codex

A Subsidiary of MOTOROLA INC.

MPL Program Reference Manual

Document No. 72791

First Edition
Through Version 5.0

Preface

This User's Guide provides a description of MPL, a high-level systems programming language used with the Codex Disk Operating System (CODOS) intended for programmers and other technically-oriented personnel. This Guide includes general information, descriptions of program formats, constants, variables, expressions, and statements, and sample MPL programs.

Other Codex publications that may be of interest include the Codex Disk Operating System (CODOS) User's Guide, the CODOS Reference Manual, and the Operator's Guides and Hardware Reference Manuals appropriate to the user's system configuration.

CHAPTER 1. INTRODUCTION

MPL is a high-level systems programming language. Designed for use with Codex Intelligent Terminal Systems, MPL permits users to generate operating systems, utility routines, and other system software with a minimum of programming time and effort. Based on the popular PL/1 syntax, MPL simplifies the translation from functional requirements to an operating program.

MPL is a modular programming language designed for flexibility and ease of use. The high level of self-documentation makes MPL programs easy to read and write; MPL's block structure encourages software modularity and structured programming. Free-format input simplifies MPL program writing, reducing training requirements and development times. MPL programs can be optimized for execution speed or memory space quickly and without rewriting the assembly language output. In addition, the high-level orientation of MPL permits emphasis on correcting algorithms and design flaws rather than on the details of an assembly language implementation.

MPL output is in assembly language, permitting the user to add additional assembly language program segments. This capability allows the programmer to develop different programs by adding different subroutines to a single MPL program acting as a basic framework.

Hardware Support Required

The minimum hardware configuration required to support MPL consists of:

- -- CDX-68 Basic Display Terminal with the appropriate firmware options
- -- 56k bytes of user memory (RAM)
- -- .5 Mb or 1 Mb Diskette Storage (CDX-FS Series) or 10 Mb Disk Storage (CDX-FS/DR)
- -- Microcomputer Module D (CDX-SBC/D)
- -- System Self-Test firmware package (CDX-SST/D)

Optional Hardware Supported

MPL also supports a variety of printers, including Matrix and Character printers (the Codex SP Series). These optional printers are linked to the Basic Display Terminal through either the Microcomputer Module D or the Printer Interface Module (CDX-PI).

Software Support Required

No additional software is required to run the MPL as it comes shipped on the system disk.

Software Installation

There is no software installation that need be performed. All MPL software is on the disk containing the selected software package.

CHAPTER 2. STATEMENT TYPES AND ORDER

An MPL program is a sequence of procedures; a procedure is a named routine that performs a task. Procedures called subroutine procedures execute when called from within procedures.

Procedures are sequences of statements defining (1) the type and arrangement of the data and (2) the sequence of actions. A procedure is composed of any combination of the following statements.

The PROCEDURE statement denotes that the following part of the program is a procedure. It specifies the procedure's name, and whether it is a main or a subroutine procedure. If it is a subroutine procedure, the statement specifies the assigned values of any parameters.

DECLARE statements define the type and arrangement of the data used by the procedure. They specify names, possible initial values, and other attributes of the data items.

Executable statements define the sequence of actions when the procedure is executed. They reference the data items named in the "DECLARE" statements.

The RETURN statement is used only with subroutine procedures. It specifies return from the procedure to the point of call. It may, in some cases, specify results that are to be returned.

The END statement denotes the end of the procedure.

Program Format

Record Format

The statements of an MPL program must be located in one or more CODOS ASCII-record disk files. The disk files are organized into records in the following format:

[<sequence number> <space>] <sequence of characters> <CR>

The "<sequence number>" is an optional four-digit number. If used, it must appear in each record. The "<sequence of characters>" are characters from the ASCII character set. The compiler recognizes a maximum input record of 80 characters. The "<space>" is an ASCII space character. The "<CR>" is an ASCII carriage return character.

Except for the restrictions, (see Restrictions), the statement formats for records are unrestricted. A statement too long for one record continues onto successive records. Multiple statements may appear on a record when separated by spaces, semicolons, or comments.

Character Set

MPL programs are written with the ASCII character set. Character-string constants and comments may contain any displayable ASCII characters. Other language elements, such as arithmetic operators and variables, may contain characters chosen from a subset of ASCII, called the MPL character set. The MPL character set is alphabetic and numeric characters, collectively called alphameric and special characters.

The alphabetic characters are the characters A through Z (upper-case only).

The numeric characters or the decimal digits are the characters 0 through 9. The decimal digits and the characters A through F are hexadecimal digits.

The special characters and their meanings or uses, outside of character-string constants and comments, are as follows:

Character	Name	Meaning or Use
	Space	Separator, otherwise ignored
!	Exclamation point	Start of comment
$	Dollar sign	Embedded assembly language or start of hex constant
%	Percent sign	Rotate or arithmetic shift operator
&	Ampersand	Logical and operator
'	Single quote	Character-string constant
(	Left parenthesis	Grouping or begin argument list
)	Right parenthesis	Grouping or end argument list
*	Asterisk	Multiply
+	Plus sign	Add
,	Comma	Separator
-	Minus sign	Subtract or minus
.	Period	Decimal point
/	Slash	Divide
:	Colon	Label
;	Semicolon	Separator, otherwise ignored
<	Less-than sign	Less than or begin argument list
=	Equal sign	Assignment
>	Greater-than sign	Greater than or end argument list

Certain symbols, called two-character symbols, are composed of pairs of adjacent characters:

Symbol	Name	Meaning or Use
->	Arrow	Pointer
/*	Slash-asterisk	Begin comment
*/	Asterisk-slash	End comment

Identifiers

The programmer may assign names to statements and data items in an MPL program. These names, together with words reserved by the compiler for special purposes, are identifiers. A name consists of a string of (1 to 6) alphameric characters. The first character must always be alphabetic. Any such string may be assigned a name, provided that the following two conditions are met:

. The string is not already assigned in the MPL program.

. The string is not a word reserved by the compiler for a special purpose.

Note: names may not contain special characters.

The compiler's reserved words are the following:

A	DCL	GIVING	LE	RETURN
ADDR	DEC	GLOBAL	LONG	SHIFT
AND	DECLARE	GO	LT	SHORT
B	DEF	GOTO	MAIN	SIGNED
BASED	DEFINED	GT	NARG	SS
BIN	DO	IAND	NE	THEN
BIT	DSCT	IEOR	NOT	TO
BSCT	ELSE	IF	OPTIONS	WHILE
BY	END	INIT	OR	X
CALL	EQ	INITIAL	PROC	
CHAR	EXTERNAL	IOR	PROCEDURE	
CSCT	GE	LABEL	PSCT	

The compiler interprets the longest possible string of adjoining characters as an identifier. Because of this, an identifier may not be followed by another identifier or by a numeric constant without a separator, such as a "space," interposed. For example:

Valid Names	Invalid Names	
FRED	INIT	(reserved)
COST23	2TIMES	(initial digit)
X1Y1Z1	A.B	(not alphameric)
SMALL	small	(not alphameric)
T	$ABC	(not alphameric)

Spaces and Semicolons

Spaces, semicolons, or comments must be used to separate adjacent identifiers or numeric constants. They may not be used within identifiers, numeric constants, or two-character symbols. They are only significant as data character-string constants. In address constants, spaces and semicolons are ignored, while comments may not be used. Except for these rules and the restrictions (see Restrictions), spaces, semicolons, or comments may appear throughout an MPL program.

Labels

MPL statements may be preceded by statement labels. A statement label is a name that is immediately followed by a colon. Control may be transferred to a labelled statement to alter sequential statement execution. "PROCEDURE" statements must have a label.

Comments

Comments designate character strings that document a program or explain the function of various statements or procedures. Comments print when the program is listed, but are otherwise ignored by the compiler. Comments have the following two formats:

- /* <character string> */

  where the "<character string>" may span several records and contain any displayable ASCII characters, provided that it does not contain an asterisk-slash. The slash-asterisk and asterisk-slash are two-character symbols, with the two characters adjacent on the same record.

- ! <character string>

  where the "<character string>" is the remainder of the "<sequence of characters>" on the record containing the exclamation point. It may contain any displayable ASCII characters.

Comments act only as separators unless they appear in character-string constants. Comments within character-string constants are considered as data. Comments may not be used in address constants.

Data

A data item is a storage area with a value and attributes. It is represented in the program text by a constant or a variable.

All data items have certain attributes. For example, a data item may have values that are decimal numbers with six significant digits and two digits to the right of the decimal point. It may have values that are strings of ten characters; or it may have values that are pointers to other data items.

A constant data item is a data item whose value does not change when a program executes. The attributes of a constant data item may be determined from the textual form of the constant that denotes the data item and the context in which it appears. For a variable data item, a data item whose value may change when a program executes, the attributes must be declared to the compiler.

There are four classes of data: arithmetic, string, label, and pointer.

Arithmetic Data

Arithmetic data items have numeric values. They have the attributes of base, precision, and sign. A data item's base attribute is either binary (BIN) or decimal (DEC). The precision attribute specifies the amount of storage the value requires and, if its base is "DEC," the number of decimal places. The sign attribute states whether the data item's value is non-negative (default) or not (SIGNED).

String Data

String data items have values that are strings of displayable ASCII characters or binary digits. They have the attribute of length. A data item's length attribute specifies the number of displayable ASCII characters or binary digits it contains.

Label Data

Label data items have values that are statement labels. The label data type is listed below:

LABEL

A data item of this type requires two bytes of storage. It assumes values that are statement labels.

Pointer Data

Pointer data items have values that are the memory addresses of variable data items. They are called pointer data items because they "point" to the data items whose addresses are their value.

MPL does not have a special reserved word signifying the pointer data type. Instead, pointer data items are considered to be of type "BIN(2)."

Constants

Constant data items are data items whose values do not change when a program executes. They are represented in the program text by character strings called "constants." A constant is said to "denote" the data item it represents. Occasionally the phrase, "the value of the constant," is used to mean "the value of the data item denoted by the constant." This section describes the format of constants.

Corresponding to the four classes of data: arithmetic, string, label, and pointer, there are four classes of constants. These are: numeric, character-string, label, and address.

Numeric Constants

Numeric constants are used to denote arithmetic data items. A numeric constant may not contain spaces, semicolons, or comments. There are three subclasses: integer, decimal, and hexadecimal.

Integer constants

An integer constant is a string of (a maximum) 30 decimal digits. For example:

Valid	Invalid
26	2A
131072	-4.60
0	+2

Decimal constants

A decimal constant is a string of (a maximum) 22 decimal digits. It is followed by a decimal point and optionally followed by a string of (a maximum) 30 decimal digits. For example:

Valid	Invalid
2698.273	+26.03
00.00	2,468
0.1	.1

Hexadecimal constants

A hexadecimal constant is a dollar sign followed by a string of (a maximum) 29 hexadecimal digits. For example:

Valid	Invalid
$FFFF	FFFF
$0	0F3H
$2A	$$2
$0D	$12XF

Character-string Constants

Character-string constants denote data items whose values are strings of (a maximum) 30 displayable ASCII characters. A character-string constant is a string of ASCII characters enclosed in single quotation marks. Within the string, a single quotation mark is represented by two adjacent single quotation marks. An exclamation point is represented by two adjacent exclamation points. Spaces, semicolons, and comments within the string are significant data. For example:

Valid	Invalid
'HELLO!!'	'!'
'''QUOTE'''	'A
'THIS IS A STRING'	'IT'S'

There are no constants to denote data items whose values are strings of binary digits.

Label Constants

Label constants are used to denote label data items. A label constant is a name assigned to label a statement in the program.

Address Constants

Address constants are used to denote pointer data items. An address constant is an expression, enclosed in parentheses, preceded by the reserved word, "ADDR." The expression may not contain single quotation marks or exclamation points. It may contain spaces or semicolons for readability.

The expression designates the data item whose memory address is the value of the pointer data item. A subset of the set of allowable address constants is described later. Comments may not appear within the expression. The expression may contain a maximum of 30 characters, not including spaces and semicolons. For example:

Valid	Invalid
ADDR(SAM)	'SAM'
ADDR(SAM + 1)	ADDR((SAM + 1)

Restrictions

Except for the following restrictions, the statement format is unrestrained for records:

. A label and its colon must appear on the same record with no intervening characters.

. The iteration clause of a "DO" statement must appear on the record containing the "DO." At least one space must precede and follow the "TO" in the iteration clause.

. An embedded assembly language statement must be on one record and be the only statement on that record. Its dollar sign must be the first character in the "<sequence of characters>" of its record.

. The dollar sign of a hexadecimal constant may not be the first character in the "<sequence of characters>" of its record.

. No symbol (identifier, constant, or two-character symbol) may continue from one record to the next.

CHAPTER 3. PROCEDURES

A program is organized as a main procedure, a sequence of statements bracketed by a "PROCEDURE OPTIONS(MAIN)" statement and an "END" statement. The compilation, assembly, and linking-load process converts main procedures into free-standing programs.

This section describes main and subroutine procedures and their invocation and return.

Main Procedures

The main procedure is the portion of a program that initially has control. It has no parameters. It may not contain a "RETURN" statement and may not be called by a "CALL" statement. The main procedure may call procedures which in turn may call others and so forth.

MPL allocates space for statements of a program in "PSCT." Variable data may be allocated storage in BSCT, CSCT, DSCT, or PSCT as desired with a default of DSCT. The stack used by a program is allocated space at the end of the main procedure's "DSCT." Storage for temporary results is allocated in a labelled common block in "DSCT."

PROCEDURE OPTIONS(MAIN) statement

The "PROCEDURE OPTIONS(MAIN)" statement has one of the four forms:

```
. label constant: PROCEDURE OPTIONS(MAIN)
. label constant: PROC OPTIONS(MAIN)
. label constant: PROCEDURE OPTIONS(MAIN,
                            SS = integer constant)
. label constant: PROC OPTIONS(MAIN,
                         SS = integer constant)
```

The first and second forms are equivalent, as are the third and fourth forms. Note that the statement must be labelled.

The "PROCEDURE OPTIONS(MAIN)" statement acts as a "left parenthesis" for the statements in the main procedure. The corresponding "END" statement acts as a "right parenthesis."

The "SS = integer constant" form of the statement specifies the number of bytes allocated to the stack. For example:

```
MNPGM:  PROC OPTIONS(MAIN, SS = 40)
```

specifies a stack size of 40 bytes (decimal). If you omit the "SS" clause, MPL allocates 100 bytes (decimal) for the stack.

Subroutine Procedures

A subroutine procedure is a sequence of statements performing a specific task. The procedure has a name so that the program may "call" it. A subroutine may have parameters. Parameters allow it to perform the same task but with different data. It is called by a "CALL" statement, giving its name and the data (if any) it uses. It must contain a "RETURN" statement to return control to the calling procedure. Procedures may call other procedures, but they may not cause themselves to be called recursively unless performing required data stacking.

A subroutine procedure may be compiled during the same time that procedures call it. It may also be compiled separately and combined with the calling procedures by the Linking Loader. Since all variables are global, all procedures in one compilation have a common access to variables declared in any procedure in the compilation.

PROCEDURE statement

The "PROCEDURE" statement has one of the following four forms:

```
. label constant:  PROCEDURE (fp1, fp2, ..., fpn)
. label constant:  PROC (fp1, fp2, ..., fpn)
. label constant:  PROCEDURE <fp1, fp2, fp3>
. label constant:  PROC <fp1, fp2, fp3>
```

The first and second forms and the third and fourth forms are equivalent. The lists of formal parameters in parentheses and angle brackets are optional. The statement must have a label.

The "PROCEDURE" statement acts as a "left parenthesis" for the statements in the subroutine procedure. The corresponding "END" statement acts as a "right parenthesis."

The statement label is the name the procedure is called by a "CALL" statement. The list of formal parameters is a list of unqualified nonsubscripted variables. These variables, like any others, are declared following the "PROCEDURE" statement.

The idea here is similar to the definition of functions in mathematics. A function "f" may be defined by "f(x) = 2x," where "x" has no meaning other than as a placeholder or dummy argument. References to "f(2)" or "f(3)" implies a temporary association of 2 or 3, respectively, with the dummy "x" during the computation of "2x." In this same way, the formal parameters of a procedure are dummies. A procedure with formal parameters is called by a "CALL" statement with actual parameters, whose values are temporarily associated with the formal parameters.

For example, the MPL procedure implements the "f(x)" function by:

```
F:  PROC(XX, RESULT)
    DCL XX BIN(2), RESULT BIN(2)
    RESULT = XX + XX
    RETURN
    END
```

The calls:

```
CALL F(2, RES1)
CALL F(3, RES2)
```

results in "RES1" having the value of 4 and "RES2" having the value of 6.

Two procedures in the same compilation may not have the same formal parameters. A procedure may not attain the formal parameters of another procedure.

Within a procedure, the formal parameters may not be used as subscripts, actual parameters of parentheses "CALLS," or result variables. They may not appear in computed "GOTO" statements or "DO" statements.

The angle-brackets form of the "PROCEDURE" statement may be used when the formal parameters satisfy certain data type restrictions. It generates a more efficient calling sequence. In this form, any of the three formal parameters may be omitted, but all commas prior to the last included parameter must appear. For instance, the first and third parameters may be omitted by:

```
     P:  PROC <, fp2>
but not:
     P:  PROC <fp2, >
```

All three parameters must be scalar variables. The first two must have size 1 (such as SIGNED BIN(1) or CHAR(1)) and the third must have size 2 (such as CHAR(2) or LABEL).

Invocation

Subroutine procedures are called by the "CALL" statement, which has one of the following two forms:

```
     . CALL label constant (apl, ap2, ..., apn)
     . CALL label constant <apl, ap2, ap3>
         GIVING <rl, r2, r3>
```

The first form calls procedures beginning with the first and second form of the "PROCEDURE" statement. The second form calls procedures beginning with the third and fourth form of the "PROCEDURE" statement. The label constant is the name of the called procedure. The parameters in parentheses and angle brackets may be constants or variables. Variables in the first form may not be formal parameters and may not be qualified or subscripted.

The parameters must agree in number, order, and type with the formal parameters specified in the "PROCEDURE" statement. This means that if any or all of the formal parameters are omitted, the corresponding actual parameters should be omitted.

In the second form, "GIVING <rl, r2, r3>" should only appear if the procedure returns by the angle-brackets form of the "RETURN" statement. In this case, rl, r2, and r3 must be unqualified nonsubscripted variables that are not formal parameters. They must agree in the same ways with the result parameters in the "RETURN" statement.

The effect of the "CALL" statement is to execute the named procedure, associating the actual parameters in the call with the formal parameters in the procedure. When the procedure terminates by a "RETURN" statement, control goes to the statement that follows the "CALL" statement.

With the first form, if the procedure has modified the values of any of its formal parameters, the values of the corresponding actual parameters (which in this case must be variables) are changed accordingly. With the second form, if the procedure executes the angle-brackets form of the "RETURN" statement, the values returned are assigned to the result variables.

Return

Subroutine procedures return control to their callers by a "RETURN" statement. The "RETURN" statement has one of the following two forms:

- RETURN
- RETURN <rp1, rp2, rp3>

The result parameters in the second form may be constants or variables. The same data type restrictions and rules for omission apply as in the angle-brackets form of the "PROCEDURE" statement.

The effect of the "RETURN" statement is that it returns control to the calling procedure at the statement following the invoking "CALL" statement. The second form specifies that the current values of the result parameters are to be returned for assignment to the result variables in the "CALL" statement.

CHAPTER 4. DECLARE STATEMENTS

When variables are needed in assignment statements, "CALL" statements, etc., "DECLARE" statements must be used to declare them to the compiler. "Declaring variables means telling the compiler what attributes the data item possess. All variables defined in one compilation are global to that compilation.

This section describes how to use the "DECLARE" statement for different kinds of variables. It first gives a description of the different types of variables. Then it shows the additional features needed to declare array variables, structure variables, and to specify the Linking Loader section in which variables are allotted storage.

Variables

Variable data items are data items whose values may change when a program executes. They are represented in the program text by character strings called "variables." A variable "refers to" the data item it represents. The phrase, "the value of the variable," may be used to mean "the value of the data item referred to by the variable." The phrase, "the attributes of the variable," may be used to mean "the attributes of the data item referred to by the variable."

Variable Attributes

A variable data item may have certain attributes that a constant data item may not have. For instance, the data item may be located in the base section rather than the data section. Since names are used to refer to variable data items, a variable's attributes must be declared to the compiler. This is done for each variable before the first usage of that variable.

In addition to its type attributes, each variable may have combinations of the following attributes: DEFINED, INITIAL, GLOBAL, EXTERNAL, section, BASED, scalar, array, and structure.

A variable can be allocated storage at an absolute memory address, or at the same memory address where another variable has storage. This variable has the "DEFINED" attribute. A variable may also be assigned a certain initial value before executing the program. This variable has the "INITIAL" attribute. The "GLOBAL" attribute specifies that a variable is obtainable in separately compiled MPL programs.

A variable that is declared with the "GLOBAL" attribute can be subsequently referenced in a separately compiled MPL program using the "EXTERNAL" attribute. Variables can be allotted storage in the data section, the base section, blank common, or the program section. This is done by declaring the group of variables to have one of the section attributes: DSCT, BSCT, CSCT, or PSCT.

The "BASED" attribute refers to constructs of a data-item template used as pointer variables. The template is a map or pattern describing a fictitious variable data item with certain attributes. With MPL, the programmer obtains a real variable data item with a memory address the value of a pointer variable, as if it were a variable data item matching the template.

Such a template is not really a variable, since a variable refers to a defined memory address. For language consistency, the template is called a variable having the "BASED" attribute.

A "BASED" variable describing a fictitious variable data item, not contained in a fictitious array, represents a real variable data item when qualified by a pointer variable. MPL does this qualification through the use of an arrow. For example, if "P" is a pointer variable and "T" is a "BIN(1) BASED" variable (template), then the "BIN(1)" variable data item with an address value of "P," is referenced by:

```
P -> T
```

A data item that does not have the array or structure attribute has the scalar attribute and is called a scalar. MPL allows the programmer to organize data into a collection of data items having the same attributes. This collection is called a data item with the array attribute, or an array.

An array is a named 1-, 2-, or 3-dimensional collection of unnamed similar data items. A name is given to the array as a whole. An individual data item in the array is referred to by a variable containing a subscript reference. This indicates the position of the data item with respect to the start of the array. The size of each of the 1-, 2-, or 3-dimensions of the array is specified when its array variable is declared.

The data items collected to form the array may be scalars or structures, but the number of subscripts must never exceed 3.

MPL allows the programmer to organize data into a collection of data items with different attributes. This collection is called a data item, with the structure attribute, or a structure.

A structure is a collection of named, dissimilar data items. A name is given to the structure and to the scalars, the arrays, and the structures comprising the structure. The result is a hierarchical collection. The names and attributes of the data items that form the structure are a part of the declaration of its structure variable. The structure, as a whole, has the "CHAR" attribute.

The data items forming the structure may be scalars, arrays, or structures. The maximum nesting level of structures must not exceed 5. The length of a structure must not exceed 327 bytes.

Attribute Restrictions

There are additional rules and restrictions on the attributes of structure variables. They are listed below:

. Type attribute

Variables that refer to composite data items may not have type attributes.

Successive "BIT(m)" data items within a structure, are packed into bytes as long as byte boundaries are not crossed. Whenever the packing results in byte-boundary crossing, the next byte is used.

. INITIAL attribute

Variables that refer to composite data items may not have "INITIAL" attributes.

. BASED attribute

Level-k variables, where "k > 1," may not be declared with the "BASED" attribute. Level-1 variables may be declared with the "BASED" attribute, in which case the entire structure is "BASED." For example:

```
DCL 1 BSTR BASED,
      2 BASED1 BIN(2),
      2 BASED2 CHAR(5)
```

declares BSTR to refer to a "BASED" structure with a "BIN(2)" component and a "CHAR(5)" component.

. DEFINED attribute

Remarks analogous to those made above for the "BASED" attribute apply to the hexadecimal constant form of the "DEFINED" attribute. For the variable form, the rule is that a level-k variable, where "k >1," may only be "DEFINED" to a previously declared "brother" within the structure. No other kind of variable may be "DEFINED" to level-k variables where "k > 1." For example, this declaration is valid:

```
DCL 1 S,
      2 ABC,
      2 DEFF CHAR(5),
      2 GHI CHAR(1) DEF ABC
```

This one is invalid:

```
DCL 1 BAD
      2 BR1,
        3 ABC,
      2 BR2,
        3 GHI CHAR(1) DEF ABC
```

This rule may be stated more precisely as follows: two level-k variables related by "DEFINED," where "k > 1," must refer to data items contained in the same level-(k - 1) data item.

. EXTERNAL and GLOBAL attributes

Remarks analogous to those made above for the "BASED" attribute apply as well to the "EXTERNAL" and "GLOBAL" attributes.

. Array attribute

MPL allows you to describe and refer to arrays of composite data items, as well as arrays of elementary data items. This means that you may construct an array of four structures, each containing an array of five "BIN(2)" data items and an array of nine "DEC(5, 2)" data items. There can never be more than a total of three dimensions in a structure, no matter how many levels the dimensions are distributed. For example, a variable referring to the structure array just described is:

```
DCL 1 STRARR(4),
      2 BINARR(5) BIN(2),
      2 DECARR(9) DEC(5, 2)
```

As many subscripts as necessary to completely specify data items should be used. For example, you would refer to the fourth "DEC(5, 2)" data item in the third structure by:

```
DECARR(3, 4)
```

Declaring Simple Variables

The "DECLARE" statement begins with one of the reserved words, DECLARE or DCL. Following the reserved word, a list of variable declarations, separated by commas, is written.

Each variable declaration begins with the variable itself and ends with the list of attributes of the data item. The allowed forms of the attribute list depend on whether the variable refers to "LABEL" data. The next two sections describe the allowed forms.

LABEL variables

The attribute list of a "LABEL" variable may have one of the three forms:

- LABEL
- LABEL INITIAL(label constant)
- LABEL INIT(label constant)

The second and third forms are the same. "INIT" is an acceptable abbreviation for "INITIAL." If the second form is used, the "LABEL" variable is contained within the parentheses. For example, in a program containing a statement labelled, "LETTER," you may declare L1 and L2 to be "LABEL" variables with L1 initialized to "LETTER" by the statement:

```
DCL L1 LABEL INIT(LETTER), L2 LABEL
```

The initial value of L2 in this example is undefined.

Non-LABEL variables

The attribute list of a non-LABEL variable begins with a type attribute. If this type attribute is omitted, the variable assumes type "BIN(1)." A type attribute begins with one of the type designators: BIN, SIGNED BIN, DEC, SIGNED DEC, CHAR, or BIT.

Any one of these may be followed by a size specification of the form (integer constant). "DEC" and "SIGNED DEC" may be followed by a size specification of the form (integer constant, integer constant). If the size specification is omitted, "(1)" is assumed. Alternatively, a type attribute may be the single word, "SIGNED," in which case the variable is assumed to be of type "SIGNED BIN(1)."

For example, if B1 is declared a "BIN(1)" variable, SB1 a "SIGNED BIN(1)" variable, and BIT4 a "BIT(4)" variable by the statement:

```
DCL B1, SB1 SIGNED, BIT4 BIT(4)
```

Following the type attribute, the attribute list concludes with any of the following six mutually exclusive attributes.

INITIAL(constant) (or INIT(constant))

This gives the declared variable an initial value of the constant contained within the parentheses. The constant should be compatible with the type of the variable. "BIT" variables may not be initialized by this attribute. For example:

```
DCL B1 INIT(25)
DCL STRING CHAR(8) INIT('HI THERE')
DCL N DEC(5, 2) INIT(2.69)
```

. BASED

This states that the variable is a "BASED" variable. It is a template or pattern describing a fictional data item. It is not a variable. For example:

```
DCL P DEC(6) BASED
DCL B1 BASED
```

. DEFINED variable (or DEF variable)

This causes the declared variable to be allocated storage at the same memory address where a previously declared variable has storage. A variable data item is then considered a different type. For example:

```
DCL B1, C1 CHAR(1) DEF B1
```

. DEFINED hexadecimal constant (or DEF hexadecimal constant)

This causes the declared variable to have storage at the absolute memory address indicated by the hexadecimal constant. For example:

```
DCL ACIA BIN(2) DEF $EC14
```

. EXTERNAL

This states that the variable is to have or has allocated storage in a separately compiled MPL program. The variable must be declared with the "GLOBAL" attribute in the separately compiled program. For example:

```
DCL REV DEC(4, 2) EXTERNAL
```

. GLOBAL

This states that separately compiled MPL programs execute this variable by declaring it to have the "EXTERNAL" attribute. For example:

```
DCL REV DEC(4, 2) GLOBAL INIT(2.01), B1 GLOBAL
```

Declaring Array Variables

A variable is referred to an array by writing a dimension designator immediately following the "DECLARE" statement. This dimension designator is a parenthesized list of no more than three integer constants separated by commas. Each integer constant must denote a value greater than 1. The number of integer constants in the list is the number of dimensions in the array. The minimum subscript used for the kth dimension, when referring to an array element, is 1. The maximum is the kth integer constant in the list of the "DECLARE" statement. For example:

```
DCL ARRAY(2, 3) CHAR(5)
```

declares an array variable, "ARRAY," consisting of the six "CHAR(5)" elements: ARRAY(1, 1), ARRAY(1, 2), ARRAY(1, 3), ARRAY(2, 1), ARRAY(2, 2), and ARRAY(2, 3). The "INITIAL" attribute for an array variable may take one of the following two forms:

. INITIAL(constant, constant, ..., constant)
. INIT(constant, constant, ..., constant)

where the number of constants in the parenthesized list is less than or equal to the product of the integer constants in the variable's dimension designator. This assigns the values of the constants to the array elements as initial values. Array elements match to constants in the order described by the phrase, "last subscript varies most rapidly." For example:

```
DCL BB(2, 2) INIT(1, 2, 3, 4)
```

assigns the initial values:

```
1 to BB(1, 1),
2 to BB(1, 2),
3 to BB(2, 1), and
4 to BB(2, 2).
```

Declaring Structure Variables

In order for a variable to refer to a structure or component of a structure, a level number is written immediately before the variable in the "DECLARE" statement. This level number is an integer constant denoting a value from 1 to 5, inclusive. All components of a structure variable are declared in order, in a single "DECLARE" statement.

A structure is a level-1 data item. It is composed of a number of level-2 data items. A level-2 data item may itself have the form of a structure, in which case it is composed of a number of level-3 data items, and so on, up to a level of 5. Levels may not be skipped in a "DECLARE" statement. For example, a level-2 declaration may not be followed by a level-4 declaration without an intervening level-3 declaration.

Structure components that have the form of structures are called "composite data items." Structure components that do not have the form of structure are called "elementary data items."

For example, consider the declaration:

```
DCL 1 L1,
      2 L2A,
        3 L3A CHAR(1),
        3 L3B BIN(2),
      2 L2B LABEL
```

This example declares the structure variable L1. L1 refers to a structure containing two level-2 data items, which are referred to by the two level-2 variables, L2A and L2B. The first level-2 data item is further composed of two level-3 data items, which are referred to by the "CHAR(1)" variable, L3A and the "BIN(2)" variable L3B. L2B is a "LABEL" variable. L1 and L2A refer to composite data items. L3A, L3B, and L2B refer to elementary data items.

Constant Interpretation

Contexts

The MPL compiler determines the attributes and the value of a constant from its textual form and its context. The context of a constant is a list of type attributes dependent on the constant's textual position within an MPL program. This section lists the thirteen possible places where constants appear. The classes of constants and the context are given. The next two sections discuss attribute and value determination.

This list refers to MPL statement types. It is useful primarily as a reference aid.

. Expressions (allows integer, decimal, hexadecimal, character-string, address)

If a constant within an expression is not the first operand, its context is the list of type attributes of the previous operand. If it is the first operand and the expression is on the right-hand-side of an assignment statement, then its context is the list of type attributes of the rightmost variable, on the left-hand-side of the assignment statement. In no other case is a constant the first operand in an expression.

. Structure level number (allows integer)

The context of a structure level number is "BIN(1)."

. Dimension (allows integer)

The context of a dimension is "BIN(1)."

. Subscript (allows integer)

The context of a constant within a subscript is "BIN(1)."

. Precision attribute specification (allows integer)

The context of constants within a precision attribute specification is "BIN(1)."

. DEFINED attribute specification (allows hexadecimal)

The context of an absolute memory address in a "DEFINED" attribute specification is "BIN(2)."

. Iterative DO (allows integer, decimal, hexadecimal, character-string, address)

The context of a constant used as the initial value of an iterative "DO" is the list of type attributes of the index variable of the "DO."

The context of a constant used as the final value of an iterative "DO" is the list of type attributes of the initial value of the "DO."

The context of a constant used as the increment of an iterative "DO" is X(1), where "X" is the list of base and sign attributes of the final value of the "DO."

. GOTO statement (allows integer)

The context of a constant in a "GOTO" statement is "LABEL."

. Label list of computed "GOTO" statement (allows label)

The context of a constant within the label list of a computed "GOTO" statement is "LABEL."

. Procedure name in "CALL" statement (allows label)

The context of a constant used as the procedure name in a "CALL" statement is "LABEL."

. INITIAL attribute specification (allows integer, decimal, hexadecimal, character-string, label, address)

The context of a constant within an "INITIAL" attribute specification is the list of type attributes of the variable given the "INITIAL" attribute.

. Argument list of CALL (allows integer, decimal, hexadecimal, character-string, address)

The context of integer, decimal, hexadecimal, and address constants within the argument list of a parentheses "CALL" is "BIN(2)." The context of a character-string constant within the argument list of a parentheses "CALL" is "CHAR(m)." "m" is the number of characters in the character string denoted by the constant.

The context of constants appearing as the first two arguments of an angle-brackets "CALL" is "BIN(1)." The context of a constant appearing as the third argument of an angle-brackets "CALL" is "BIN(2)."

. Argument list of RETURN (allows integer, decimal, hexadecimal, character-string, address)

The context of constants appearing as the first two arguments of a "RETURN" is "BIN(1)." The context of a constant appearing as the third argument of a "RETURN" is "BIN(2)."

Attribute Determination

This section gives the dependence of attributes upon context for each class of constant.

In the following, reference is made to the "apparent value" of a constant. For example: if "SB1" is a "SIGNED BIN(1)" variable, then in the statement "SB1 = 255," the constant, 255, has a "SIGNED BIN(1)" context. Its apparent value is 255, but its value is -1, since its type attributes are "SIGNED BIN(1)."

Integer constants, decimal constants

. [SIGNED] BIN(1) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(1)." Otherwise, its type attributes are "[SIGNED] BIN(1)" or "[SIGNED] BIN(2)," depending on whether its apparent value is less than 256.

. [SIGNED] BIN(2) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(2)." Otherwise, its type attributes are "[SIGNED] BIN(2)."

. [SIGNED] DEC(m, n) context

The constant's type attributes are "[SIGNED] DEC(m, n)."

. CHAR(m) context; m <= 2

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(m)." Otherwise, its type attributes are "CHAR(m)."

. CHAR(m) context; m > 2

The constant's type attributes are "CHAR(m)."

. BIT(m) context

The constant's type attributes are "BIN(1)."

. LABEL context

The constant's type attributes are "BIN(2)."

Hexadecimal constants

. [SIGNED] BIN(1) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(1)." Otherwise, its type attributes are "[SIGNED] BIN(1)" or "[SIGNED] BIN(2)," depending on whether its apparent value is less than 256.

. [SIGNED] BIN(2) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(1)." Otherwise, its type attributes are "[SIGNED] BIN(2)."

. CHAR(1) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(1)." Otherwise, its type attributes are "CHAR(1)" or "CHAR(2)," depending on whether its apparent value is less than 256.

. CHAR(2) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(2)." Otherwise, its type attributes are "CHAR(2)."

. BIT(m) context

The constant's type attributes are "BIN(1)."

. LABEL context

The constant's type attributes are "BIN(2)."

Character-string constants

. [SIGNED] BIN(m) context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "CHAR(m)." Otherwise, its type attributes are "[SIGNED] BIN(m)."

. CHAR(m) context

The constant's type attributes are "CHAR(m)."

. BIT(m) context

The constant's type attributes are "BIN(1)."

. LABEL context

If the constant appears in an "INITIAL" attribute specification, its type attributes are "CHAR(2)." Otherwise, its type attributes are "BIN(2)."

Label constants

If the constant appears in an "INITIAL" attribute specification, its type attributes are "BIN(2)." Otherwise, its type attribute is "LABEL."

Address constants

. [SIGNED] BIN(m) context

The constant's type attributes are "[SIGNED] BIN(2)."

. LABEL context

The constant's type attributes are "BIN(2)."

Value Determination

This section gives the dependence of value upon attributes for each class of constants. In the following, the apparent value of a constant is denoted by "A."

Integer constants, decimal constants:

- BIN(1)

  A < 256: value = integer part of A.
  A >= 256: value is undefined.

- SIGNED BIN(1)

  A < 128: value = integer part of A.
  128 <= A < 256: value = (integer part of A) - 256.
  A >= 256: value is undefined.

- BIN(2)

  A < 65536: value = integer part of A.
  A >= 65536: value is undefined.

- SIGNED BIN(2)

  A < 32768: value = integer part of A.
  32768 <= A < 65536: value = (integer part of A) - 65536.
  A >= 65536: value is undefined.

- DEC(m, n)

  A < 10**(m - n): value = (integer part of (A*10**n))/10**n.
  A >= 10**(m - n): value is undefined.

- SIGNED DEC(m, n)

  A < 10**(m - n - 1): value = (integer part of (A*10**n))/10**n.
  A >= 10**(m - n - 1): value is undefined.

- CHAR(m)

  "A" is converted to a character string. It is truncated on the right or right justified and space filled, if required, on the left. If the constant is the character string, "c1c2...cn," (including leading zeros) then the value is given by the rule:

  m >= n: value = ' c1c2...cn' (m - n spaces).
  m < n: value = 'c1c2...cm'.

Hexadecimal constants:

. BIN(1)

  A < 256: value = A.
  A >= 256: value is undefined.

. SIGNED BIN(1)

  A < 128: value = A.
  128 <= A < 256: value = A - 256.
  A >= 256: value is undefined.

. BIN(2)

  A < 65536: value = A.
  A >= 65536: value is undefined.

. SIGNED BIN(2)

  A < 32768: value = A.
  32768 <= A < 65536: value = A - 65536.
  A >= 65536: value is undefined.

. CHAR(1)

  A < 128: value = the character string of length 1 containing the single character whose ASCII code is A.
  A >= 128: value is undefined.

. CHAR(2)

  Let a1 = integer part of (A/256) and a2 = A mod 256.
  a1 < 128, a2 < 128: value = the character string of length 2 whose first character has ASCII code a1 and whose second character ASCII code a2.
  a1 >= 128 or a2 >= 128: value is undefined.

Character-string constants:

. BIN(1)

  Length of A = 1 (A = 'c'): value = ASCII code of c.
  Length of A > 1: value is undefined.

- SIGNED BIN(1)

  Length of A = 1 (A = 'c') and (ASCII code of c) < 128: value = ASCII code of c.
  Length of A = 1 (A = 'c') and (ASCII code of c) >= 128: value = (ASCII code of c) - 256.
  Length of A > 1: value is undefined.

- BIN(2)

  Length of A = 1 (A = 'c'): Compute v = (ASCII code of c)*256 + (ASCII code of space). Value = v.
  Length of A = 2 (A = 'c1c2'): Compute v = (ASCII code of c1)*256 + (ASCII code of c2). Value = v.
  Length of A > 2: value is undefined.

- SIGNED BIN(2)

  Length of A <= 2: Compute v as in BIN(2) case above. If v < 32768, then value = v. If v >= 32768, then value = v - 65536.
  Length of A > 2: value is undefined.

- CHAR(m)

  "A" is left justified and if required, space filled on the right. In other words, if A = 'c1c2...cn', then:

  m >= n: value = 'c1c2...cn ' (m - n spaces).
  m < n: value is undefined.

Label constants:

- BIN(2)

  Value = a number equal to the runtime address of the statement labelled by the constant.

- LABEL

  Value = the address of the statement labelled by the constant.

Address constants:

The value of "A" of an address constant is determined from the Macroassembler expression. The expression may consist of an unqualified nonsubscripted variable, optionally followed by a plus sign, followed by an integer constant. "A" is equal to the memory address of the data item referred to by the variable plus the apparent value of the constant.

- BIN(2)

  Value = A.

- SIGNED BIN(2)

  A < 32768: value = A.
  A >= 32768: value = A - 65536.

Section Specification

A group of variables may be allocated storage in any of the Linking Loader sections: BSCT (base section), CSCT (blank common section), DSCT (data section), or PSCT (program section). This is done by declaring the group in a single "DECLARE" statement where the word "DECLARE" (or DCL) is immediately followed by the reserved word, BSCT, CSCT, DSCT, or PSCT. If a section in a "DECLARE" statement is not specified, the compiler allocates storage for the variables in "DSCT."

For example, AA, BB, and CC may be declared in blank common by:

```
DCL CSCT AA, BB, CC
```

Variables declared in "CSCT" should not have the INITIAL, BASED, EXTERNAL, or GLOBAL attributes.

The use of the hexadecimal constant form of the "DEFINED" attribute, in the declaration of a variable, overrides any express or implied section specification. Variables so declared are allocated storage in "ASCT" (absolute section).

The compiler allocates space for the executable statements of the MPL program in "PSCT." This means that you may have to precede "PSCT" declarations by "GOTO" statements to transfer control, depending on the exact placement of the declarations. For example:

```
START:
  PROC OPTIONS(MAIN)
  GOTO ENTRY
  DCL PSCT AA, BB
ENTRY:
  (rest of program)
```

CHAPTER 5. EXPRESSIONS

Arithmetic Expressions

Calculations performed in an MPL program are specified by arithmetic expressions. Arithmetic expressions consist of arithmetic operands combined by arithmetic operators.

Arithmetic Operands

Arithmetic operands may be constants or variables of any of the BIN, SIGNED BIN, DEC, SIGNED DEC, or CHAR types or variables of types BIT or LABEL. They may also be arithmetic expressions as defined in this section. Here are some arithmetic operands:

```
ADDR(SAM)
-2.06
((SBIN1 + SBIN2)*SBIN2)
SBIN1A(BIN1 + 2)
PTR->DEC52
```

Arithmetic Operators

The arithmetic operators provided in MPL are the following:

Arithmetic Operator	Indicated Operation
+	Addition
-	Subtraction
- (unary minus)	Negation
*	Multiplication
/	Division
SHIFT or %	Rotate or arithmetic shift, depending on context
IAND or &	Bitwise AND
IOR	Bitwise OR
IEOR	Bitwise EXCLUSIVE OR

Each arithmetic operator except for unary minus, takes two arithmetic operands. Unary minus takes only one arithmetic operand. The two arithmetic operands must both either have the BIN, DEC, or CHAR attribute. This is usually described by: "MPL does not allow mixed-mode expressions." They may differ as to the "SIGNED" attribute and the size. For example, "SBIN1 + BIN2" is a legal combination, but "BIN1 + CHAR1" is not.

The arithmetic operands of each arithmetic operator are subject to further data type restrictions. The following table shows the data types to which each of the arithmetic operators are applied:

Arithmetic Operator	Allowed Types of Arithmetic Operands
+,-	CHAR(1), BIN, SIGNED BIN, DEC(m, n), SIGNED DEC (m, n), m <> 2
- (unary minus)	SIGNED BIN, numeric constants
*, /	SIGNED BIN, DEC(m, n), SIGNED DEC(m, n), m > 2
SHIFT or %	SIGNED BIN, CHAR(1)
IAND or &, IOR	BIN, SIGNED BIN, CHAR(1)
IEOR	Same as IOR, but different sizes not allowed

More needs to be said about the two arithmetic operators, "*" and "%." First, multiplication must be explicitly specified. "RS" is not the same as "R*S." Second, the second arithmetic operand of "%" must be a non-zero integer or hexadecimal constant, possibly preceded by a minus sign. The value of the minus sign indicates the number of bit positions the first arithmetic operand rotates or shifts.

If the value of the second arithmetic operand is positive, it rotates or shifts to the left (toward the most significant bit). If it is negative, it rotates to the right (toward the least significant bit). "SIGNED BIN" arithmetic operands shift arithmetically. "CHAR(1)" arithmetic operands rotate.

Because of the rules on constant interpretation, integer constants used as rotate counts are interpreted as having type "CHAR(1)." Thus, hexadecimal constants should be used. For example, if "SBIN1" has the value -4 and "CHAR1" has the value "E'," then "SBIN1 % -1" has the value -2 and "CHAR1 % $4" has the value "'T'."

No two arithmetic operators may be adjacent except in the case of unary minus: "W*-Y" is allowed and equivalent to "W*(-Y)."

Order of Evaluation

High-precedence arithmetic operators are applied before low-precedence arithmetic operators. Equal-precedence arithmetic operators are applied from left to right. This is always the case unless the order changes through the use of parentheses. Unary minus has the highest precedence, as shown in this table:

Arithmetic Operator	Precedence
- (unary minus)	High
SHIFT or %	
IAND or &, IOR, IEOR	
*, /	
+, -	Low

Parentheses are used to override this ordering. For example, in the arithmetic expression, "V*Y + Z*W&I," first "W" and "I" are "ANDed." Then "V" and "Y" are multiplied. The result of "W" and "I" being "ANDed" is multiplied by "Z." That product is added to the product of "V" and "Y."

In the arithmetic expression, "B + ((C + D)*E) + C&2," first "C" and "D" are added, and the result is multiplied by "E." Then "B" is added to that product, "C" and "2" are "ANDed," and the sum of "B" and the product are added to that.

Use of Arithmetic Expressions

Arithmetic expressions are used in two places in an MPL program: assignment statements and logical expressions. When an arithmetic expression appears in an assignment statement, this means that the arithmetic expression is evaluated, and the assigned value is the current value of one or more variables. An arithmetic expression in a logical expression means that the arithmetic expression is evaluated, and its value compared with the value of some other arithmetic expression.

Logical Expressions

Logical expressions specify the conditions under which certain statements in an MPL program execute or repeat. Logical expressions consist of logical operands combined by logical operators using the ordinary rules of Boolean algebra. For example, the logical expression "(C < D OR E < F)" is true if the value of "C" is less than the value of "D," or if the value of "E" is less than the value of "F."

Logical Operands

A logical operand is a pair of arithmetic expressions separated by one of the relational operators: EQ, NE, LT (or <), GT (or >), GE, or LE. The relational operators indicate comparisons for equal, not equal, less than, greater than, greater than or equal, and less than or equal, respectively. The values of the two arithmetic expressions may not be of mixed modes. Both values must have the "BIN," "DEC," or the "CHAR" attribute, although they may differ as to the "SIGNED" attribute and the size.

If the relational operator is "EQ" or "NE," there are two other allowable combinations:

. Both arithmetic expressions may be "LABEL" variables.

. The first arithmetic expression may be a "BIT(m)" variable, and the second arithmetic expression may be the integer constant 0. If "m = 1," the second arithmetic expression may be the integer constant 1.

. A logical operand may also be a logical expression, as defined in this section.

Logical Operators

The logical operators used in MPL are the following:

Logical Operator	Result
NOT	True if logical operand is false
AND	True if both logical operands are true
OR	True if either logical operand is true

Order of Evaluation

High-precedence logical operators are applied before low-precedence logical operators, and equal- precedence logical operators are applied from left to right, unless parentheses are used. "NOT" has the highest precedence.

Logical Operator	Precedence
NOT	High
AND	
OR	Low

Parentheses are used to override this ordering. For example, the logical expression, "(C EQ D OR E EQ F AND G EQ H)," is true either if the current values of "C" and "D" are equal, or if the current values of "E" and "F" are equal, and the current values of "G" and "H" are equal. The logical expression "((C EQ D OR E EQ F) AND G EQ H)" is true if the current values of "G" and "H" are equal, and either the current values of "C" and "D" are equal, or the current values of "E" and "F" are equal.

Use of Logical Expressions

Logical expressions are used in two places in an MPL program: "IF" statements and "DO" statements containing "WHILE" clauses. A logical expression in an "IF" statement means that one of two sequences of statements is executed, depending on the truth or falsity of the logical expression. A logical expression in a "DO" statement, containing a "WHILE," clause means that a sequence of statements is repeatedly executed, as long as the logical expression is true.

CHAPTER 6. STATEMENTS

Assignment Statements

The assignment statement is MPL's chief way of modifying values of variables. It allows the values of one or more variables to change the value of some arithmetic expression. This section discusses the assignment statement's format, its effect, some rules regarding conversions, and some examples.

Format

The format of the assignment statement is:

<variable list> = <arithmetic expression>

where "<variable list>" is a list of variables separated by commas, and "<arithmetic expression>" is an arithmetic expression. The "<variable list>" usually consists of a single variable. A variable in the "<variable list>" may not have the array attribute. The types of the variables and the arithmetic expression must be related by the rules described below.

Effect

The arithmetic expression is evaluated and converted, if necessary, to the type of the rightmost variable. It is then assigned the value of that variable. That value is then assigned the value of the next variable to the left, after any necessary conversion. This process continues until all variables in the list are assigned the value.

Implicit Conversions

Some combinations are not allowed in assignments. Others cause the conversion of data from one type to another. This section outlines the rules regarding these conversions.

. "CHAR(m)" values may be assigned only to "CHAR(n)" variables. If "m" and "n" are equal, no conversion is performed. If "m" is less than "n," the value is extended on the right with "n-m" ASCII space characters before assignment. If "m" is greater than "n," the rightmost "m-n" characters of the value are discarded before assignment.

. Either "BIN" or "DEC" values may be assigned to "BIN" variables. In the case of "BIN" values, a conversion is not performed. If the value is outside the permitted range of values for the variable, the result is undefined.

In the case of "DEC(m, n)" values, the value is first converted to an integer. If the value is a constant, this conversion is a truncation. Otherwise, it is a multiplication by "10**n." If the variable is not "SIGNED," the converted value is replaced by its absolute value. The result is converted to the appropriate "BIN" type and assigned. If it is outside the permitted range of values for the variable, the result is undefined.

. "LABEL" values may be assigned to "LABEL" variables. A conversion is not performed.

. Either "BIN" or "DEC" values may be assigned to "DEC(m, n)" variables. In the case of "BIN" values, the value is first converted to a decimal number. If the value is a constant, this conversion is the obvious one; otherwise it is a division by "10**n." If the variable is not "SIGNED," the converted value is replaced by its absolute value. The result is converted to the appropriate "DEC" type and assigned. If it is outside the permitted range of values for the variable, the result is undefined.

In the case of "DEC" values, fractional digits are dropped. Then if the variable is not "SIGNED," the value is replaced by its absolute value. If it is outside the permitted range of values for the variable, the result is undefined.

. Either "BIN" or "DEC" values may be assigned to "CHAR" variables. In either case, the value is converted to a character string containing its decimal representation with leading zeros replaced by spaces, a leading minus sign inserted if the value was negative, and a decimal point inserted as required for "DEC" values.

. The only arithmetic expression that may be assigned to "BIT(n)" variables is an integer or hexadecimal constant, denoting the value "0" or "2**n-1." These values result in the assignment of an all-0 bit string or an all-1 bit string, respectively.

Examples:

```
DCL C3 CHAR(3), C5 CHAR(5)
DCL BIT4 BIT(4)
DCL D86 DEC(8, 6), D5A DEC(5), D5B DEC(5)
DCL SB1A SIGNED, SB1B SIGNED, SB1C SIGNED
DCL AVE SIGNED
C3 = 'ABC'      /* Gets 'ABC' */
C5 = C3         /* Gets 'ABC  ' */
BIT4 = $F       /* Gets all ones */
D86 = D5A/D5B   /* Gets quotient to six places */
SB1A = SB1A + 1 /* Increments SB1A */
AVE = (SB1A + SB1B + SB1C)/3 /* Computes average */
```

GOTO Statements

MPL permits the alteration of the flow of control in a program. Ordinarily, control flows from one statement to the next. Some MPL statements, like the "IF" statement and the "DO" statement, cause implicit control transfers. "GOTO" statements specify explicit control transfers.

This section describes the three kinds of "GOTO" statements and gives examples of their use.

Label Constant GOTO Statement

The label constant "GOTO" statement has one of the following forms:

. GO TO label constant
. GOTO label constant

The two forms are equivalent. This statement causes a transfer of control to the statement (elsewhere in the program) labelled by the label constant. For example:

```
          .
          .
          .
          GOTO SAM
   SKIP:  I = 1
          .
          .
          .
   SAM:   I = 2
          .
          .
          .
```

If this example is executed, the "GOTO" statement causes the statements beginning with "SKIP" to be disregarded. The statement executed immmediately after the "GOTO" statement is the statement, "I = 2."

LABEL Variable GOTO Statement

The "LABEL" variable "GOTO" statement has one of the following forms:

. GO TO <variable>
. GOTO <variable>

where "<variable>" is a variable referring to a "LABEL" data item. The two forms are equivalent. This causes a transfer of control to the statement (elsewhere in the program) labelled by the current value of the variable. For example:

```
              .
              .
              .
              DCL I, LARRAY(3) LABEL INIT(GOOD, BAD, UGLY)
              .
              .
              .
              I = 2
              GOTO LARRAY(I)
       GOOD:  I = 0
              GOTO OUT
       BAD:   I = 1
              GOTO OUT
       UGLY:  I = -99
              GOTO OUT
              .
              .
              .
```

If this example is executed, the "LABEL" variable "GOTO" statement causes the statement labelled "BAD" to execute, followed by the next "GOTO OUT" statement.

Computed GOTO Statement

The computed "GOTO" statement has one of the following forms:

- GO TO (label constant, ..., label constant), <name>
- GOTO (label constant, ..., label constant), <name>

where "<name>" is an unqualified nonsubscripted "BIN(1)" variable that is not a formal parameter. The two forms are equivalent. This statement causes a transfer of control to the statement (elsewhere in the program) labelled by the first, second, third, ..., or nth label constant, whether the current value of the variable is 1, 2, 3, ..., or n, respectively. If the current value is zero or greater than the number of label constants in the statement, the effect is undefined. For example:

```
GOTO (GOOD, BAD, UGLY), I
```

This statement has the same effect as the statement "GOTO LARRAY(I)" in the previous example.

IF Statements

MPL provides a statement that allows the programmer to specify a course of action based on the truth or falsity of a condition. This statement is the "IF" statement. This section describes the "IF" statement and gives examples of its use.

IF Statement Format

The "IF" statement has one of the following forms:

- IF <logical expression> THEN <statement>
- IF <logical expression> THEN <statement> ELSE <statement>

where "<logical expression>" is a logical expression and the "<statement>s" are executable statements. The effect of the first form is to execute the "<statement>" only if the "<logical expression>" is true. If it is false, control goes to the statement following the "IF" statement. The effect of the second form is that it executes either the first "<statement>" or the second "<statement>," dependent on whether the "<logical expression>" is true or false, respectively. After the appropriate "<statement>" executes, control goes to the statement following the "IF" statement.

To choose between sequences of statements, rather than single statements, the "DO-END" brackets are used. For example:

```
IF I < 25 THEN I = 25
IF (I NE 0 AND I LE 3) THEN GOTO
(GOOD, BAD, UGLY), I
ELSE GOTO ERROR
```

DO Statements

MPL allows you to group a sequence of statements in order to consider them as a single statement. It also allows conditions to be specified so that the sequence repeats. The "DO" statement is provided for these purposes. This section describes the permissible forms of the "DO" statement.

Form

"DO" statements have the general form:

```
DO [<iteration clause>] [<WHILE clause>]
```

This means that a "DO" statement consists of the word "DO," optionally followed by an iteration clause, optionally followed by a "WHILE" clause. psll

If the iteration clause is present, it has the following form:

```
<variable> = <DO operand> TO <DO operand>
                          [BY <DO operand>]
```

where "<variable>" is an unqualified nonsubscripted variable that is not a formal parameter of a procedure and the "<DO operand>s" are either variables or constants meeting the same description. The "BY <DO operand>" is optional. "<variable>" may have type "BIN(1)" or "BIN(2)." The first two "<DO operand>s" must have the same type as "<variable>." The third "<DO operand>" must have type "BIN(1)."

If the "WHILE" clause is present, it has the form:

```
WHILE <logical expression>
```

where "<logical expression>" is a logical expression.

Each "DO" statement in your program must have a corresponding "END" statement. The corresponding "END" statement is found the same way the right parenthesis, corresponding to a given left parenthesis, is found in an expression. For example:

```
  DO
  .
  .
  .
   DO
   .
   .
   .
    DO
   .
   .
   .
    END
  .
  .
  .
  END
.
.
.
END
```

The first "DO" matches the third "END." The second "DO" matches the second "END," and the third "DO" matches the first "END." "DOs" so arranged are called "nested DOs." This example shows three levels of nesting. "DOs" may be nested to a level of ten. Indentation of nested "DOs" is not required, but is suggested for readability.

Simple DO

In its simplest form, the "DO" statement acts as a "left parenthesis" to group statements. The corresponding "END" statement acts as the "right parentheses." For example, to assign values to either I, J, and K or L, M, and N, depending on the sign of Q:

```
IF Q < 0 THEN
  DO
  I = V1
  J = V2
  K = V3
  END
ELSE
  DO
  L = V1
  M = V2
  N = V3
  END
```

DO with Iteration Clause

An iteration clause on a "DO" statement specifies that the sequence of statements, beginning with the one following the "DO" statement and ending with the corresponding "END" statement, is repeated a certain number of times.

```
DO V = D1 TO D2 BY D3
```

The effect of the above "DO" statement is the following. When the statement is encountered, the variable "V" is given the value of "D1" and the sequence of statements executes. The value of "V" is compared with the value of "D2." If it is equal (in the BIN(2) case) or greater or equal (in the BIN(1) case), the process continues with the statement following the "END" statement. Otherwise, "V" increments by the value of "D3," and the sequence of statements executes again. This process repeats until "V" reaches or exceeds "D2."

If "BY D3" is omitted in the above example, "V" would increment by 1 each time.

Note that if the values of V, D1, D2, or D3 change, the looping process may continue in an unpredictable manner.

This example illustrates the zeroing of every element of the array "AA" and every other element of the array "BB":

```
DCL AA(10), BB(10), I
DO I = 1 TO 10
AA(I) = 0
END
DO I = 1 TO 9 BY 2
BB(1) = 0
END
```

DO with WHILE Clause

A "WHILE" clause on a "DO" statement specifies that the sequence of statements, beginning with the one following the "DO" statement and ending with the corresponding "END" statement, repeats if a certain condition is met. This condition is represented by an MPL logical expression.

```
DO WHILE <logical expression>
```

The effect of the preceding "DO" statement is as follows. If the logical expression is false, execution continues with the statement following the "END" statement. Otherwise, the sequence of statements executes and the logical expression is examined again. This process repeats until the logical expression is false.

The following example illustrates the end of a linked list, pointed to by "LIST." The last node in the list assumes a zero link.

```
DCL LIST BIN(2), P BIN(2)
DCL 1 NODE BASED,
      2 VAL,
      2 NEXT BIN(2)
P = LIST
DO WHILE P -> NEXT NE 0
P = P -> NEXT
END
```

DO with Iteration and WHILE Clauses

A "DO" statement may have an iteration clause and a "WHILE" clause. In this case, the sequence of statements executes until the iteration completes or the "WHILE" condition fails, whichever comes first. The sequence of statements in the following example executes from 0 to 5 times, depending on when, if ever, "Y" becomes less than 4.

```
DO I = 1 TO 5 WHILE Y < 4
.
.
.
END
```

END Statements

"END" statements terminate statement groups begun by "DO" or "PROCEDURE" statements. "END" statements have the form:

```
     END [label constant]
```

The label constant is an optional documentation aid. For example:

```
     LOOP:  DO I = 1 TO 50
            .
            .
            .
            END LOOP
```

Embedded Assembly Language

Legal Macroassember Language statements may be included in an MPL program. Such statements are called embedded assembly language statements. In contrast to MPL statements, the format of these statements is not unrestricted with respect to records. An embedded assembly language statement is a record whose "<sequence of characters>" consists of a dollar sign followed by a legal Macroassember Language statement. This assembly language statement inserts into the output assembly language program without further processing by the compiler.

Names (label constants and variables) defined in MPL statements may be used as operands in embedded assembly language statements. Labels defined in embedded assembly language statements may be used in "GOTO" statements.

Embedded assembly language statements are syntactically distinct from MPL statements. The contexts are slightly different. An embedded assembly language statement is like a label because it may precede any MPL statement.

Assembly language statements embedded within the MPL program should make no assumptions regarding the contents of the processor accumulators, index register, program counter, stack pointer, condition codes register, or stack. For example:

```
CODOS:
  PROC
$ NAM CODOS
$ SWI
$ FCB $1A
  RETURN
  END

IF T > 1 THEN
$ PAGE
  DO
   .
   .
   .
  END
ELSE
$ PAGE
  DO
   .
   .
   .
  END
```

CHAPTER 7. ASSEMBLY LANGUAGE ROUTINE LINKAGE

This section gives guidelines on how to write assembly language routines. It covers MPL calling, returning, and parameter passing conventions. This also applies to embedded assembly language statements in an MPL program.

GLOBAL and EXTERNAL Considerations

An assembly language routine may obtain an MPL "GLOBAL" variable by naming the variable in an "XREF" directive. An MPL "EXTERNAL" variable may be located in an assembly language routine if the routine names the variable in an "XDEF" directive.

An MPL Procedure Call

Routine name

The entry point of an assembly language routine called by an MPL procedure must have a name in an "XDEF" directive.

Parentheses CALL

An assembly language routine called by an MPL statement of the form:

```
     CALL entry point(apl, ap2, ..., apn)
```

is guaranteed the following conditions on entry:

. The return address is on top of the stack. When this is true, the routine returns by an "RTS" instruction.

. A list of addresses of the actual parameters, "apl, ap2, ..., apn," is at the address equal to 2 plus the address on top of the stack. When this is true, the routine may load the X register with the address of the kth actual parameter by the instruction sequence:

```
     TSX
     LDX 0,X
     LDX 2k,X
```

Angle-brackets CALL

An assembly language routine called by an MPL statement of the form:

```
CALL entry point<ap1, ap2, ap3>
  GIVING <r1, r2, r3>
```

is guaranteed the following conditions on entry:

. The return address is on top of the stack. When this is true, the routine returns by an "RTS" instruction.

. The A, B, and X registers contain the values of the first, second, and third actual parameters, respectively. If any of the actual parameters are not specified in the "CALL" statement, the contents of the corresponding registers are undefined.

Before returning, the routine must load the A, B, and X registers with the values of the first, second, and third result parameters, respectively. If any of the result variables are not specified in the "CALL" statement, the corresponding registers need not be loaded.

Calling MPL Procedures

Procedure name

The name of an MPL procedure that is called by an assembly language routine is named in a "XREF" directive.

Parentheses PROCEDURE

An MPL procedure defined by a statement of one of the forms:

```
label constant:  PROCEDURE (fp1, fp2, fpn)
label constant:  PROC (fp1, fp2, fpn)
```

is called by the instruction sequence:

```
          JSR label constant
          BRA L
          FDB address of first actual parameter
          FDB address of second actual parameter
          .
          .
          .
          FDB address of nth actual parameter
        L .
          .
          .
```

Angle-brackets PROCEDURE

An MPL procedure defined by a statement of one of the forms:

```
label constant:  PROCEDURE <fpl, fp2, fp3>
label constant:  PROC <fpl, fp2, fp3>
```

is called by an instruction sequence with the effect:

```
Load A with value of first actual parameter
Load B with value of second actual parameter
Load X with value of third actual parameter
JSR label constant
```

If any of the formal parameters is not specified in the "PROCEDURE" statement, the corresponding registers are not loaded.

If the MPL procedure returns with a statement of the form:

```
RETURN <rpl, rp2, rp3>
```

then on return, the A, B, and X registers contain the values of the first, second, and third result parameters, respectively. If any of the result parameters is not specified in the "RETURN" statement, the contents of the corresponding registers are undefined.

CHAPTER 8. RELOCATABLE OBJECT MODULE LIBRARIES

Three relocatable object module libraries are provided with MPL: MPLSLIB.RO, MPLULIB.RO, and MPLUTLIB.Ro.

MPLSLIB.RO contains all of the "dot-F" routines called by compiled MPL programs. These are routines named ".FHH," where "HH" is two hexadecimal digits. It should always be included in the linking-load step. The "dot-F" routines perform functions such as BIN multiplication and division, DEC arithmetic, CHAR manipulation, and array subscripting.

MPLULIB.RO

MPLULIB.RO contains several procedures that user programs may call. It is not required during linking-load unless procedures being linking-loaded call its members. Other compiled procedures of general utility may be merged with MPLULIB.RO. This section defines the contents of MPLULIB.RO

Some procedures in MPLULIB.RO may only be used if the resultant absolute load module is running under CODOS. These load modules must meet the standard requirements for CODOS commands. They must be linking-loaded with the BASE command, contain no initialized BSCT, allocate sufficient stack space (at least 80 bytes), and fit within the available contiguous memory.

MPLULIB.RO Contents

DSPLY

Name: DSPLY

Function: Display the value of a "CHAR(m)" variable on the console.

CODOS Required: Yes

Calling Sequence:

```
CALL DSPLY<, , p>
```

where "p" is a pointer variable or address constant pointing to a "CHAR(m)" variable that terminates in or is followed by an ASCII carriage return character, "($D)." On return, the value of the "CHAR(m)" variable displays. For example:

```
DCL GREET CHAR(5) INIT('HELLO'),
    CR CHAR(1) INIT($D)
CALL DSPLY<, , ADDR(GREET)>
```

KEYIN

Name: KEYIN

Function: Read the value of a "CHAR(m)" variable from the console.

CODOS Required: Yes

Calling Sequence:
\as CALL KEYIN<, c, p> or CALL KEYIN<, c, p> GIVING <, n>
where "c" is an integer constant or "BIN(1)" variable with value "v." "p" is a pointer variable or address constant pointing to a "CHAR(m)" variable where "m >= v + 1," and "n" is a "BIN(1)" variable. "KEYIN" does not return until the user has typed k characters, "ala2...ak" followed by an ASCII carriage return character, "<CR>," on the keyboard.

On return, the first "d" characters of the "CHAR(m)" variable changes to "ala2...ad," where "d = min(v, k)," and the "(d + 1)st" character changes to "<CR>." If the second sequence is used, "n" is assigned the value of "d." For example:

```
DCL INPUT CHAR(81)
CALL KEYIN<, 80, ADDR(INPUT)>
```

CODOS

Name: CODOS

Function: Return control to CODOS.

CODOS Required: Yes

Calling Sequence:

```
CALL CODOS
```

Example:

```
CALL CODOS
```

PRINT

Name: PRINT

Function: Print the value of a "CHAR(m)" variable on the line printer.

CODOS Required: Yes

Calling Sequence:

```
     CALL PRINT<, , p>
```

where "p" is a "CHAR(m)" variable that terminates in or is followed by an ASCII carriage return character, "($D)." On return, the value of the "CHAR(m)" variable prints. For example:

```
     DCL GREET CHAR(5) INIT('HELLO'),
         CR CHAR(1) INIT($D)
     CALL PRINT<, , ADDR(GREET)>
```

PULL2

Name: PULL2

Function: Pull a "BIN(2)" value off of the stack.

CODOS Required: No

Calling Sequence:

```
     CALL PULL2 GIVING <, , v>
```

where "v" is a "BIN(2)" variable. On return, the "BIN(2)" value on top of the stack is pulled off and assigned to "v." For example:

```
     DCL P BIN(2)
     CALL PULL2 GIVING <, , P>
```

PUSH2

Name: PUSH2

Function: Push a "BIN(2)" value onto the stack.

CODOS Required: No

Calling Sequence:

```
     CALL PUSH2<, , v>
```

where "v" is a "BIN(2)" variable. On return, the value of "v" pushes onto the stack. For example:

```
     DCL P BIN(2)
     CALL PUSH2<, , P>
```

CKBRK

Name: CKBRK

Function: Check console for "BREAK" key depression.

CODOS Required: Yes

Calling Sequence:

```
     CALL CKBRK
```

On return, if "BREAK" key is depressed, the carry bit of the condition code register is set.

MPLUTLIB.RO

MPLUTLIB.RO is a library of utility subroutines that may be called from MPL programs. It contains two subroutine categories, mathematical and I/O.

MPLUTLIB.RO Contents

Mathematical Subroutines

ABS

Name: ABS

Function: Find the absolute value of a SIGNED BIN(1) variable.

CODOS Required: No

Calling Sequence:

```
CALL ABS<I> GIVING <J>
where I is a SIGNED BIN(1) variable,
and   J is a SIGNED BIN(1) variable
which is to receive the absolute value
of I.  J may be the same as I.
```

Result: J := |I|.

ABS2

Name: ABS2

Function: Find the absolute value of a SIGNED BIN(2) variable.

CODOS Required: No

Calling Sequence:

```
CALL ABS2<, , P> GIVING <, , Q>
where P is a SIGNED BIN(2) variable,
and   Q is a SIGNED BIN(2) variable
which is to receive the absolute of P.
Q may be the same as P.
```

Result: Q := |P|.

MOD

Name: MOD

Function: Find the modulus of a BIN(2) variable or constant with respect to a BIN(1) variable or constant.

CODOS Required: No

Calling Sequence:

```
     CALL MOD<I, , P> GIVING <J>
     where I is a BIN(1) variable or
     unsigned integer constant less
     than 256.
           P is a BIN(2) variable or
     unsigned integer constant less
     than 65536, and
           J is a BIN(1) variable which is
     to receive the modulus of P with
     respect to I.  J may be the same as I.
```

Result: J := P - [P/I]*I.

SETBIT

Name: SETBIT

Function: Find the exponential (base 2) of a BIN(1) variable.

CODOS Required: No

Calling sequence:

```
     CALL SETBIT<I> GIVING <, , P>
     where I is a BIN(1) variable, and
           P is a BIN(2) variable which is
     to receive a bit pattern with bit I,
     and bit I alone, set.
```

Result: P := 2**I.

I/O Subroutines

Three forms of I/O are provided by the MPLUTLIB.RO library: input, unformatted output, and formatted output. All three operate on ASCII record format files using record I/O and allow a maximum of 80 characters.

OPEN

Name: OPEN

Function: Open a disk file or the console for input or output, or the printer for output.

CODOS Required: Yes

Calling Sequence:

```
CALL OPEN<NBUFF, IO, FNPTR> GIVING <FILE,
ERROR, P>
```

where NBUFF is a BIN(1) variable or unsigned integer contant signifying the number of buffers to be used.
where IO is a BIN(1) variable or unsigned integer constant having value 1 (meaning open for input) or 2 (meaning open for output).
FNPTR is a BIN(2) variable or address constant pointing to a text area containing the file name (in ordinary operator-input format) or the device name (#CN or #LP), preceded and followed by file name terminators.
FILE is a BIN(1) variable which is to receive the file number for this file or device.
ERROR is a BIN(1) variable which is to receive the error code for this "open" operation, and
P is a BIN(2) variable which is to receive a pointer to the file or device name terminating character and the IOCB.

Result: If there are no errors, the file or device whose name is in the text area pointed to by FNPTR is opened for input or output (depending on the value of IO), FILE is assigned a unique number identifying the file or device for future I/O subroutine calls. If there is an error, FILE is set to 0 and ERROR is assigned a nonzero error code (see Error Table).

READ

Name: READ

Function: Read a record from a disk file or the console.

CODOS Required: Yes

Calling Sequence:

CALL READ<FILE, RSIZ, RPTR> GIVING <ERROR> where FILE is the BIN(1) variable which received the file number for the file or device when it was opened.

RSIZ is a BIN(1) variable which contains the record size in characters. A zero value defaults to 80 characters.

RPTR is a BIN(2) variable or address constant pointing to a text area of length 81 characters into which the record is to be read, and

ERROR is a BIN(1) variable which is to receive the error code for this "read" operation.

Result: If there are no errors, the next record from the file or device identified by FILE is read into the text area pointed to by RPTR and ERROR is set to 0. A carriage return follows the last character of the record. If FILE identifies the console, a question mark is printed as a prompt character. If an error occurs, ERROR is assigned a nonzero error code (see Error Table).

WRITE

Name: WRITE

Function: Write a record to a disk file, the console, or the printer.

CODOS Required: Yes

Calling Sequence:

CALL WRITE<FILE, RSIZ, RPTR> GIVING <ERROR>
where FILE is the BIN(1) variable which received the file number for the file or device when it was opened.

RSIZ is a BIN(1) variable which contains the record size in characters. A zero value defaults to 80 characters.
RPTR is a BIN(2) variable or address constant pointing to a text area containing the record to be written followed by a carriage return,
ERROR is a BIN(1) variable which is to receive the error code for this "write" operation.

Result: If there are no errors, the next record for the file or device identified by FILE is written from the text area pointed to by RPTR and ERROR is set to 0. All characters before the carriage return (up to a maximum of 80) are written. If there is an error, ERROR is assigned a nonzero error code (see Error Table).

WRITEF

Name: WRITEF

Function: Write the values of zero or more variables or constants to a disk file, the console, or the printer under control of a format string.

CODOS Required: Yes

Calling Sequence:

CALL WRITEF(FILE, FMT, I1, I2, ...)
where FILE is the BIN(1) variable which received the file number for the file or device when it was opened.
FMT is a text area containing a format string, and the
Ik's are BIN(n) variables, unsigned integer constants less than 65536, or text areas, the values of which are to be written.

Result: If there are no errors, the values of I1, I2, ... have been written to the file or device identified by FILE under control of the format string in FMT. Essentially, this controlled writing is a simple copying of the format string to the file or device, with special actions being taken when control sequences of the form '%c' are encountered in the string. Control sequences are not copied. The legal control sequences and their corresponding special actions are shown in this table:

%E	Terminate this "write" operation. '%E' marks the end of the format string. EVERY FORMAT STRING MUST END WITH '%E'.
%N	Write a new line, that is, terminate and write the current record and begin a new one. A record is not actually written until a '%N' is encountered.
%I	The next Ik in the parameter list is a BIN(1) variable. Write its value at this point.
%J	The next Ik in the parameter list is a SIGNED BIN(1) variable. Write its value at this point.
%P	The next Ik in the parameter list is a BIN(2) variable or an unsigned integer constant less than 65536. Write its value at this point.
%Q	The next Ik in the parameter list is a SIGNED BIN(2) variable. Write its value at this point.
%S	The next Ik in the parameter list is a text area of length STRSIZ characters. Write its value at this point.
%%	Write a percent sign.

Following a percent sign by any character other than the ones shown in the table results in a question mark being written.

CLOSE

Name: CLOSE

Function: Close a disk file, the console, or the printer.

CODOS Required: Yes

Calling Sequence:

```
CALL CLOSE<FILE> GIVING <ERROR>
where FILE is the BIN(1) variable
 which received the file number for
the file or device when it was
opened, and
      ERROR is a BIN(1) variable which
is to receive the error code for this
"close" operation.
```

Result: If there are no errors, the file or device identified by FILE is closed and ERROR is set to 0. If there is an error, ERROR is assigned a nonzero error code (see Error Table).

ERROR TABLE

1:	Device name not found
2:	Device already reserved
3:	Device not reserved
4:	Device not ready
5:	Invalid device
6:	Duplicate file name
7:	File name not found
8:	Invalid open/closed flag
9:	End of file
10:	Invalid file type
11:	Invalid data transfer type
12:	End of media
13:	Buffer overflow
14:	Checksum error
15:	File is write protected
16:	File is delete protected
17:	Logical sector number out of range
18:	Insufficient disk space
19:	Directory space full
20:	Segment descriptor space full
21:	Invalid directory entry no.
22:	Invalid RIB
23:	Cannot deallocate all space, directory entry exists
24:	Record length too large
25:	Sector buffer size error
32:	FILE does not identify a file or device
33:	No room for IOCB or sector buffer
34:	Memory allocation error
64:	IO is not 1 or 2, or SUFFIX is not alphabetic
65:	File name contains *
66:	Suffix contains *
67:	File name and suffix contain *
68:	Device other than #CN or #LP
192:	File name is null
193:	File name is *
194:	File name is null, suffix contains *
195:	File name is *, suffix contains *

CHAPTER 9. MPL COMPILER

Invocation

The MPL compiler is invoked from the CODOS command level by a command line of the form:

```
MPL <source file 1>,...,<source file n>;<options>
```

The "<source file 1>s" are the names of CODOS ASCII-record disk files containing the text of the MPL program. The files are logically linked before inputed to the compiler. Usually the contents of only one source file are compiled. Missing suffixes or drive numbers default to .SA or :0, respectively.

The clause ";<options>" is optional. If it appears, "<options>" is a concatenation of the following specifications:

Option	Meaning
L	Produce a source listing on the printer during compilation.
-L	Don't produce a source listing on the printer during compilation. Default.
L = filename	Produce a source listing in the output file during compilation.
M	If -L is specified, print compilation error messages on the printer.
-M	If -L is specified, don't print compilation error messages on the printer. Default.
S	Include the MPL source in the output file in the form of assembly-language comments.
-S	Don't include the MPL source in the output file in the form of assembly-language comments. Default.
O=<AI file>	Produce assembly-language output in the CODOS file called <AI file>. AI stands for assembler input.
-O	Don't produce assembly-language output. Default.

If the O option is specified, it must be the last option on the command line. A missing suffix or drive number in "<AI file>" defaults to ".SA" or ":0," respectively. If the source disk is too full to contain the "<AI file>," it should be created on a disk on the other drive.

If any option letters are omitted, MPL defaults to option specifications chosen from the string "-L-M-S-O." If "<options>" contains a sequence of mutually contradictory option specifications (such as S-S), the rightmost takes effect.

Another compiler option that implements as a local switch within the program text, rather than as a global switch on the command line, is the "SHORT/LONG" option.

Ordinarily, the compiler generates conditional control transfers as 5-byte code sequences. These sequences have the property to assemble without causing out-of-range-branch errors. The reserved word "SHORT" may be included anywhere within the text of a program (other than in comments, character-string constants, etc.) to instruct the compiler to generate 2-byte code sequences.

The 2-byte sequences cause out-of-range-branch errors if the destination is too far from the control transfer. "SHORT" is used only for this purpose. It is otherwise invisible and ignored during compilation. Conditional control transfers occur with the shorter sequence from the time "SHORT" is seen until the reserved word "LONG" is seen, or the end of the program text is encountered, whichever happens first. "LONG" causes a reversion to the longer sequence generation until "SHORT" appears again, and so on.

The "SHORT-LONG" option also affects the compiler's choice of a 2-byte or 3-byte code sequence for unconditional control transfers.

A program should be debugged without using "SHORT" or "LONG." If necessary, "SHORT-LONG" pairs may be inserted into the program text to optimize control transfers where it is possible to do so.

It is difficult to formulate a general rule for the exact placement of "SHORT-LONG" pairs to optimize the desired control transfers. This is because the compiler's scan of the source text does not exactly track its code generation. In the example below, 2-byte code sequences are generated for the conditional control transfer at the "DO WHILE" and for the unconditional control transfer at the "END."

```
/**/ SHORT /**/
DO WHILE P -> NEXT NE 0
P = P -> NEXT
END
/**/ LONG /**/
```

Here are some sample chain files that demonstrate the compilation, assembly, and linking-loading of MPL programs.

```
PAGE 001  MPLCA   .CF:0

/IFC F
/* NO FILE SPECIFIED
/ABORT
/ELSE
@SET,M 8
MPL %F%;L%C%O=%F%.AI
DEL %F%.RO
CMAP %F%.AI;LX%A%
DEL %F%.AI
/XIF
```

```
PAGE 001 MPLL      .CF:0

/IFC IN
/* NO INPUT SPECIFIED
/ABORT
/ELSE
/IFC OUT
/* NO OUTPUT SPECIFIED
/ABORT
/ELSE
@SET,M 8
DEL %OUT%.CM
RLOAD
IF=%OUT%
BASE
LOAD=%IN%
LIB=MPLSLIB,MPLULIB
OBJA=%OUT%.CM
MO=#LP
MAPF
EXIT
/XIF
/XIF

PAGE 001 MPLCAL    .CF:0

/IFC F
/* NO FILE SPECIFIED
/ABORT
/ELSE
@SET,M 8
MPL %F%;L%C%O=%F%.AI
CMAP %F%.AI;LX%A%
@TST,T EQ,0
@JMP L1
DEL %F%.CM
RLOAD
IF=%F%
BASE
LOAD=%F%
LIB=MPLSLIB,MPLULIB
OBJA=%F%.CM
MO=#LP
MAPF
EXIT
@LBL L1
DEL %F%.AI
DEL %F%.RO
/XIF
```

Results

If one of the source files does not exist, the CODOS message:

```
FILENAME NOT FOUND
```

appears on the console and compilation ceases. If the output file exists, the CODOS message:

```
DUPLICATE FILE NAME
```

appears. if the "L" or "M" option specification appears on the command line and the printer is offline when the compiler attempts to use it, the message:

```
PRINTER NOT READY
```

appears, and the compiler waits for the printer to come back online. Otherwise, compilation proceeds. An output file suitable for subsequent assembly is created on request.

If "L" appears on the command line, a source listing prints during compilation. If the file contains sequence numbers, these appear on the listing. If not, numbers ascending from 10, in steps of 10, appear instead.

If any errors are encountered during compilation, error messages print on the printer (if L or M appears on the command line) and are included in the output file. The CODOS system error status word is set to "$80." In any case, the number "n" of errors prints on the console in the form:

```
TOTAL ERRORS n
```

Error messages have the form:

```
***** nnnn cc...c
*ERROR eee    *
```

where "nnnn" is the sequence number; "cc...c" is the text of the last line read; and "eee" is the error number as described in the Appendix. The error involves one of the two symbols prior to the position of the second asterisk on the "*ERROR" line. If there is only one symbol prior to this position, the error may involve the last symbol on the previous line.

For example, an attempted CHAR(1) to BIN(1) assignment results in an error 552 as shown here:

```
  10 DCL B1, C1 CHAR
  20 B1 = C1
  30 GOTO L
*****   30 GOTO L
*ERROR 552      *
```

It is not advisable to assemble the output file until you have an error-free compilation.

Appendices

APPENDIX A. MPL EXAMPLES

This section shows some sample MPL programs and demonstrates the use of the chain files MPLCAL, MPLCA, and MPLL.

Sample MPL Program

The sample program below illustrates some typical MPL statements.

```
SHORT
WRSTR:
   PROCEDURE <, , CHPTR>
   DECLARE CHPTR BIN(2), CH CHAR BASED
   CALL WRNL
   DO WHILE CHPTR -> CH NE $4
      CALL WRCH<CHPTR -> CH>
      CHPTR = CHPTR + 1
      END
   RETURN
   END
```

The above program is a subroutine procedure called "WRSTR." A subroutine procedure is a named routine to perform a particular task. Such a routine may be executed at many different times by other routines and may be instructed to perform its task using different sets of data. It is executed by a single MPL statement specifying its name and the date it is to use. "WRSTR's" job is to show, on a display screen, an ASCII character string terminated by an ASCII EOT character (hexadecimal 04). It is executed via an MPL "CALL" statement specifying its name, "WRSTR," and the address of the character string it is to display.

The program begins with the MPL reserved word "SHORT." This word is included for optimization. It has no effect on the task performed by "WRSTR."

The procedure begins with its name, "WRSTR," followed by a colon, the word "PROCEDURE," and the formal parameter list "<, , CHPTR>." "WRSTR:" is a statement label.

The "PROCEDURE" statement is labelled "PROCEDURE <, , CHPTR>." This establishes the procedure's name, so that other procedures may name it in "CALL" statements. The "PROCEDURE" statement announces that what follows is a procedure with one formal parameter or dummy argument (CHPTR). This means that when the procedure is activated, the memory address of some character string is in the X register.

"CHPTR" is a symbolic name, chosen by the programmer, to refer to this address within the procedure. "CHPTR" is an abbreviation of "character pointer," since it is the address of (points to) a character.

Following the "PROCEDURE" statement is a "DECLARE" statement. This states the variables the procedure uses and the type of data to which they refer. Neither the "PROCEDURE" statement nor the "DECLARE" statement is executable. When the program is executed, nothing will happen as a direct result of these statements. They simply state some facts about the program.

"CHPTR," the formal parameter, is a 2-byte binary variable, which is the type MPL provides to refer to memory addresses. "CH," a "BASED" variable, refers to an ASCII character. The fact that it is "BASED" means that by itself, it doesn't really refer to a character. It provides a method for getting to the byte addressed by some "BIN (2)" variable as a character. For example, "CHPTR -> CH" means "the character whose address is the value of CHPTR."

Note that nothing would prevent the declaration of a "BASED BIN(2)," variable "PTR," and the subsequent use of "CHPTR -> PTR" to mean "the memory address whose memory address is the value of CHPTR." In this case, such use would be questionable since "CHPTR" contains the address of a character string, not a meaningful memory address.

The next statement is a "CALL" statement. It is the first executable statement of this procedure. This means that when the program is executed, the first thing "WRSTR" does is to call another procedure: "WRNL." "WRNL" is not shown, but it is a procedure that successively writes a carriage return, linefeed, and null to the display screen. Execution then continues in "WRSTR" at the statement following the "CALL" statement.

The next statement is called a "DO" statement. When the word "WHILE" is used, as in this example, all statements down to the matching "END" statement are repeatedly executed, as long as the condition in the "DO" statement is true. The condition here is "CHPTR -> CH NE $4." This means "the character whose address is the value of 'CHPTR' is not equal to a hexadecimal 4 (ASCII EOT)." Thus, the statements between the "DO" statement and the next "END" statement are executed when CHPTR "points to" an EOT.

The first of the two repeated statements is another "CALL" statement. This calls the procedure "WRCH," and passes it the character whose address is the value of "CHPTR." "WRCH" is not shown, but it is a procedure that writes the character it receives to the display screen.

The second of the repeated statements is the assignment statement. The assignment statement evaluates the arithmetic expression on the right of the equal sign and assigns the resulting value to the variable on its left. This results in the incrementing of "CHPTR" by 1, which makes it contain the address of the character following the one just written.

The "END" statement marks the end of the statements affected by the "DO" statement. The "DO" statement writes each character in the string, up to but not including the EOT.

The next statement is a "RETURN" statement. It specifies that at this point, control is to return to the statement following the "CALL" statement that invokes "WRSTR."

The following "END" statement marks the end of the procedure. "END" statements are not executable; they only mark the end of DO-groups or procedures.

The MPL Compiler

An MPL program is analyzed by a program called the MPL compiler. This compilation process produces an assembly language equivalent of the program. This equivalent is in a format suitable for assembly by the Macroassembler. After assembly, the resultant relocatable object module may be combined with others by the Linking Loader to produce an absolute load module in a format suitable for loading by CODOS loader.

During the analysis of the MPL program, the MPL compiler can also print a source listing and diagnostic messages at points where the structure of the program fails to comform to the structure of a legal MPL program.

Suppose the EDITOR has been used to store the sample program on disk in a file called WRSTR.SA. The CODOS MPL and CMAP commands may then be used to create and assemble an assembly input file WRSTR.AI as shown:

```
=MPL WRSTR;O=WRSTR.AI

CODOS MPL COMPILER X.XX
COPYRIGHT BY CODEX 1980
TOTAL ERRORS 0
=CMAP WRSTR.AI;L
CODOS MACROASSEMBLER X.XX
COPYRIGHT BY CODEX 1980


=
```

The result of this process is a relocatable object module in the file WRSTR.RO and an assembly source listing, shown here:

```
00001                       *** COMPILED WITH MPL X.XX
00002                              OPT    REL
00003                       *  10 SHORT
00004                       *  20
00005                       *  30 WRSTR:
00006                       *  40   PROCEDURE <, , CHPTR>
00007          0000   P WRSTR  EQU     *
00008                              XDEF   WRSTR
00009P 0000 FF 0000  D             STX    CHPTR
00010       *  50   DECLARE CHPTR BIN(2), CH CHAR BASED
00011D 0000                        DSCT
00012D 0000    0002  A CHPTR   RMB    2
00013          0000  A CH      EQU    0
00014P 0003                        PSCT
00015                       *  60   CALL WRNL
00016P 0003 BD 0000  A             JSR    WRNL
00017                 *  70    DO WHILE CHPTR -> CH NE $4
00018P 0006 FE 0000  D .001      LDX    CHPTR
00019P 0009 A6 00    A             LDAA    0,X
00020P 000B 81 04    A             CMPA    #4
00021P 000D 27 0C 001B             BEQ     .002
00022                     *  80      CALL WRCH<CHPTR -> CH>
00023P 000F BD 0000  A             JSR    WRCH
00024                       *  90       CHPTR = CHPTR + 1
00025P 0012 FE 0000  D             LDX    CHPTR
00026P 0015 08                     INX
00027P 0016 FF 0000  D             STX    CHPTR
00028P 0019 20 EB 0006             BRA    .001
00029                       * 100       END
00030                       * 110      RETURN
00031P 001B 39              .002     RTS
00032                       * 120     END
00033                              XREF   WRCH
00034                              XREF   WRNL
00035N 0000                 T$     COMM   DSCT
00036N 0000    000B  A .T      RMB    11
00037                              END
TOTAL ERRORS 00000
```

Echo Program Example

Figure 1 shows the text of the first sample MPL program. It is a main procedure called "ECHO" whose function is to read a line from the console and print it back on the console. "ECHO" assumes the existence of three procedures, KEYIN, DSPLY, and CODOS, which do console input, console output, and return to CODOS, respectively. These and other routines are in MPLULIB.RO. They consist of the appropriate CODOS system calls. The line is read into and written from the CHAR(80) variable "BUFFER." "BUFEND" is provided for the terminating carriage return on 80-character lines.

Figure 2 shows the invocation of MPLCAL to compile, assemble, and linking-load this program which is contained in the file SAMPLE.SA:0. The compilation produces the source listing in Figure 1. The assembly produces the assembly listing in Figure 3, and the linking-load produces the load map in Figure 4.

Note the subsequent execution in Figure 2 of the program, which is left in the file "SAMPLE.CM:0," and the state of the disk directory when the process completes.

Binary Tree Sort Program

Figures 5, 6, and 7 show the text of the second sample MPL program. Its main procedure, "MN," repeatedly accepts two-digit decimal numbers from the console and calls a procedure "INSERT" to insert them into the appropriate positions in a binary tree. When "00" is entered, "MN" calls the procedure "PTREE" to print the tree and return to CODOS.

"INSERT" begins at the head of the tree (pointed to by the EXTERNAL BIN(2) variable HEAD) and "walks" down to the appropriate terminal node and takes the right branch when the number exceeds the value at the current node and the left branch otherwise.

Note the use of the form "PT + ADDR(BA)" to achieve the effect of the (forbidden) form "ADDR(PT -> BA)." When the terminal node is reached, a new node is allocated from the tree space variable "SPC" and filled with the number and null left-branch and right-branch pointers.

"PTREE" prints the tree recursively, by calling itself. Basically it operates as follows:

1. Call myself to print this node's left branch;
2. Print this node's value;
3. Call myself to print this node's right branch.

The exception is when it is called to print a null tree, it does nothing but return. The recursive calls are effected by saving the previous tree pointer on the stack using MPLULIB.RO procedure PUSH2, setting up the new tree pointer, calling PTREE, and restoring the previous tree pointer using the MPLULIB.RO procedure PULL2.

Figures 8, 9, and 10 show the separate compilations and assemblies of these three procedures using the chain file MPLCA. Figure 11 shows the use of MPLL to linking-load the resulting object modules into the load module TREESORT.CM:0. Figure 12 shows the load map resulting from the linking-load. Figure 13 shows the execution of TREESORT to sort 10 numbers.

```
10 /*    This is a sample MPL program    */
20 ECHO:
30   PROC OPTIONS(MAIN)
40 $ NAM ECHO
50   DCL BUFFER CHAR(80), BUFEND CHAR
60   CALL KEYIN<, 80, ADDR(BUFFER)>
70   CALL DSPLY<, , ADDR(BUFFER)>
80   CALL CODOS
90   END
```

Figure 1

```
=CHAIN MPLCAL;F%SAMPLE%
@SET FOFF 0800
MPL SAMPLE;LO=SAMPLE.AI

CODOS MPL COMPILER X.XX
COPYRIGHT BY CODEX 1980
TOTAL ERRORS     0
CMAP SAMPLE.AI;LX
CODOS MACROASSEMBLER X.XX
COPYRIGHT BY CODEX 1980


@TST,00FF 0000 0027
DEL SAMPLE.CM
SAMPLE  .CM:0 DOES NOT EXIST
RLOAD
CODOS LINKING LOADER REV 2.03
COPYRIGHT BY CODEX 1980
?IF=SAMPLE
?BASE
?LOAD=SAMPLE
?LIB=MPLSLIB,MPLULIB
?OBJA=SAMPLE.CM
?MO=#LP
?MAPF
?EXIT
@LBL 2F23
DEL SAMPLE.AI
SAMPLE  .AI:0 DELETED
DEL SAMPLE.RO
SAMPLE  .RO:0 DELETED
END CHAIN
=SAMPLE
THIS MESSAGE WILL BE ECHOED
THIS MESSAGE WILL BE ECHOED
=DIR SAMPLE.*;A
DRIVE : 0   DISK I.D. : CODOS022X
SAMPLE  .CM   ...C.2 0680 0004 53    00 0680 004
SAMPLE  .SA   .....5 04B4 0004 9B    00 04B4 004
TOTAL NUMBER OF SECTORS : 0008/$008
TOTAL DIRECTORY ENTRIES SHOWN : 002/$02
=
```

Figure 2

```
PAGE  001    ECHO

00001                        *** COMPILED WITH MPL X.XX
00002                               OPT    REL
00003                        *  10 /*    This is a sample MPL
                             program     */
00004                        *  20  ECHO:
00005                        *  30   PROC OPTIONS(MAIN)
00006           0000  P ECHO    EQU    *
00007                               XDEF    ECHO
00008P 0000  8E 00B4  D .000    LDS     #.S
00009                               NAM     ECHO
00010                        *  50   DCL BUFFER CHAR(80), BUFEND
                                                          CHAR
00011D 0000                         DSCT
00012D 0000     0050  A BUFFER RMB    80
00013D 0050     0001  A BUFEND RMB    1
00014P 0003                         PSCT
00015                        *  60   CALL KEYIN<, 80, ADDR(BUFFER)>
00016P 0003 C6 50     A             LDAB   #80
00017P 0005 FE 0014   P             LDX    .366
00018P 0008 BD 0000   A             JSR    KEYIN
00019                        *  70   CALL DSPLY<, , ADDR(BUFFER)>
00020P 000B FE 0014   P             LDX    .366
00021P 000E BD 0000   A             JSR    DSPLY
00022                        *  80   CALL CODOS
00023P 0011 BD 0000   A             JSR    CODOS
00024                        *  90   END
00025                               XREF   CODOS
00026                               XREF   DSPLY
00027P 0014     0000  D .366    FDB    BUFFER
00028                               XREF   KEYIN
00029N 0000              T$      COMM   DSCT
00030N 0000     000A  A .T      RMB    10
00031D 0051                         DSCT
00032D 0051     0064  A             RMB    100
00033           00B4  D .S      EQU    *-1
00034           0000  P             END    .000
TOTAL ERRORS 00000
```

```
 P 0000 .000   00008*00034
 P 0014 .366   00017 00020 00027*
 D 00B4 .S     00008 00033*
ND 0000 .T     00030*
 D 0050 BUFEND 00013*
 D 0000 BUFFER 00012*00027
R       DSPLY  00021 00026*
DP 0000 ECHO   00006*00007
R       KEYIN  00018 00028*
R       CODOS  00023 00025*
ND      T$     00029*
```

Figure 3

```
  NO UNDEFINED SYMBOLS
MAP
 S SIZE  STR  END COMN
 B 0000 0000 FFFF 0000
 C 0000 0000 FFFF 0000
 D 00BF 2000 20BE 000A
 P 001F 20BF 20DD 0000
MODULE NAME BSCT DSCT PSCT
  ECHO       0000 2000 20BF
  CODOS      0000 20B5 20D5
  DSPLY      0000 20B5 20D8
  KEYIN      0000 20B5 20DB
COMMON
 NAME   S SIZE  STR
T$      D 000A 20B5
DEFINED SYMBOLS
 NAME   S  STR  NAME   S  STR  NAME   S  STR  NAME   S  STR
ECHO    P 20BF CODOS   P 20D5 DSPLY   P 20D8 KEYIN   P 20DB
```

Figure 4

```
/* GET 2-DIGIT NUMBERS (ENDED BY 00) AND SORT THEM
   USING A BINARY TREE SORT                        */
SHORT
MN:
  PROC OPTIONS(MAIN)
$ NAM MN
  DCL HEAD BIN(2) GLOBAL INIT(0), N GLOBAL
  DCL P BIN(2) GLOBAL
  DCL ND DEC(2), CR CHAR(1)

/* GET AND INSERT NUMBERS UNTIL YOU GET 00 */
  CALL KEYIN<, 2, ADDR(ND)>
  N = ND
  DO WHILE N NE 0
    CALL INSERT
    CALL KEYIN<, 2, ADDR(ND)>
    N = ND
    END

/* PRINT THE TREE, SORTED */
  P = HEAD
  CALL PTREE

  CALL CODOS
  END
```

Figure 5

```
 10 /* INSERT - INSERT N INTO BINARY TREE */
 20 SHORT
 30 INSERT:
 40   PROC
 50 $ NAM INSERT
 60   DCL Q BIN(2), R BIN(2)
 70   DCL HEAD BIN(2) EXTERNAL, N EXTERNAL
 80   DCL PTR BIN(2) BASED,
 90       1 NODE BASED,
100         2 VALUE,
110         2 LEFT BIN(2),
120         2 RIGHT BIN(2)
130   DCL SPC CHAR(255), SPCTOP BIN(2) INIT(ADDR(SPC))
140
150 /* FIND LEAF WHERE N GOES */
160   Q = ADDR(HEAD)
170   DO WHILE Q -> PTR NE 0
180     R = Q -> PTR
190     IF N LE R -> VALUE THEN Q = R + ADDR(LEFT)
200                        ELSE Q = R + ADR(RIGHT)
210     END
220
230 /* CREATE NEW NODE FOR N THERE */
240   R = SPCTOP
250   SPCTOP = SPCTOP + 5
260   Q -> PTR = R
270   R -> VALUE = N
280   R -> LEFT = 0
290   R -> RIGHT = 0
300
310   RETURN
320   END
```

Figure 6

```
 10 /* PTREE - RECURSIVELY PRINT BINARY TREE */
 20 SHORT
 30 PTREE:
 40   PROC
 50 $ NAM PTREE
 60   DCL P BIN(2) EXTERNAL
 70   DCL VALUED DEC(2), CR CHAR(1) INIT($0D)
 80   DCL 1 NODE BASED,
 90         2 VALUE,
100         2 LEFT BIN(2),
110         2 RIGHT BIN(2)
120
130   IF P NE 0 THEN
140     DO
150
160 /* SAVE THIS NODE ON STACK AND RECURSIVELY PRINT LEFT
    BRANCH */
170     CALL PUSH2<, , P>
180     P = P -> LEFT
190     CALL PTREE
200     CALL PULL2 GIVING <, , P>
210
220 /* PRINT VALUE AT THIS NODE */
230     VALUED = P -> VALUE
240     CALL DSPLY<, , ADDR(VALUED)>
250
260 /* SAVE THIS NODE ON STACK AND RECURSIVELY PRINT RIGHT
    BRANCH */
270     CALL PUSH2<, , P>
280     P = P -> RIGHT
290     CALL PTREE
300     CALL PULL2 GIVING <, , P>
310     END
320
330   RETURN
340   END
```

Figure 7

```
=CHAIN MPLCA;F%MN%
@SET FOFF 0800
MPL MN;LO=MN.AI

CODOS MPL COMPILER X.XX
COPYRIGHT BY CODEX 1980
TOTAL ERRORS
DEL MN.RO
MN      .RO:0 DOES NOT EXIST
CMAP MN.%AI;LX
CODOS MACROASSEMBLER X.XX
COPYRIGHT BY CODEX 1980

DEL MN.AI
MN      .AI:0 DELETED
END CHAIN
=
```

Figure 8

```
=CHAIN MPLCA;F%INSERT%
@SET FOFF 0800
MPL INSERT;LO=INSERT. AI

CODOS MPL COMPILER X.XX
COPYRIGHT BY CODEX 1980
TOTAL ERRORS    0
DEL INSERT.RO
INSERT  .RO:0 DOES NOT EXIST
CMAP INSERT.AI;LX
CODOS MACROASSEMBLER X.XX
COPYRIGHT BY CODEX 1980


DEL INSERT.AI
INSERT  .AI:0 DELETED
END CHAIN
=
```

Figure 9

```
=CHAIN MPLCA;F%PTREE%
@SET FOFF 0800
MPL PTREE;LO=PTREE.AI

CODOS MPL COMPILER X.XX
COPYRIGHT BY CODEX 1980
TOTAL ERRORS     0
DEL PTREE.RO
PTREE   .RO:0 DOES NOT EXIST
CMAP PTREE.AI;LX
CODOS MACROASSEMBLER X.XX
COPYRIGHT BY CODEX 1980


DEL PTREE.AI
PTREE   .AI:0 DELETED
END CHAIN
=
```

Figure 10

```
=CHAIN MPLL;IN%PTREE,INSERT,MN%,OUT%TREESORT%
@SET FOFF 0800
DEL TREESORT.CM
TREESORT.CM:0 DOES NOT EXIST
RLOAD
CODOS LINKING LOADER REV X.XX
COPYRIGHT BY CODEX 1980
?IF=TREESORT
?BASE
?LOAD=PTREE,INSERT,MN
?LIB=MPLSLIB,MPLULIB
?OBJA=TREESORT.CM
?MO=#LP
?MAPF
?EXIT
END CHAIN
=
```

Figure 11

```
  NO UNDEFINED SYMBOLS
MAP
 S SIZE  STR  END  COMN
 B 0000 0000 FFFF 0000
 C 0000 0000 FFFF 0000
 D 0181 2000 2180 000B
 P 0281 2181 2401 0000
MODULE NAME BSCT DSCT PSCT
  PTREE      0000 2000 2181
  INSERT     0000 2003 21CA
  MN         0000 2108 2243
  .F00       0000 2174 228E
  .F01       0000 2174 2328
  .F07       0000 2174 23BF
  .F073      0000 2174 23C6
  .K10K      0000 2174 23D3
  CODOS      0000 2174 23DD
  DSPLY      0000 2174 23E0
  KEYIN      0000 2174 23E3
  PUSH2      0000 2174 23E6
  PULL2      0000 2176 23F8
COMMON
 NAME  S SIZE  STR
T$     D 000B 2176
DEFINED SYMBOLS
 NAME  S  STR  NAME   S  STR  NAME   S  STR  NAME   S  STR  NAME
  S  STR
PTREE  P 2181 INSERT P 21CA HEAD   D 2108 MN     P 2243 N
  D 210A
P      D 210B .F00   P 228E .F01   P 2328 .F07   P 23BF .F073
  P 23C6
.K10K  P 23D3 CODOS  P 23DD DSPLY  P 23E0 KEYIN  P 23E3 PUSH2
  P 23E6
PULL2  P 23F8
```

Figure 12

```
=TREESORT
25
10
99
12
67
25
26
66
65
68
00
10
12
25
25
26
65
66
67
68
99
=
```

Figure 13

APPENDIX B. ASCII CHARACTER SET

BITS 4 to 6 --		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	'	p
B	1	SOH	DC1	!	1	A	Q	a	q
I	2	STX	DC2	"	2	B	R	b	r
T	3	ETX	DC3	#	3	C	S	c	s
S	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
0	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
3	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	¦
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	—	o	DEL

APPENDIX C. MPL SYNTAX

```
<program> ::= [<statement list>]


<statement list> ::= <statement> ¦ <statement list>
<statement>


<statement> ::= <label> <statement>
              ¦ <procedure>
              ¦ <DCL statement>
              ¦ <IF statement>
              ¦ <DO>
              ¦ <GOTO statement>
              ¦ <CALL statement>
              ¦ <assignment statement>
              ¦ <RETURN statement>
              ¦ <END statement>


<label> ::= <label constant> : ¦ $ <assembly language
statement>

<procedure> ::= <label constant> : {PROC ¦ PROCEDURE}
                                   [<options clause>
                                 ¦ < <formal parameter list> >
                                ¦ ( <formal parameter list> )]
                                  <block end>

<options clause> := OPTIONS (MAIN [, SS = <integer
                                          constant>])

<formal parameter list> ::= [<formal parameter>]
                          ¦ <formal parameter list> ,
                            [<formal parameter>]

<formal parameter> ::= <undeclared name> ¦ <declared name>

<block end> ::= [<statement list>] [label list>] <END
statement>

<label list> ::= <label> ¦ <label list> <label>

<END statement> ::= END [<label constant>]

<DCL statement> ::= {DCL ¦ DECLARE} [<section name>] <item
list>

<section name> ::= BSCT ¦ CSCT ¦ DSCT ¦ PSCT

<item list> ::= <item> ¦ <item list> , <item>
```

```
<item> ::= [<level number>] <undeclared name>
                            [<dimension designator>]
                            {LABEL [<INITIAL attribute>]
                           ¦ [<type attribute>]
                             [<other attribute>]}

<level number> ::= 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5

<dimension designator> ::= ( <integer constant> )
                         ¦ ( <integer constant> ,
                             <integer constant> )
                         ¦ ( <integer constant> ,
                             <integer constant> ,
                             <integer constant> )

<INITIAL attribute> ::= {INIT ¦ INITIAL} ( <constant list> )

<constant list> ::= <constant> ¦ <constant list> ,<constant>

<type attribute> ::= SIGNED
                   ¦ BIN [( <integer constant> )]
                   ¦ SIGNED BIN [( <integer constant> )]
                   ¦ DEC [( <integer constant>
                           [, <integer constant>] )]
                   ¦ SIGNED DEC [( <integer constant>
                                  [, <integer constant>] )]
                   ¦CHAR [( <integer constant> )]
                   ¦BIT [( <integer constant> )]

<other attribute> ::= <INITIAL attribute>
                    ¦ BASED
                    ¦ {DEF  ¦ DEFINED}
                  {<declared name> ¦ <hexadecimal constant>}
                    ¦ EXTERNAL
                    ¦ GLOBAL

<IF statement> ::= IF <logical expression> THEN <statement>
                                           [ELSE <statement>]

<logical expression> ::= <logical term>
                       ¦ <logical expression> OR <logical term>


<logical term> ::= <logical factor>
                 ¦ <logical term> AND <logical factor>

<logical factor> ::= [NOT] <logical primary>

<logical primary> ::= <relation> ¦ ( <logical expression> )

<relation> ::= <arithmetic expression>
               <relational operator> <arithmetic expression>
```

```
<arithmetic expression> ::= <arithmetic term>
                          ¦ <arithmetic expression>
                            + <arithmetic term>
                          ¦ <arithmetic expression>
                            - <arithmetic term>

<arithmetic term> ::= <arithmetic factor>
                    ¦ <arithmetic term> * <arithmetic factor>
                    ¦ <arithmetic term> / <arithmetic factor>

<arithmetic factor> ::= <arithmetic pattern>
                      ¦ <arithmetic factor>
                        {IAND ¦ &} <arithmetic pattern>
                      ¦ <arithmetic factor>
                        IOR <arithmetic pattern>
                      ¦ <arithmetic factor>
                        IEOR <arithmetic pattern>

<arithmetic pattern> ::= <arithmetic slide>
                       ¦ <arithmetic pattern>
                         {SHIFT ¦ %} [-] <constant>

<arithmetic slide> ::= [-] <arithmetic primary>

<arithmetic primary> ::= <constant>
                       ¦ <variable>
                       ¦ ( <arithmetic expression> )

<variable> ::= <declared name> [-> <declared name>]
                               [( <subscript list>  )]

<subscript list> ::= <subscript>
                   ¦ <subscript> , <subscript>
                   ¦ <subscript> , <subscript> , <subscript>

<subscript> ::= <integer constant>
              ¦ <declared name> [{+ ¦ -} <integer constant>]

<relational operator> ::= EQ ¦ NE ¦ LE ¦ GE ¦ LT ¦ < ¦ GT ¦ >


<DO> ::= [<iteration clause>] [<WHILE clause>] <block end>


<iteration clause> ::= <variable> = <DO operand> TO <DO
operand>
                                                [BY <DO operand>]


<DO operand> ::= <variable> ¦ <constant>

<WHILE clause> ::= WHILE <logical expression>
```

```
<GOTO statement> ::= {GOTO ¦ GO TO} {<label constant>
                                    ¦ <variable>
                                    ¦ ( <label constant list>)
                                      [,] <variable>}

<label constant list> ::= <label constant>
                        ¦ <label constant list> ,
                          <label constant>

<CALL statement> ::= CALL <label constant>
                         [( <actual parameter list> )
                       ¦ [< <actual parameter list> >]
                         [GIVING < <formal parameter list>>]]

<actual parameter list> ::= [<actual parameter>]
                          ¦ <actual parameter list> ,
                            [<actual parameter>]

<actual parameter> ::= <variable> ¦ <constant>

<assignment statement> ::= <variable list>
                            = <arithmetic expression>

<variable list> ::= <variable> ¦ <variable list> , <variable>

<RETURN statement> ::= RETURN [< <actual parameter list> >]
```

APPENDIX D. MPL COMPILER OPTIONS

Command line:

```
MPL <source file 1>[,...,<source file n>][;<options>]
```

Options:

L	Source listing (on printer).
-L	No source listing.
L = filename	Source listing is filename.
M	Error listing (on printer) if -L.
-M	No error listing (if -L).
S	Include source in AI file
-S	Don't include source in AI file.
O=<AI file>	Put output in disk file <AI file>. Must be last.
-O	No output. Must be last.

Defaults:

```
-L-M-S-O
File suffixes = .SA
File drives = :0
```

In program text:

```
SHORT => generate short branch sequence
LONG => generate long branch sequence
```

APPENDIX E. MPL RESERVED WORDS

These words may not be used as names of statements or data items by the MPL programmer:

A	DCL	GIVING	LE	RETURN
ADDR	DEC	GLOBAL	LONG	SHIFT
AND	DECLARE	GO	LT	SHORT
B	DEF	GOTO	MAIN	SIGNED
BASED	DEFINED	GT	NARG	SS
BIN	DO	IAND	NE	THEN
BIT	DSCT	IEOR	NOT	TO
BSCT	ELSE	IF	OPTIONS	WHILE
BY	END	INIT	OR	X
CALL	EQ	INITIAL	PROC	
CHAR	EXTERNAL	IOR	PROCEDURE	
CSCT	GE	LABEL	PSCT	

APPENDIX F. MPL COMPILER ERROR MESSAGES

501: Compiler error. (Perge net stack overflow.)

502: Symbol table overflow.

503: Syntax error. Different kind of symbol expected.

504: Declared name D found when undeclared name expected in one of these contexts:

```
DCL ..., D ...
GOTO (..., D, ...), I
CALL D ...
```

505: Undeclared name U found in one of these contexts:

```
CALL P(..., U, ...)          (expecting variable
                              or constant)
CALL P<..., U, ...>          (expecting variable,
                              constant, or comma)
RETURN <..., U, ...>         (expecting variable,
                              constant, or comma)
DCL ..., AA ... DEF U, ...   (expecting declared
                              name or hexadecimal
                              constant)
```

```
Variable, that is
   U       (expecting declared name)
   AA => U (expecting declared name)
   AA(U)   (expecting declared name or integer
            constant)
```

506: Scan error. One of these cases:

- Character-string constant empty or containing single !.
- Address constant containing ' or ! or).
- Hexadecimal constant containing extra $ or letter beyond F or '.
- Integer constant or decimal constant containing letter.

507: Symbol over 30 characters long.

508: Missing level number in structure declaration.

509: Redeclaration of declared name, or declaration of a formal parameter within a structure.

510: n > 9 in DEC(m, n) or SIGNED DEC(m, n) declaration, or n specified in BIN, SIGNED BIN, BIT, CHAR, or LABEL declaration.

511: Character-string constant longer than context, or error in INITIAL attribute in one of these ways:

- A CHAR)m) variable, m > 2, initialized to an integer, decimal, or hexadecimal constant.
- A DEC(m, n) or SIGNED DEC(m, n) variable initialized to a character-string or hexadecimal constant, or an integer or decimal constant with too many digits.
- A variable with size other than 2 initialized to an address constant.

512: INITIAL attribute specified for BIT(m) variable or a variable contained in a DEFINED substructure.

515: Level number greater than 5 in structure declaration.

516: GOTO undeclared formal parameter.

517: Declared name or undeclared formal parameter N found in the context P: PROC(..., N, ...).

518: Index of computed GOTO is not BIN.

519: Index of computed GOTO does not have size 1.

520: Index of computed GOTO or actual parameter in parentheses CALL is qualified or subscripted, or is a formal parameter.

522: Compiler error.

523: Too many operands in expression.

524: Variable requires more than 3 subscripts.

525: Compiler error.

526: Pointer variable is contained in a structure.

527: Pointer variable has the array attribute.

528: Pointer variable does not have size 2.

529: Pointer variable is not BIN.

530: Compiler error.

531: Variable has more than three variable subscripts.

532: Subscript is contained in a structure or is a formal parameter.

533: Subscript has the array attribute.

534: Subscript does not have size 1.

535: Subscript is not BIN.

537: Constant used in invalid context. One of these cases:

- Address constant used in CHAR, DEC, or SIGNED DEC context.
- Character-string constant used in DEC or SIGNED DEC context.
- Hexadecimal constant used in CHAR(m) context, m > 2, or in DEC or SIGNED DEC context.

538: Constant used as first operand of expression.

539: Wrong number of subscripts used.

540: Variable or constant in DO statement is not BIN.

541: DOs and IFs are nested more than 20 deep.

542: DO-IF stack underflow. Usually too many END statements, or caused by other error.

543: DO-IF stack overlap. Usually too many END statements, or caused by other error.

544: DOs are nested more than 10 deep.

545: Increment of DO does not have size 1.

546: Initial or final value of DO does not have size equal to that of the index of the DO.

547: Variable in DO statement is qualified or subscripted, or is a formal parameter.

548: Shift-rotate count is not a constant.

549: Shift-rotate count is zero.

551: Operation is illegal for operands of these types.

552: Mixed-mode expression, or illegal implicit conversion in assignment.

555: In a BIT(m) operation, the second operand is not a constant with size 1 and value 0 or 2**m-1; or, if the operation is comparison for equality or inequality and m > 1, it is not a constant with size 1 and value 0.

556: A BIT(m) operation is neither assignment nor comparison for equality or inequality.

558: Undefined action requested.

560: Variable referring to composite data item has type or INITIAL attribute.

561: BIT(m) data item crosses byte boundary.

564: Undeclared name found when BIN, CHAR, or DEC expected in declaration.

565: Array variable declared with more than three dimensions.

566: Array variable referring to composite data item declared with more than 1 dimension.

567: Amount of storage required for initial values in declaration exceeds size of variable being declared.

568: A variable contained in a structure has been given the "DEFINED hexadecimal constant," BASED, GLOBAL, or EXTERNAL attribute, or a variable that is EXTERNAL or contained in a BASED or DEFINED structure has been given the INITIAL attribute.

595: One variable has been DEFINED to another. At least one of the two is contained in a structure, but the two are not "brothers" within the same structure.

596: Useless test like IF BIN1 LT 0 (branch never) or IF BIN1 GE 0 (branch always).

597: Unlabelled PROCEDURE statement.

598: Missing END statements.

599: Compiler error.

codex

A Subsidiary of MOTOROLA INC.

CODEX CORPORATION
20 Cabot Boulevard
Mansfield, Massachusetts 02048

CODEX PHOENIX
INTELLIGENT TERMINAL SYSTEMS
2002 West 10th Place
Tempe, Arizona 85281
(602) 994-6580

Member of
IDCMA

Printed in U.S.A.