECLARE I WORD;
            :
        RETURN I;
            :
    END B;
    C:  PROCEDURE (J);
        DECLARE J WORD;
            :
        RETURN;
            :
        RETURN;
            :
    END C;
    W = 1;
    X = B(W)
    CALL B(Y);
    CALL C(Z);
        :
END A;
```

In this example, procedure B is invoked once as a function and once as a subroutine. Procedure C is invoked only as a subroutine. Procedure C cannot be invoked as a function since it is not declared to return a value.

APPENDIX

SYNTAX OF MPL

This appendix gives the actual syntax rules to which the MPL compiler is implemented. The syntax rules which are used in the text are correct as state. However, not all features of the MPL compiler were fully described. The features which were not described are the abbreviations and noise words. The following syntax rules are complete and describe all of the MPL language features. The only primitive which is not specified in the syntax rules is non_quote_char which is self-explanatory.

MPL.SYNTAX

```
 1. program ::=
        external_procedure

 2. external_procedure ::=
        entry_name : external_procedure_head procedure_body [EOF]...

 3. entry_name ::=
        identifier

 4. external_procedure_head ::=
        MAIN [procedure] ; |
        procedure_head

 5. procedure ::=
        PROCEDURE |
        PROC

 6. procedure_head ::=
        INTERRUPT [procedure] ( parameter ) ; |
        procedure [( parameter_list )] [simple_size] ;

 7. parameter ::=
        identifier

 8. parameter_list ::=
        identifier [, identifier]...

 9. simple_size ::=
        BYTE |
        WORD |
        DOUBLE

10. procedure_body ::=
        [declare_statement]... [block_sentence]... end_statement

11. declare_statement ::=
        declare declare_element [, declare_element]... ;

12. declare ::=
        DECLARE |
        DCL

13. declare_element ::=
        identifier literally string |
        entry_name type_attribute |
        item_list size_attribute [area_attribute]

14. literally ::=
        LITERALLY |
        LIT
```

```
15. type_attribute ::=
        [external] procedure [simple_size] |
        ( konstant ) MICRO [simple_size]

16. external ::=
        EXTERNAL |
        EXT

17. item_list ::=
        identifier [( konstant )] |
        ( identifier [( konstant )] [, identifier [( konstant )]]... )

18. size_attribute ::=
        simple_size |
        POINTER [TO] simple_size |
        BIT ( konstant )

19. area_attribute ::=
        external |
        STATIC |
        CONSTANT string |
        CONSTANT ( [+ | -] konstant [, [+ | -] konstant]... ) |
        initial string |
        initial ( [+ | -] konstant [, [+ | -] konstant]... ) |
        BASED konstant |
        BASED identifier

20. initial ::=
        INITIAL |
        INIT

21. block_sentence ::=
        entry_name : procedure_head procedure_body |
        executable_unit

22. executable_unit ::=
        if_statement |
        unconditional_executable_unit

23. if_statement ::=
        if_clause executable_unit |
        if_clause balanced_executable_unit ELSE executable_unit

24. if_clause ::=
        [label_list] if_then

25. if_then ::=
        IF expression THEN
```

```
26. balanced_executable_unit ::=
        if_clause balanced_executable_unit ELSE
             balanced_executable_unit |
        unconditional_executable_unit

27. unconditional_executable_unit ::=
        [label_list] block |
        [label_list] group |
        [label_list] statement

28. label_list ::=
        [label :]...

29. label ::=
        identifier

30. block ::=
        BEGIN ; procedure_body

31. group ::=
        group_heading ; [block_sentence]... end_statement

32. group_heading ::=
        REPEAT expression [TIMES] |
        DO [do_specification]

33. do_specification ::=
        FOREVER |
        WHILE expression |
        CASE expression |
        identifier replace_op expression TO expression [BY expression]

34. replace_op ::=
        = | :=

35. statement ::=
        null_statement |
        return_statement |
        goto_statement |
        call_statement |
        assignment_statement

36. null_statement ::=
        ;

37. return_statement ::=
        RETURN [expression] ;

38. goto_statement ::=
        go_to label ;
```

```
39. go_to ::=
        GOTO |
        GO [TO]

40. call_statement ::=
        CALL procedure_reference ;

41. procedure_reference ::=
        entry_name [( procedure_argument [, procedure_argument]... )]

42. procedure_argument ::=
        expression |
        array_name

43. array_name ::=
        identifier

44. assignment_statement ::=
        storage_reference assignment_operator expression ;

45. assignment_operator ::=
        = | := | +=

46. expression ::=
        conditional_expression |
        simple_expression |
        storage_reference := expression

47. conditional_expression ::=
        if_then expression ELSE expression

48. simple_expression ::=
        logical_term [or_operator logical_term]...

49. or_operator ::=
        ! | ⊥ | XOR

50. logical_term ::=
        logical_factor [& logical_factor]...

51. logical_factor ::=
        numeric_expression [comparison_operator numeric_expression]...

52. comparison_operator ::=
        < | <= | ¬> | = | ¬= | >= | ¬< | > |
        LLT | LLE | LEQ | LNE | LGE | LGT

53. numeric_expression ::=
        numeric_term [add_operator numeric_term]...
```

```
54. add_operator ::=
        + | -

55. numeric_term ::=
        numeric_factor [multiply_operator numeric_factor]...

56. multiply_operator ::=
        * | / | MOD | MULD | DIVD | SLL | SRA | SRL | SLC

57. numeric_factor ::=
        unary_operator numeric_factor |
        numeric_primary

58. unary_operator ::=
        + | - | ¬

59. numeric_primary ::=
        konstant |
        procedure_reference |
        storage_reference |
        @ variable |
        ( expression ) |
        PRTNUM ( entry_name )

60. storage_reference ::=
        reference [$ ( expression [: expression] )]

61. reference ::=
        variable [@ [( expression )]]

62. variable ::=
        identifier [( expression )]

63. end_statement ::=
        [label_list] END [label | entry_name] ;

64. konstant ::=
        constant |
        string

65. constant ::=
        decimal_number |
        bit_string

66. decimal_number ::=
        digit...

67. bit_string ::=
        " [ [( legal_size )] [legal_digit] ]... "
```

```
68. legal_size ::=
        1 | 2 | 3 | 4

69. legal_digit ::=
        digit | A | B | C | D | E | F

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

71. string ::=
        ' [non_quote_char]... ' [string]

72. identifier ::=
        alphabetic_character [alphameric]...

73. alphameric ::=
        alphabetic_character | digit

74. alphabetic_character ::=
        lower_case |
        upper_case |
        # | _

75. lower_case ::=
        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

76. upper_case ::=
        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
```

APPENDIX B

SYNTAX CROSS REFERENCE

The following is a cross reference of the syntax notation variables and reserved key words that appear in the MPL syntax. Each syntax notation variable is preceeded with the syntax rule number in which it is defined. Following each syntax notation variable and reserved key word is a list of syntax rule numbers where they are referenced.

54 add_operator
53
74 alphabetic_character
72,73
73 alphameric
72
19 area_attribute
13
43 array_name
42
44 assignment_statement
35
45 assignment_operator
44
26 balanced_executable_unit
23,26
BASED
19
BEGIN
30
BIT
18
67 bit_string
65
30 block
27
21 block_sentence
10,31
BY
33
BYTE
9
CALL
40
40 call_statement
35
CASE
33
52 comparison_operator
51
47 conditional_expression
46
CONSTANT
19
65 constant
64
DCL
12
66 decimal_number
65
13 declare_element
11
12 declare
11
DECLARE
12
11 declare_statement
10
70 digit
66,73
DIVD
56
DO
32
33 do_specification
32
DOUBLE
9
ELSE
23,26,47
END
63
EOF
2
63 end_statement
10,31
3 entry_name
2,13,21,41,59,63
22 executable_unit
21,23
46 expression
25,32,33,37,42,44,46,47,59,60,61,62
16 external
15,19
EXT
16
EXTERNAL
16
2 external_procedure
1
4 external_procedure_head
2
FOREVER
33
GO
39
39 go_to
38
GOTO
39

38 goto_statement
35
31 group
27
32 group_heading
31
72 identifier
3,7,8,13,17,19,29,33,43,62
IF
25
24 if_clause
23,26
23 if_statement
22
25 if_then
24,47
20 initial
19
INIT
20
INITIAL
20
INTERRUPT
6
17 item_list
13
64 konstant
15,17,18,19,59
29 label
28,38,63
28 label_list
24,27,63
69 legal_digit
67
68 legal_size
67
LEQ
52
LGE
52
LGT
52
LIT
14
14 literally
13
LITERALLY
14
LLE
52
LLT
52
LNE
52
51 logical_factor
50
50 logical_term
48
75 lower_case
74
MAIN
4
MICRO
15
MOD
56
MULD
56
56 multiply_operator
55
non_quote_char
71
36 null_statement
35
53 numeric_expression
51
57 numeric_factor
55,57
59 numeric_primary
57
55 numeric_term
53
49 or_operaotr
48
7 parameter
6
8 parameter_list
6
POINTER
18
PROC
5
PROCEDURE
5
5 procedure
4,6,15
42 procedure_argumment
41
10 procedure_body
2,21,30

6 procedure_head
4,21
41 procedure_reference
40,59
1 program

PRTNUM
59
61 reference
60
REPEAT
32
34 replace_op
33
RETURN
37
37 return_statement
35
48 simple_expression
46
9 simple_size
6,15,18
18 size_attribute
13
SLC
56
SLL
56
SRA
56
SRL
56
35 statement
27
STATIC
19
60 storage_reference
44,46,59
71 string
13,19,64,71
THEN
25
TIMES
32
TO
18,33,39
15 type_attribute
13
58 unary_operator
57
27 unconditional_executable_unit
22,26
76 upper_case
74
62 variable
59,61
WHILE
33
WORD
9
XOR
49

APPENDIX C

COMPILER TOGGLES

Control over the compilation process is through the mechanism of toggles. Toggles are internal to the source program; there is no external operating system mechanism for controlling compiler operation. Each toggle controls a single aspect of compiler operation and has a binary value: on or off.

All toggles assume a default value when the compiler is invoked. Toggles change during program compilation as specified by the user or occasionally are altered by the compiler. At the start of subsequent programs toggles reassume their default values unless frozen by the P-toggle.

Toggles may be altered by the user in any comment string. Three characters are used as toggle control operators:

$ (dollar sign)	- If on set off, if off set on
& (ampersand)	- Set on
% (percent sign)	- Set off

The character that immediately follows the toggle operator is interpreted as the toggle to be altered. Should this character not be an implemented toggle, nothing is done.

Toggle changes are honored by the compiler as soon as they occur. Since a listing line is buffered, toggles that affect the listing format become effective before the line in which they appear is listed. Since the compiler parse is top-down, toggles that affect object generation should appear before the external procedure entry.

TOGGLE	DEFAULT SETTING	FUNCTION
A	On	List symbol table
B	On	Ignore high source bit
C	Off	List generated code
D	Off	Continue object on blunder
E	Off	Space for top-of-form
F	On	List flags
H	Off	List object program
I	On	Indent code listing
L	On	List source program
M	On	Honor memory checks
N	Off	Format for narrow page
O	Off	Generate object program
P	On	Reset toggles at program start
Q	Off	Chop source program listing
R	On	Collect symbol references
S	On	Format for short page
U	Off	Upspace listing
V	Off	Check memory each record
W	Off	Suppress warnings
X	Off	Continue on abort
Y	On	List program summary
Z	On	Honor listing requests (A C F H L Y)
#	Off	Honor object in source
?	Off	Honor early source truncation

APPENDIX D

32/S STANDARD SYMBOL-CODE CORRESPONDENCES

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
00	0		12-0-9-8-1	NUL	00	✓			NULL:used for time or media fill
01	1		12-9-1	SOH	01	✓			Start of Header: Communications Control
02	2		12-9-2	STX	02	✓			Start of Text: Communications Control
03	3		12-9-3	ETX	03	✓			End of Text: Communications Control
04	4		9-7	EOT	37	✓			End of Transmission: Communications Control
05	5		0-9-8-5	ENQ	2D	✓			Enquiry: Communications Control
06	6		0-9-8-6	ACK	2E	✓			Acknowledge: Communications Control
07	7		0-9-8-7	BEL	2F	✓			Bell: An audible signal
08	8		11-9-6	BS	16	✓			Backspace:
09	9		12-9-5	HT	05	✓			Horizontal Tab:
0A	10		0-9-5	LF	25	✓			Line Feed:also (New Line)
0B	11		12-9-8-3	VT	0B	✓			Vertical Tab;
0C	12		12-9-8-4	FF	0C	✓			Form Feed:
0D	13		12-9-8-5	CR	0D	✓			Carriage Return:
0E	14		12-9-8-6	SO	0E	✓			Shift Out: Code Control
0F	15		12-9-8-7	SI	0F	✓			Shift In: Code Control

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
10	16		12-11-9-8-1	DLE	10	✓			Data Link Escape: Communications Control
11	17		11-9-1	DC1	11	✓			Device Control 1: (X-ON) Tape reader on
12	18		11-9-2	DC2	12	✓			Device Control 2: Punch on
13	19		11-9-3	DC3	13	✓			Device Control 3: (X-OFF) Tape reader off
14	20		9-8-4	DC4	3C	✓			Device Control 4: (Stop) Punch off
15	21		9-8-5	NAK	3D	✓			Negative Acknowledge: Communications Control
16	22		9-2	SYN	32	✓			Synchronous Idle: Communications Control
17	23		0-9-6	ETB	26	✓			End of Transmission Block: Communications Control
18	24		11-9-8	CAN	18	✓			Cancel:
19	25		11-9-8-1	EM	19	✓			End of Medium:
1A	26		9-8-7	SUB	3F	✓			Substitute:
1B	27		0-9-7	ESC	27	✓			Escape:
1C	28		11-9-8-4	FS	1C	✓			File Separator:
1D	29		11-9-8-5	GS	1D	✓			Group Separator:
1E	30		11-9-8-6	RS	1E	✓			Record Separator:
1F	31		11-9-8-7	US	1F	✓			Unit Separator:

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
20	32			SP	40	√	b̸	SP	Space:
21	33	!	11-8-2	Explanation Point	5A	√	!	!	
22	34	"	8-7	Quotation Marks	7F	√	"	"	
23	35	#	8-3	Number Sign	7B	√	#	#	
24	36	$	11-8-3	Dollar Sign	5B	√	$	$	
25	37	%	0-8-4	Percent	6C	√	%	%	
26	38	&	12	Ampersand	50	√	&	&	
27	39	'	8-5	Apostrophe	7D	√	'	'	
28	40	(	12-8-5	Opening Parenthesis	4D	√	(	(	
29	41	)	11-8-5	Closing Parenthesis	5D	√	)	)	
2A	42	*	11-8-4	Asterisk	5C	√	*	*	
2B	43	+	12-8-6	Plus	4E	√	+	+	
2C	44	,	0-8-3	Comma	6B	√	,	,	
2D	45	-	11	Hyphen	60	√	-	-	(Minus)
2E	46	.	12-8-3	Period	4B	√	.	.	(Decimal Point)
2F	47	/	0-1	Slant	61	√	/	/	(Slash)

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
30	48	0	0	Zero	F0	✓	0	0	
31	49	1	1	One	F1	✓	1	1	
32	50	2	2	Two	F2	✓	2	2	
33	51	3	3	Three	F3	✓	3	3	
34	52	4	4	Four	F4	✓	4	4	
35	53	5	5	Five	F5	✓	5	5	
36	54	6	6	Six	F6	✓	6	6	
37	55	7	7	Seven	F7	✓	7	7	
38	56	8	8	Eight	F8	✓	8	8	
39	57	9	9	Nine	F9	✓	9	9	
3A	58	:	8-2	Colon	7A	✓	:	:	
3B	59	;	11-8-6	Semicolon	5E	✓	;	;	
3C	60	<	12-8-4	Less Than	4C	✓	<	<	
3D	61	=	8-6	Equals	7E	✓	=	=	
3E	62	>	0-8-6	Greater Than	6E	✓	>	>	
3F	63	?	0-8-7	Question Mark	6F	✓	?	?	

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
40	64	@	8-4	Commercial At	7C	✓	@	@	
41	65	A	12-1	Upper Case A	C1	✓	A	A	
42	66	B	12-2	Upper Case B	C2	✓	B	B	
43	67	C	12-3	Upper Case C	C3	✓	C	C	
44	68	D	12-4	Upper Case D	C4	✓	D	D	
45	69	E	12-5	upper Case E	C5	✓	E	E	
46	70	F	12-6	Upper Case F	C6	✓	F	F	
47	71	G	12-7	Upper Case G	C7	✓	G	G	
48	72	H	12-8	Upper Case H	C8	✓	H	H	
49	73	I	12-9	Upper Case I	C9	✓	I	I	
4A	74	J	11-1	Upper Case J	D1	✓	J	J	
4B	75	K	11-2	Upper Case K	D2	✓	K	K	
4C	76	L	11-3	Upper Case L	D3	✓	L	L	
4D	77	M	11-4	Upper Case M	D4	✓	M	M	
4E	78	N	11-5	Upper Case N	D5	✓	N	N	
4F	79	O	11-6	Upper Case O	D6	✓	O	O	

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
50	80	P	11-7	Upper Case P	D7	√	P	P	
51	81	Q	11-8	Upper Case Q	D8	√	Q	Q	
52	82	R	11-9	Upper Case R	D9	√	R	R	
53	83	S	0-2	Upper Case S	E2	√	S	S	
54	84	T	0-3	Upper Case T	E3	√	T	T	
55	85	U	0-4	Upper Case U	E4	√	U	U	
56	86	V	0-5	Upper Case V	E5	√	V	V	
57	87	W	0-6	Upper Case W	E6	√	W	W	
58	88	X	0-7	Upper Case X	E7	√	X	X	
59	89	Y	0-8	Upper Case Y	E8	√	Y	Y	
5A	90	Z	0-9	Upper Case Z	E9	√	Z	Z	
5B	91	[	12-8-2	Opening Bracket	4A	√	¢		
5C	92	\	0-8-2	Reverse Slant	E0	√	0 8 2		
5D	93	]	12-11	Closing Bracket	6A	√			
5E	94	^	11-8-7	Circumflex	5F	√			↑ on some devices
5F	95	_	0-8-5	Underline	6D	√	_	_	← on some devices

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
60	96	`	8-1	Accent Grave	79				
61	97	a	12-0-1	Lower Case a	81			a	
62	98	b	12-0-2	Lower Case b	82			b	
63	99	c	12-0-3	Lower Case c	83			c	
64	100	d	12-0-4	Lower Case d	84			d	
65	101	e	12-0-5	Lower Case e	85			e	
66	102	f	12-0-6	Lower Case f	86			f	
67	103	g	12-0-7	Lower Case g	87			g	
68	104	h	12-0-8	Lower Case h	88			h	
69	105	i	12-0-9	Lower Case i	89			i	
6A	106	j	12-11-1	Lower Case j	91			j	
6B	107	k	12-11-2	Lower Case k	92			k	
6C	108	l	12-11-3	Lower Case l	93			l	
6D	109	m	12-11-4	Lower Case m	94			m	
6E	110	n	12-11-5	Lower Case n	95			n	
6F	111	o	12-11-6	Lower Case o	96			o	

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
70	112	p	12-11-7	Lower Case p	97			p	
71	113	q	12-11-8	Lower Case q	98			q	
72	114	r	12-11-9	Lower Case r	99			r	
73	115	s	11-0-2	Lower Case s	A2			s	
74	116	t	11-0-3	Lower Case t	A3			t	
75	117	u	11-0-4	Lower Case u	A4			u	
76	118	v	11-0-5	Lower Case v	A5			v	
77	119	w	11-0-6	Lower Case w	A6			w	
78	120	x	11-0-7	Lower Case x	A7			x	
79	121	y	11-0-8	Lower Case y	A8			y	
7A	122	z	11-0-9	Lower Case z	A9			z	
7B	123	{	12-0	Opening Brace	C0				
7C	124	|	12-8-7	Vertical Line	4F		|	|	
7D	125	}	11-0	Closing Brace	D0				
7E	126	~	11-0-1	Overline	A1				(Tilde)
7F	127		12-9-7	DEL	07	✓			Delete:used to erase characters on paper tape

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
80	128		11-0-9-8-1		20				
81	129		0-9-1		21				
82	130		0-9-2		22				
83	131		0-9-3		23				
84	132		0-9-4		24				
85	133		11-9-5		15				
86	134		12-9-6		06				
87	135		11-9-7		17				
88	136		0-9-8		28				
89	137		0-9-8-1		29				
8A	138		0-9-8-2		2A				
8B	139		0-9-8-3		2B				
8C	140		0-9-8-4		2C				
8D	141		12-9-8-1		09				
8E	142		12-9-8-2		0A				
8F	143		11-9-8-3		1B				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
90	144		12-11-0-9-8-1		30				
91	145		9-1		31				
92	146		11-9-8-2		1A				
93	147		9-3		33				
94	148		9-4		34				
95	149		9-5		35				
96	150		9-6		36				
97	151		12-9-8		08				
98	152		9-8		38				
99	153		9-8-1		39				
9A	154		9-8-2		3A				
9B	155		9-8-3		3B				
9C	156		12-9-4		04				
9D	157		11-9-4		14				
9E	158		9-8-6		3E				
9F	159		11-0-9-1		E1				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
A0	160		12-0-9-1		41				
A1	161		12-0-9-2		42				
A2	162		12-0-9-3		43				
A3	163		12-0-9-4		44				
A4	164		12-0-9-5		45				
A5	165		12-0-9-6		46				
A6	166		12-0-9-7		47				
A7	167		12-0-9-8		48				
A8	168		12-8-1		49				
A9	169		12-11-9-1		51				
AA	170		12-11-9-2		52				
AB	171		12-11-9-3		53				
AC	172		12-11-9-4		54				
AD	173		12-11-9-5		55				
AE	174		12-11-9-6		56				
AF	175		12-11-9-7		57				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
B0	176		12-11-9-8		58				
B1	177		11-8-1		59				
B2	178		11-0-9-2		62				
B3	179		11-0-9-3		63				
B4	180		11-0-9-4		64				
B5	181		11-0-9-5		65				
B6	182		11-0-9-6		66				
B7	183		11-0-9-7		67				
B8	184		11-0-9-8		68				
B9	185		0-8-1		69				
BA	186		12-11-0		70				
BB	187		12-11-0-9-1		71				
BC	188		12-11-0-9-2		72				
BD	189		12-11-0-9-3		73				
BE	190		12-11-0-9-4		74				
BF	191		12-11-0-9-5		75				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
C0	192		12-11-0-9-6		76				
C1	193		12-11-0-9-7		77				
C2	194		12-11-0-9-8		78				
C3	195		12-0-8-1		80				
C4	196		12-0-8-2		8A				
C5	197		12-0-8-3		8B				
C6	198		12-0-8-4		8C				
C7	199		12-0-8-5		8D				
C8	200		12-0-8-6		8E				
C9	201		12-0-8-7		8F				
CA	202		12-11-8-1		90				
CB	203		12-11-8-2		9A				
CC	204		12-11-8-3		9B				
CD	205		12-11-8-4		9C				
CE	206		12-11-8-5		9D				
CF	207		12-11-8-6		9E				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
D0	208		12-11-8-7		9F				
D1	209		11-0-8-1		A0				
D2	210		11-0-8-2		AA				
D3	211		11-0-8-3		AB				
D4	212		11-0-8-4		AC				
D5	213		11-0-8-5		AD				
D6	214		11-0-8-6		AE				
D7	215		11-0-8-7		AF				
D8	216		12-11-0-8-1		B0				
D9	217		12-11-0-1		B1				
DA	218		12-11-0-2		B2				
DB	219		12-11-0-3		B3				
DC	220		12-11-0-4		B4				
DD	221		12-11-0-5		B5				
DE	222		12-11-0-6		B6				
DF	223		12-11-0-7		B7				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
E0	224		12-11-0-8		B8				
E1	225		12-11-0-9		B9				
E2	226		12-11-0-8-2		BA				
E3	227		12-11-0-8-3		BB				
E4	228		12-11-0-8-4		BC				
E5	229		12-11-0-8-5		BD				
E6	230		12-11-0-8-6		BE				
E7	231		12-11-0-8-7		BF				
E8	232		12-0-9-8-2		CA				
E9	233		12-0-9-8-3		CB				
EA	234		12-0-9-8-4		CC				
EB	235		12-0-9-8-5		CD				
EC	236		12-0-9-8-6		CE				
ED	237		12-0-9-8-7		CF				
EE	238		12-11-9-8-2		DA				
EF	239		12-11-9-8-3		DB				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

HEX	DEC	ASCII	HOLLERITH	NAME	Δ	Θ	∇	§	COMMENTS
F0	240		12-11-9-8-4		DC				
F1	241		12-11-9-8-5		DD				
F2	242		12-11-9-8-6		DE				
F3	243		12-11-9-8-7		DF				
F4	244		11-0-9-8-2		EA				
F5	245		11-0-9-8-3		EB				
F6	246		11-0-9-8-4		EC				
F7	247		11-0-9-8-5		ED				
F8	248		11-0-9-8-6		EE				
F9	249		11-0-9-8-7		EF				
FA	250		12-11-0-9-8-2		FA				
FB	251		12-11-0-9-8-3		FB				
FC	252		12-11-0-9-8-4		FC				
FD	253		12-11-0-9-8-5		FD				
FE	254		12-11-0-9-8-6		FE				
FF	255		12-11-0-9-8-7		FF				

Δ - Equivalent EBCDIC code
Θ - Available on teletype model 33
∇ - Equivalent character on 029
§ - Equivalent character on 2741

APPENDIX E

SAMPLE MPL PROGRAM

```
PRIME     75SEP02    11:09:54  MPL 1.0           ***    P R O G R A M    L I S T I N G    ***          PAGE   1

 DEC  HEX LINE  ---------SOURCE---------------------------------------------------------------------  DL BN LL  BLOCK

  0 0000    1  PRIME:                                                                                   0  0  0
  0 0000    2  /***********************************************************************/                0  0  0
  0 0000    3  /*  THIS IS A PROCEDURE TO PRINT ALL PRIME NUMBERS FROM 1 TO 1000      */                0  0  0
  0 0000    4  /***********************************************************************/                0  0  0
  0 0000    5       MAIN PROCEDURE;                                                                     1  1  1  PRIME
  0 0000    6       DECLARE                                                                             1  1  1  PRIME
  0 0000    7            PRINT PROCEDURE,                                                               1  1  1  PRIME
  0 0000    8            P(200) WORD, /* THE ARRAY OF PRIMES */                                         1  1  1  PRIME
  0 0000    9            TEST WORD, /* VALUE BEING TESTED */                                            1  1  1  PRIME
  0 0000   10            (I,J) WORD; /* SUBSCRIPTS */                                                   1  1  1  PRIME
  0 0000   11       DO I = 1 TO 3;                                                                      2  1  1  PRIME
 13 000D   12            CALL PRINT(I,P(I):=I);                                                         2  1  1  PRIME
 27 001B   13       END;                                                                                1  1  1  PRIME
 29 001D   14       TEST = 5;                                                                           1  1  1  PRIME
 32 0020   15       DO WHILE TEST < 1000;                                                               2  1  1  PRIME
 41 0029   16            J = 3;                                                                         2  1  1  PRIME
 44 002C   17            DO WHILE TEST/P(J) >= P(J);                                                    3  1  1  PRIME
 59 003B   18                 IF TEST MOD P(J) = 0 THEN GOTO PR01;                                      3  1  1  PRIME
 75 004B   19                 J += 1;                                                                   3  1  1  PRIME
 78 004E   20            END;                                                                           2  1  1  PRIME
 80 0050   21            CALL PRINT(I,P(I):=TEST);                                                      2  1  1  PRIME
 94 005E   22            I += 1;                                                                        2  1  1  PRIME
 97 0061   23            PR01:                                                                          2  1  1  PRIME
 97 0061   24            TEST += 2;                                                                     2  1  1  PRIME
103 0067   25       END;                                                                                1  1  1  PRIME
105 0069   26       /*****************************************************************/                 1  1  1  PRIME
105 0069   27       /*   THIS IS A PROCEDURE TO PRINT TWO DECIMAL NUMBERS             */                 1  1  1  PRIME
105 0069   28       /*****************************************************************/                 1  1  1  PRIME
105 0069   29       PRINT:                                                                              1  1  1  PRIME
105 0069   30            PROCEDURE (L,M);                                                               2  2  2  PRINT
108 006C   31            DECLARE                                                                        2  2  2  PRINT
108 006C   32                 (L,M) WORD, /* NUMBERS TO PRINT */                                        2  2  2  PRINT
108 006C   33                 I WORD, /* LOOP INDEX */                                                  2  2  2  PRINT
108 006C   34                 TYPE PROCEDURE;                                                           2  2  2  PRINT
108 006C   35            /************************************************************/                 2  2  2  PRINT
108 006C   36            /*   THIS IS A PROCEDURE TO CONVERT A 16 BIT BINARY VALUE TO */                 2  2  2  PRINT
108 006C   37            /*   FIVE PRINTEABLE DECIMAL CHARACTERS AND PRINT THEM       */                 2  2  2  PRINT
108 006C   38            /************************************************************/                 2  2  2  PRINT
108 006C   39            DECCONV:                                                                       2  2  2  PRINT
111 006F   40                 PROCEDURE (K);                                                            3  3  3  DECCONV
```

```
PRIME     75SEP02    11:09:54  MPL 1.0           ***    P R O G R A M    L I S T I N G    ***          PAGE   2

 DEC  HEX LINE  ---------SOURCE---------------------------------------------------------------------  DL BN LL  BLOCK

114 0072   41                 DECLARE                                                                   3  3  3  DECCONV
114 0072   42                      ASCII(4) BYTE,                                                       3  3  3  DECCONV
114 0072   43                      K WORD;                                                              3  3  3  DECCONV
114 0072   44                 DO I = 4 TO 0 BY -1;                                                      4  3  3  DECCONV
128 0080   45                      ASCII(I) = K MOD 10 + '0';                                           4  3  3  DECCONV
142 008E   46                      K = K/10;                                                            4  3  3  DECCONV
148 0094   47                 END;                                                                      3  3  3  DECCONV
150 0096   48                 DO I = 0 TO 3;                                                            4  3  3  DECCONV
160 00A0   49                      IF ASCII(I) ^= '0' THEN GOTO DECCONV1;                               4  3  3  DECCONV
177 00B1   50                      ASCII(I) = ' ';                                                      4  3  3  DECCONV
186 00BA   51                 END;                                                                      3  3  3  DECCONV
188 00BC   52                 DECCONV1:                                                                 3  3  3  DECCONV
188 00BC   53                 DO I = 0 TO 4;                                                            4  3  3  DECCONV
201 00C9   54                      CALL TYPE(ASCII(I));                                                 4  3  3  DECCONV
214 00D6   55                 END DECCONV1;                                                             3  3  3  DECCONV
216 00D8   56            END DECCONV;                                                                   2  2  2  PRINT
217 00D9   57            /************************************************************/                 2  2  2  PRINT
217 00D9   58            /*   THIS IS A PROCEDURE TO OUTPUT THE SPECIFIED CHARACTER   */                 2  2  2  PRINT
217 00D9   59            /*   TO THE TELETYPE                                         */                 2  2  2  PRINT
217 00D9   60            /************************************************************/                 2  2  2  PRINT
217 00D9   61            TYPE:                                                                          2  2  2  PRINT
217 00D9   62                 PROCEDURE (N);                                                            3  4  3  TYPE
220 00DC   63                 DECLARE                                                                   3  4  3  TYPE
220 00DC   64                      N BYTE,                                                              3  4  3  TYPE
220 00DC   65                      TTY(7) WORD BASED "F004";                                            3  4  3  TYPE
220 00DC   66                 DO WHILE TTY(0); END; /* WAIT UNTIL CONTROLLER NOT BUSY */                3  4  3  TYPE
233 00E9   67                 TTY(1) = "2";   /* START THE TTY */                                       3  4  3  TYPE
239 00EF   68                 DO WHILE TTY(0)&"4"=0; END; /* WAIT FOR DATA SERVICE */                   3  4  3  TYPE
253 00FD   69                 TTY(2) = N; /* OUTPUT THE CHACT */                                        3  4  3  TYPE
260 0104   70                 TTY(1) = 4; /* STOP THE TTY */                                            3  4  3  TYPE
266 010A   71            END TYPE;                                                                      2  2  2  PRINT
267 010B   72            /* NEXT STATEMENT IS START OF PROCEDURE 'PRINT' */                             2  2  2  PRINT
267 010B   73            CALL TYPE("0D"); /* OUTPUT A CARRIAGE RETURN */                                2  2  2  PRINT
274 0112   74            CALL TYPE("0A"); /* OUTPUT A LINE FEED */                                      2  2  2  PRINT
281 0119   75            CALL DECCONV(L);                                                               2  2  2  PRINT
289 0121   76            CALL TYPE(' '); /* OUTPUT A SPACE */                                           2  2  2  PRINT
297 0129   77            CALL DECCONV(M);                                                               2  2  2  PRINT
305 0131   78       END PRINT;                                                                          1  1  1  PRIME
306 0132   79  END PRIME;                                                                               0  0  0
```

NAME	DEF	BN	LL ST	DEC	HEX	CLASS	SCOPE	SIZE	SET	DIM	REFERENCES
ASCII	42	3	3	10	000A	AUTO	I	BYTE		4	45 49 50 54
DECCONV	39	2	2	114	0072	PROCEDURE	I				56 75 77
DECCONV1	53	3	3	188	00BC	DOLABEL	I				49 55
I	10	1	1	412	019C	AUTO	I	WORD		SCALR	11 12 12 12 21 21 22
I	33	2	2	12	000C	AUTO	I	WORD		SCALR	44 45 48 49 50 53 54
J	10	1	1	414	019E	AUTO	I	WORD		SCALR	16 17 17 18 19
K	43	3	3	8	0008	PARAMETER	I	WORD		SCALR	40 45 46 46
L	32	2	2	8	0008	PARAMETER	I	WORD		SCALR	30 75
M	32	2	2	10	000A	PARAMETER	I	WORD		SCALR	30 77
N	64	4	3	9	0009	PARAMETER	I	BYTE		SCALR	62 69
P	8	1	1	8	0008	AUTO	I	WORD		200	12 17 17 18 21
PR01	24	1	1	97	0061	LABEL	I				18
PRIME	1	0	0	0	0000	MAIN	EXT				79
PRINT	7	1	1	108	006C	PROCEDURE	I				12 21 30 78
TEST	9	1	1	410	019A	AUTO	I	WORD		SCALR	14 15 17 18 21 24
TTY	65	4	3	61444	F004	CBASE	I	WORD		ARRAY	66 67 68 69 70
TYPE	34	2	2	220	00DC	PROCEDURE	I				54 62 71 73 74 76

F L A G S	SOURCE PROGRAM	OBJECT PROGRAM	SYMBOL TABLE	COMPILER STACK
0 ABORTS	79 LINES	307 BYTES PROGRAM	592 BYTES USED	5211 BYTES USED
0 BLUNDERS	49 STATEMENTS	0 BYTES STATIC	7484 BYTES SPARE	789 BYTES SPARE
0 ERRORS	4 BLOCKS	438 BYTES STACK(1)	17 SYMBOLS	259 ACCESSES
0 WARNINGS	3 LEXDEPTH	17 OBJECT RECORDS	66 REFERENCES	58 COLLISIONS

TOGGLES OFF: # ? C D H N O P Q U V W X

TOGGLES ON: A B E F I L M R S Y Z

N O F L A G S