SOFTWARE SUPPORT MANUAL

TACPOL REFERENCE MANUAL

PROGRAMMING SUPPORT SYSTEM (PSS-B)

(TACFIRE)

(LITTON DATA SYSTEMS)
DAAB07-68-C-0154

PREPARED FOR U.S. ARMY COMPUTER SYSTEMS COMMAND FORT BELVOIR, VIRGINIA

15 January 1972

LIST OF EFFECTIVE PAGES

Insert latest changed pages; dispose of superseded pages in accordance with applicable regulations.

NOTE: On a changed page, the portion of the text affected by the latest change is indicated by a vertical line, or other change symbol, in the outer margin of the page. Changes to illustrations are indicated by miniature pointing hands. Changes to wiring diagrams are indicated by shaded areas.

Total number of pages in this manual is 85 consisting of the following.

Page No.	*Change No.	Page No.	*Change No.	Page No.	*Change No.
Title	0				
A	0				
B Blank	0				
i – ii	0				
1-1 – 1-2	0				
2-1 – 2-4	0				
3-1 – 3-4	0				
4-1 – 4-7	0				
4-8 Blank	0				
5-1 – 5-2	0				
6-1 – 6-8	0				
7-1 – 7-3	0				
7-4 Blank	0				
8-1 – 8-4	0				
9-1 – 9-3	0				
9-4 Blank	0				
10-1 – 10-2	0				
11-1 – 11-3	0				
11-4 Blank	0				
12-1	0				
12-2 Blank	0				
13-1 – 13-5	0				
13-6 Blank	0				
14-1 – 14-2	0				
A-1 – A-8	0				
B-1 – B-6	0				
C-1	0				
C-2 Blank	0				
D-1 – D-3	0				
D-4 Blank	0				
D-5	0				
D-6 Blank	0				
FO-1	0				
FO-2 Blank	0				

*Zero in this column indicates an original page.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1
TACPOL LANGUAGE FOR COMMAND AND CONTROL SYSTEMS

Section I. INTRODUCTION

1-1. General

The purpose of this document is to provide TACFIRE personnel with an introduction to the Tactical Procedure Oriented Language (TACPOL), which is specifically designed for use in developing the TACFIRE software. TACPOL is a modified subset of the PL/I language and incorporates:

a. A Communication Pool (Compool) capability

b. An embedded assembler language capability

1-2. The features and capabilities of the TACPOL language will be useful in developing and maintaining the TACFIRE software:

a. Application programs

b. Operating system

c. Compiler

d. Other programming aids

e. Maintenance and diagnostic programs

f. System exerciser and training evaluator

1-3. This document is intended to present sufficient information about TACPOL to enable one to begin writing TACPOL programs. Chapter 2 provides an introduction to the basic components and special terminology of the TACPOL language. Chapters 3 through 6 discuss data, expressions, and data declarations, and assignment statements. Chapter 7 discusses blocks, and chapter 8 discusses control statements. Procedures, data scope, arguments and parameters are discussed in chapters 9 through 11. Chapter 12 discusses condition declarations. Input/output is discussed in chapter 13, while files and Compool data are discussed in chapter 14.

1-4. There are four appendices included in the document. Appendix A provides a brief description of the intrinsic procedures which are part of the TACPOL language. Appendix B lists, and provides a brief explanation of, the particles (key words) which are a part of the TACPOL language. Appendix C is a table of integer precision and its use is described in chapter 3. Appendix D contains samples of compiler outputs such as a TACPOL source listing, an attribute and reference list, a machine language listing and a cross reference set-used listing.

Section II. TACPOL LANGUAGE IN THE TACFIRE SYSTEM

1-5. General

The heart of the TACFIRE system is the computer which is programmed to monitor and direct the functions of the TACFIRE system. These programs are written in the procedure oriented TACPOL language.

1-6. TACPOL is designed so that programs may be written for the TACFIRE computer with a minimum of programming effort, but other points were also considered in the design of the language. In higher level languages, such as TACPOL, it is easily possible for the programmer to lose sight of the true size of his program since one line of TACPOL code may generate numerous machine instructions. Therefore, since the size of programs in the TACFIRE system is important, TACPOL has been designed to restrict the use of those source language elements which could easily generate large volumes of machine language code. While the restrictions of any high level language provide some inconvenience to the

programmer, these 'restrictions' actually provide three distinct advantages. The first is that the programmer, while still procedure oriented (that is, thinking primarily about the problem for which he is writing a program, rather than the mechanics of the program itself), does not have to be concerned with implicitly generating large volumes of machine language code, a common concern in other procedure oriented languages. Secondly, the programmer is provided with a greater facility to implement the most efficient method of performing his required tasks according to his needs and desires. There are also few default provisions. Most attributes must be explicitly declared. This results in fewer errors because missing specifications are flagged at compile time, whereas default provisions often provide workable but erroneous code when the compiler assumes the programmer's intent.

CHAPTER 2
ELEMENTS OF THE TACPOL LANGUAGE

Section I. TACPOL LANGUAGE

2-1. General

This chapter presents a description of the basic concepts and special terminology of the TACPOL language.

2-2. Language Format

There is no fixed length format for input of TACPOL source statements. TACPOL statements may be written in free form in columns 1-80. The default option are card columns 2-72. The compiler recognizes the termination of a statement by the semicolon which must appear at the end of each statement e.g. A = Z;. Therefore, there may be several statements on one card, one statement to a card, or one statement extending over several cards. Because TACPOL may be written in free form and because of the nature of the language itself, the TACPOL program listing may serve as the program documentation. Thus, it is most advantageous to comment extensively and to format the source statements clearly and consistently. In addition, since a programmer defined name may be any length, highly descriptive names (often literally the name intended, such as ROUNDSAVAILABLE) may be chosen which will help make the meaning of the program more apparent. Figure F0-1 illustrates a TACPOL coding form.

2-3. Character Set

There are 50 characters in the TACPOL language. These include: the English letter alphabet of 26 characters; the ten Arabic numerals 0-9 and special characters in the following chart.

NAME	CHARACTER
plus sign	+
minus sign	-
asterisk	*
virgule	/
left parenthesis	(
right parenthesis	)
equal sign	=
point or period	.
comma	,
semicolon	;
colon	:
single quotation mark (apostrophe)	'
dollar sign	$
space (blank)	no character
other mark	any character other than the above which may only be used in character strings and comments

2-4. The colon may also be represented by two periods in sequence (..) and a semicolon may be represented by a comma followed by a period (,.). The representation ** denotes exponentiation (X**2 means X^2). Imbedded blanks are not permitted in any such character combinations. The following rules apply to the use of blanks (or spaces) in TACPOL:

a. No embedded blanks are permitted within any symbol except character strings or comment strings.

b. Blanks are permitted in character strings; however, they will be counted as part of the string.

c. Blanks are not permitted within bit strings.

d. A blank is required to separate any two adjacent symbols, when the first symbol begins

with a letter and the second symbol begins with a letter or digit.

e. Otherwise blanks may arbitrarily appear.

f. Whenever one blank is required or permitted to appear, any number of blanks are arbitrarily permitted to appear.

g. Whenever one blank is required or permitted to appear, comments are permitted to appear.

2-5. Comments

Comments are permitted wherever blanks are allowed or required in a program. They may be punched into the same cards as statements, inserted between statements, or appear in the middle of statements without affecting the program. The character pair /* indicates the beginning of a comment and the same characters reversed */ indicate its end. No blanks or other characters can separate these two characters: the virgule and the asterisk must be immediately adjacent. The comment itself may contain any characters acceptable to the hardware except the */ combination which would be interpreted as terminating the comment. Examples are shown below.

```
/* COMMENTS SUCH AS THIS */
/* MAY BE SPREAD OVER */
/* SEVERAL CARDS, HOWEVER IT
*/
/* IS RECOMMENDED THAT /**/
/* PAIRS BE PUT ON EACH CARD
*/
    TO AVOID UNINTENTIONAL
    ERRORS --
    FOR EXAMPLE IF THIS LAST
    CARD
    IS REMOVED */
```

Section II. CONCEPTS AND ORGANIZATION

2-6. General

The purpose of this section is to familiarize the reader with the terminology and concepts of the TACPOL language.

a. Values. The basic unit of information in TACPOL is the value. There are four types of values: short numeric, long numeric, character string, and bit string. Short numeric and long numeric values are binary representations of numbers or numeric data. Short numeric values are represented in 31 bits or less, and long numeric values are represented in 62 bits or less, not including the sign. A character string value is a binary representation of a group of ASCII coded characters. A bit string value is a sequence of bits.

b. Literals. A literal is an explicit ('literal') representation of a value.

c. Quantities. Values are assigned to and held in quantities. A quantity can yield (produce a copy of) the value it is currently holding. Therefore a quantity is a unit of storage for values. There are four types of quantities corresponding to the four types of values: short numeric, long numeric, character string, and bit string. A quantity of a given type may be assigned, hold, or yield only values which are of the same type as the quantity itself. For example, a character string quantity may be assigned, hold, or yield only character string values. Furthermore, the type of a quantity and the type of a value are explicitly defined and no automatic conversions are performed to ensure that the types match during assignment. Attempts to store character string values, for example, in a bit string quantity will result in an error at compilation time. There are ways of redefining attributes from one type to another. See Redefinition Attribute Procedures in Appendix A.

d. Symbols. Symbols are the elementary constituents of TACPOL. A symbol is a single character or set of characters which have the effect of a single character. Operators, particles and identifiers are comprised of symbols.

e. Operators. Operators are symbols which specify operations to be performed on values. The arithmetic operators consist of the multiply and divide operators * and /, the addition and subtraction operators + and -, the exponentiation operator **, and the arithmetic operators for long operations (*), and (**). The relational operators are EQ (or =), NE (or ¬ =), GT (or >), GE (or > =), LT (or <) and LE (or < =). The logical operators are AND (or &), OR (or |) and NOT (or ¬). The string operators are CAT (or ||) and SUBSTR (or $).

f. Particles. Particles are symbols which are the 'key words' of the TACPOL language. The particles are listed and briefly defined in Appendix B. Examples of particles are: GOTO, DO, IF, CALL, READ, and WRITE.

g. Identifiers. Identifiers are symbols which serve as names for programmer defined quantities. An identifier can be from one to any number of characters in length. However, if more than eight characters are used for an identifier, the compiler will use only the first five and the last three characters for the identifier. Care must be exercised not to produce a compiler error for duplicate identifiers.

EXAMPLE: UNDERRATE
UNDERSTATE

If both names in the example were to be used as identifiers, the first five and last three characters would be obtained by the compiler. Both would produce UNDERATE and a duplicate identifier would be detected by the compiler.

Rules of identifiers:

(1) The first character of an identifier must be a letter.

(2) Identifiers may not contain imbedded blanks.

(3) No identifier may be identical to any of the reserved words shown in table 2-1.

h. Declarations, Attributes and Names. Declarations, attributes and names are defined in the following paragraphs.

(1) Declarations associate names with and/or define characteristics of quantities, sets of quantities, values, procedures and conditions.

(2) Attributes describe the characteristics of the data to be used. These are numeric (BIN FIXED), character (CHAR), bit (BIT) and value (INIT).

(3) Names are used to identify procedures, points and quantities within the program to which reference is made in TACPOL statements. All names must be defined by some means within a program.

i. Sequence of Execution, Transfer of Control, and Invocation. The order in which the statements of a program are written specifies the sequence of execution of those statements. When a statement specifies that control is to 'jump' to some point in the sequence of execution other than the statement immediately following in line, then control is said to be 'transferred', and the point to which it is transferred must be denoted by a name (a 'point name'). A procedure is said to invoke (call forth) another procedure by transferring control to the first instruction of the second procedure. This invoked procedure will later return control to the invoking procedure at the conclusion of the operation of the invoked procedure.

Table 2-1. TACPOL Reserved Word List

ABS	DO	LETTER	REP
ACOS	E	LN	RETURN
ALIGNED	ELSE	LOAD	REWIND
AND	END	LOG	REWRITE
ASIN	ENDFILE	LONG	ROUND
ATAN	ENTRY	LT	S
B	EXP	MAX	SCALE
BACK	FILE	MIN	SHORT
BEGIN	FIXED	MOVE	SIGN
BIN	FOFL	NE	SIN
BIT	FROM	NOKEY	SPACE
BOOL	GE	NOPART	SQRT
BY	GOTO	NOT	SUBSTR
CALL	GT	OLD	SWITCH
CAT	IF	ON	THEN
CELL	IGNORE	OPEN	TO
CHAR	INIT	OR	TRUNC
CHECK	INPUT	OUTPUT	UNWIND
CLEAR	INTO	PACKED	UPDATE
CLOSE	KEEP	PASS	VALUE
CODE	KEY	PROC	WAIT
COS	L	QTRN	WHILE
DCL	LABEL	READ	WRITE
DELETE	LE	REM	ZDIV
DIGIT			

20083-1

2-7. Program Structure

A TACPOL program system has certain constituents. Values from quantities and literal values are combined to form expressions. Particles and expressions are combined to form larger expressions and statements. Statements are combined to form procedures and blocks. Procedures, blocks and other statements may be contained in another procedure or block. A procedure which is not contained in any other procedure is a program. One or more programs form a programmed system.

2-8. Language Structure

The elements described above are used to construct programs in TACPOL. The language is divided into two major categories:

a. Declarations. Used to associate names with quantities, values and procedures, to define characteristics for quantities, values, procedures and conditions, and to determine their scope of definition. In TACPOL the scope of definition is where a name is known and usable. All program declarations are made prior to the execution of program statements.

(1) *Data declarations*

(a) Simple scalar

(b) Simple array

(c) Group

(d) Table

(e) Cell

(f) Value

(2) *Procedure declarations*

(a) Proper procedures

(b) Function procedures

(3) *Condition declarations*

(4) *File declarations*

b. Statements. Used to specify the execution of the operations permitted in TACPOL.

(1) *Process statements*

(a) Assignment

(b) CALL

(c) GOTO

(d) IF

(e) DO

(f) Null

(2) *Input/output statements*

(a) OPEN

(b) CLOSE

(c) READ

(d) WRITE

(e) REWRITE

(f) DELETE

(g) SPACE

(h) REWIND

(i) UNWIND

(j) WAIT

(k) LOAD

(l) ON

(3) *Blocks*

(a) BEGIN

(b) CODE

CHAPTER 3
DATA TYPES

Section I. NUMERIC AND STRING DATA

3-1. General

There are two general types of data in TACPOL: numeric and string. Numeric data are fixed point representations of numbers. String data are sets of alphanumeric and special characters or patterns of binary digits.

3-2. Numeric Data

A numeric quantity is one which holds a number. In TACPOL, fixed point decimal numbers are converted and manipulated as their binary fixed point equivalents. For example, the decimal number 6.5 will be represented in fixed point binary as 110.1. If the number 6.5 were used as a literal in an expression, TACPOL would reserve a specific number of bits for the number and assume a binary point. The total number of bits reserved excluding the sign is called the precision. The scale factor represents the number of binary places the binary point is shifted from its assumed position immediately to the right of the rightmost bit. A positive scale factor represents a binary point shift to the left and a negative scale factor represents a binary point shift to the right. Thus, the mixed decimal number 13.25 can be represented in fixed point binary as 1101.0100000, which has a precision of 11 and a scale factor of +7.

a. Values in which the binary point falls outside the precision can also be represented. For example:

(1) 0.00111 appears as 111 when the precision is 3. The scale factor represented (in this case) is 5.

(2) 11100 appears as 111 when the precision is 3, and the scale factor represented (in this case) is -2.

b. When declaring numeric quantities it is desirable to know what the range of the binary values of the quantity will be. To determine the precision and scale factor of the fixed point binary equivalent of any fixed point decimal such as 1,300.333:

(1) Multiply 3.322 by the number of decimal digits to the right of the decimal point (which is 3 in the example) and round the product (9.966) to the next integer (10) resulting in the scale factor 10.

(2) From the table of integer precision (refer to Appendix C table C-1) find the number of bits required to contain the number to the left of the decimal point (1300 in the example); add the number of bits required from table C-1 (11) to the scale factor (10), resulting in the precision 21 bits, which is required to contain the entire number.

c. Short Numeric Data. Short numeric quantities are those with a precision of 1 to 31 bits (inclusive) and a scale factor between -127 and +127.* The maximum decimal number which may be represented in short numeric form is 2,147,483,647.

d. Long Numeric Data. Long numeric quantities are those with a precision of 32 to 62 (inclusive) and a scale factor between -127 and +127.**

3-3. String Data

A string is an ordered sequence of characters or bits that is treated as a single value. The length of the string is the number of characters or bits it contains.

a. Character String Data. A character string can include any digit, letter, or special character

**In the TACPOL implementation a precision of 8 will be assigned when a precision of 1 to 7 is specified; a precision of 15 will be assigned when a precision of 8 to 15 is specified and a precision of 31 will be assigned when a precision of 16 to 31 is specified. When a precision of 1 to 7 is specified, the field is treated as an eight bit unsigned quantity.*

***In the TACPOL implementation a precision of 62 will be assigned when a precision of 32 to 62 is specified.*

acceptable to the hardware. Any blanks or comments included in the character string are considered an integral part of the string and such characters are included in the count of the length. The maximum length character string allowed is 512.

b. Bit String Data. Bit strings can include only ZEROS and ONES. No blanks or comments are allowed in bit strings. The maximum length bit string permitted is 32.

3-4. Attributes of Data Types

When a quantity is declared, an attribute must be assigned which describes the characteristics of the data. Therefore, short numeric, long numeric, character and bit data types each have an attribute which describes the data the quantity is to hold.

a. The attribute which describes short numeric data is BIN FIXED. The BIN (binary) attribute specifies that binary representation of decimal values is intended. The FIXED attribute specifies that the variable is to represent fixed point data items.

b. The attribute which describes long numeric data is also BIN FIXED. Long numeric merely specifying a numeric quantity which will occupy from 32 to 62 bits.

c. The attribute CHAR is used to describe a quantity that is to hold character string data. A quantity thus defined can only be used in character string operations.

d. The attribute BIT is used to describe a quantity that is to hold bit string data. A quantity thus defined can only be used in bit string operations.

3-5. Literal Notation

There are many instances when the explicit notation of a value has to be used in coding a problem. The explicit notation is known as a literal. Each of the data types can be expressed as a literal in one of the available declarations and in many of the available statements.

a. Short Numeric Literal. To express a short numeric literal the value is written as a decimal number. The literal may be an integer, a fraction or a mixed number (part integer, part fraction). A decimal point must always be followed by a number. Examples of short numeric literals:

```
4753
.325
56.7
```

b. Long Numeric Literal. A long numeric literal is written in the same manner as a short numeric literal except that the letter L must follow the literal. Examples of long numeric literals:

```
6576213L
.00868L
45.775L
```

c. Character String Literal. A character string literal is expressed by writing out the desired characters with the string enclosed in single quotes. Any allowable character may appear within the string. A maximum of 512 characters may appear in a single character string. Examples of character string literals:

```
'ABCDE'
'FFA 45'
'**('
```

d. Bit String Literal. A bit string literal is expressed as a series of binary digits (ONEs and ZEROs), enclosed in single quotes and followed by the letter B. A maximum of 32 binary bits can appear in a bit string. Examples of bit string literals:

```
'10001'B
'1'B
'00000000000000000000000000000011'B
```

Section II. ALLOCATION OF QUANTITIES

3-6. General

The allocation of quantities is a function of the quantity type, its size, and whether it is aligned or packed.

3-7. Aligned Quantities

A quantity is said to be aligned when it is allocated storage such that the time required to access that quantity is minimized. In TACPOL, simple scalars and arrays are normally aligned. An override is available for arrays to change the allocation to packed. Table 3-1, on the following page, illustrates the allocation of aligned quantities by quantity type (short numeric, long numeric, character and bit).

3-8. Packed Quantities

A quantity is said to be packed when it is allocated storage such that the use of internal storage is minimized. In TACPOL, groups, tables and cells are normally packed. An override is available to change each of these allocations to aligned. Table 3-2, on the following page, illustrates the allocation of packed quantities by quantity type.

Table 3-1. Allocation of Aligned Quantities

Type	Precision or Length	Aligned to Next	Size of Allocation	Justified	Remainder Filled With	Access Class
A. SIMPLE SCALARS						
Short	1 - 31	Fullword	Fullword	Right	Sign bits	Fullword
Long	32 - 62	Fullword	2 Fullwords	Right	Sign bits	Doubleword
Bit	1 - 32	Fullword	Fullword	Left	Zero bits	Fullword
Char	1 - 512	Fullword	n Fullwords	Left	Char. blanks	Multiword
B. GROUP SCALARS, TABLE SCALARS						
Short	1 - 7	Quarterword	Quarterword	Right	Sign bits	Halfword
Short	8 - 15	Halfword	Halfword	Right	Sign bits	Halfword
Short	16 - 31	Fullword	Fullword	Right	Sign bits	Fullword
Long	32 - 62	Fullword	2 Fullwords	Right	Sign bits	Doubleword
Bit	1 - 16	Halfword	Halfword	Left	Zero bits	Halfword
Bit	17 - 32	Fullword	Fullword	Left	Zero bits	Fullword
Char	1 - 2	Halfword	Halfword	Left	Char. blanks	Halfword
Char	3 - 512	Fullword	n Fullwords*	Left	Char. blanks	Multiword
C. SIMPLE ARRAYS, GROUP ARRAYS, TABLE ARRAYS						
Short	1 - 7	Quarterword	Quarterword	Right	Sign bits	Quarterword
Short	8 - 15	Halfword	Halfword	Right	Sign bits	Halfword
Short	16 - 31	Fullword	Fullword	Right	Sign bits	Fullword
Long	32 - 62	Fullword	2 Fullwords	Right	Sign bits	Doubleword
Bit	1 - 16	Halfword	Halfword	Left	Zero bits	Halfword
Bit	17 - 32	Fullword	Fullword	Left	Zero bits	Fullword
Char	1 - 2	Halfword	Halfword	Left	Char. blanks	Halfword
Char	3 - 4	Fullword	Fullword	Left	Char. blanks	Fullword
Char	5 - 512	Fullword	n Fullwords*	Left	Char. blanks	Multiword

NOTES:

* n = (number of characters +3)/4

20083-2

Table 3-2. Allocation of Packed Quantities

Type	Precision or Length	Aligned to Next	Size of Allocation	Justified	Remainder Filled With	Access Class
A. GROUP SCALARS, TABLE SCALARS						
Short	1-7	Quarterword	Quarterword	Right	Sign bits	Quarterword
Short	8 - 15	Halfword	Halfword	Right	Sign bits	Halfword
Short	16 - 31	Fullword	Fullword	Right	Sign bits	Fullword
Long	32 - 62	Fullword	2 Fullwords	Right	Sign bits	Doubleword
Bit	1 - 32	Bit*	n Bits [1]	Left	Zero bits	Packed Bit
Char	1 - 512	Byte	n Bytes [1]	Left	Char. blanks	Multiword
b. SIMPLE ARRAYS, GROUP ARRAYS, TABLE ARRAYS						
Short	1 - 7	Quarterword	Quarterword	Right	Sign bits	Quarterword
Short	8 - 15	Halfword	Halfword	Right	Sign bits	Halfword
Short	16 - 31	Fullword	Fullword	Right	Sign bits	Fullword
Long	32 - 62	Fullword	2 Fullwords	Right	Sign bits	Doubleword
Bit	1 - 16	Bit**	n Bits [1]	Left	Zero bits	Packed Bit
Bit	17 - 32	Fullword	Fullword	Left	Zero bits	Fullword
Char	1 - 512	Byte	n Bytes [1]	Left	Char. blanks	Multiword

NOTES:

* If the next field would cross a fullword boundary, that field is aligned to the fullword boundary.

** If an element in the array would cross a fullword boundary, that element is aligned to the start of the next fullword.

[1] n = length.

20083-3

CHAPTER 4
DECLARATIONS

Section I. SCALAR AND ARRAY DECLARATIONS

4-1. Scalar Declarations

Scalar declarations define a name, or a list of names, and assigns a literal type to the name(s). The literal types assigned are short numeric, long numeric, character or bit.

a. Short Numeric Scalars. Short numeric scalars define a quantity whose binary representation of a decimal value will occupy 31 bits or less.

EXAMPLE: DCL NUMB1 BIN FIXED (31,2);

(1) All declarations (except the condition declaration) start with the particle DCL (declare). Following the particle is an identifier (name) selected by the programmer which identifies the quantity. Following the identifier is the attribute for short numeric quantities, BIN FIXED. The attribute is followed by the precision and scaling specification enclosed in parentheses. The first number states the total number of bits the quantity is to occupy. The second number specifies the number of fractional bits. The numbers are separated from one another by a comma. In the above example, the quantity NUMB1 is defined as a short numeric quantity, 31 bits in length of which 2 bits are for fractional representation. Note that according to the rules of blanks, a space must always be used to separate the particle DCL, the quantity name, and the attributes BIN and FIXED. A blank space may or may not be used following the attribute for the precision and scaling specification.

(2) All declarations and all statements in TACPOL are terminated by a semi-colon. Two default options are available for short numeric scalars. If the precision and scaling specification is omitted from the declaration it will be assumed 31,0. The quantity is assumed to be 31 bits in length with no fractional bits (see the example below).

DCL NUMB2 BIN FIXED;

(3) The other default option available is to include the precision specification and omit the scaling specification as in the example below.

DCL NUMB3 BIN FIXED (15);

The precision specification in the above example specifies a quantity 15 bits in length. Since the scaling specification has been omitted it is assumed to be ZERO, therefore no fractional bits are assigned the quantity. All other attributes must be specifically stated.

b. Long Numeric Scalars. A long numeric scalar is essentially the same as a short numeric scalar with the exception that the precision specification will be not less than 32 and not greater than 62.

EXAMPLE: DCL FOX BIN FIXED (47,8);

(1) In the above example the quantity FOX is defined to be 47 bits in length of which eight bits are for a fractional portion. The particle and the attribute are the same as for short numeric. The only default option available is the scaling specification. If it is omitted it is assumed that no fractional bits will be assigned to the quantity.

EXAMPLE: DCL BEAR BIN FIXED (52);

(2) In the above example all the bits specified by the precision specification are integer bits.

c. Character Scalars. A character scalar will have the CHAR (character) attribute and a length specifier.

EXAMPLE: DCL C CHAR (51);

The declaration specifies that the quantity with the identifier C is to be a character string quantity 51 characters in length. A character string is assigned from left to right. If a character string value longer than the declared length of the quantity is assigned to the quantity, the excess characters are truncated on the right. If shorter, the value on the right is padded with blank

spaces. The maximum length character string quantity allowed is 512 characters.

d. Bit Scalars. Bit scalars have the BIT attribute and a length specifier.

EXAMPLE: DCL B BIT (15);

The quantity with the identifier B is specified as a bit string quantity 15 bits in length. Like character strings, bit strings are assigned to quantities from left to right. If a string is long that the length declared for the quantity, the rightmost bits are truncated; if shorter, padding on the right is with ZERO bits. The maximum length of a bit string quantity allowed is 32 bits.

e. Scalar Lists. It is possible to code a scalar declaration with more than one identifier appearing in the declaration.

EXAMPLE: DCL (FOX, BEAR, RABBIT) CHAR (4);

In the above example three identifiers appear in the declaration. This is an identifier list and must appear enclosed in parentheses. All three identifiers are assigned the same attribute, character, and the same length (four characters). As many names as necessary may appear in the list. A scalar declaration containing lists of identifiers may be used for short numeric, long numeric, character and bit operations. All names within the lists will have the same attributes. Each identifier in the list is separated from the following identifier by a comma.

4-2. Array Declarations

An array declaration is used to assign a name to a quantity when the quantity is a block of storage. The declaration gives the block storage area dimension. As such, one dimensional, two dimensional and three dimensional arrays can be declared. The declaration specifies the name, storage area length and the length of the data.

a. One Dimensional Array. A one dimensional array is declared as in the example below:

EXAMPLE: DCL BROWN (5) BIN FIXED (24,12);

The particle DCL is followed by an identifier selected by the coder. In the example above the array name is BROWN. Following the identifier is a subscript, enclosed in parentheses, which contains the number of quantities within the array. Following the subscript is an attribute. In the above example the short numeric attribute is used. Long numeric, character and bit attributes can also be used. Thus, the above declaration declares the one dimensional array named BROWN, containing five quantities, each quantity is short numeric, 21 bits in length, 12 of the bits are fractional. When a quantity has been defined that has dimension, the name of the quantity can be subscripted. Later on in a program, when array BROWN is referenced by TACPOL statements, individual quantities within the array can be specified such as BROWN(1), BROWN(2), BROWN(3), etc. There is no limit to the number of quantities which can be included in an array except that of the computer storage physically available to hold the information.

b. Two Dimensional Array. A two dimensional array is a set of one dimensional arrays. A two dimensional array is coded similar to a one dimensional array with the addition of a second dimension in the subscript that follows the array name.

EXAMPLE: DCL BLACK(6,3) BIT(32);

In the above example an array named BLACK has been declared with an attribute of BIT. Each quantity within BLACK can hold a bit string of 32 bits. The subscript indicated there are six quantities of BLACK in the first dimension and three quantities of BLACK in the second dimension. The total number of quantities in the array is 18. To obtain this number multiply the number of the first dimension by the number of the second dimension. Figure 4-1 illustrates how the array BLACK would be apportioned in internal memory. Note that dimension two is cycled through completely before dimension one is stepped by one. Later on in a program, when array BLACK is referenced by TACPOL statements, individual quantities within the array may be specified by following the array name with a subscript containing the desired quantity, i.e. BLACK(1,1), BLACK(2,3), BLACK(3,3), etc.

c. Three Dimensional Array. Another dimension may be added to an array declaration to form a three dimensional array.

EXAMPLE: DCL BLUE(4,3,2) BIN FIXED(31,0);

In the above example the declared array, BLUE, is a three dimensional array because three values appear within the subscript. BLUE will contain room for 24 short numeric quantities (dimension one times dimension two times dimension three). Three dimensions are the maximum allowable within an array declaration. Figure 4-2 illustrates

how the array BLUE would be allocated in internal memory. For three dimensional arrays, dimension three is cycled through completely before dimension two is stepped by one and dimension two is cycled through completely before dimension one is stepped by one. Later on in a program, when array BLUE is referenced by TACPOL statements, specific quantities within the array may be referenced through the array name followed by an appropriate subscript; i.e. BLUE(1,2,1), BLUE(3,3,2), BLUE(4,1,2), etc.

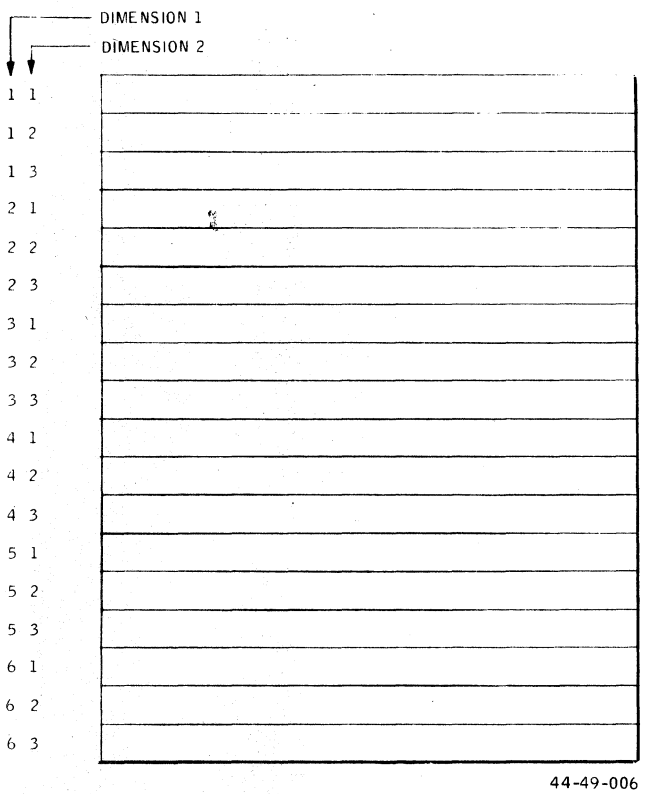


Figure 4-1. Two Dimensional Array

4-3. Array Allocations

When arrays are allocated they are established according to the rules of aligned quantities (see chapter 3). The coder may change the normal aligned allocation for arrays to a packed allocation by using the packed option in the array declaration.

EXAMPLE: DCL GREEN(5,5) CHAR(4) PACKED;

In the above example, the allocation of the two dimensional array GREEN is changed from the normal aligned allocation to a packed allocation because the PACKED option has been specified in the declaration. When used, the particle PACKED must follow the attribute in the declaration.

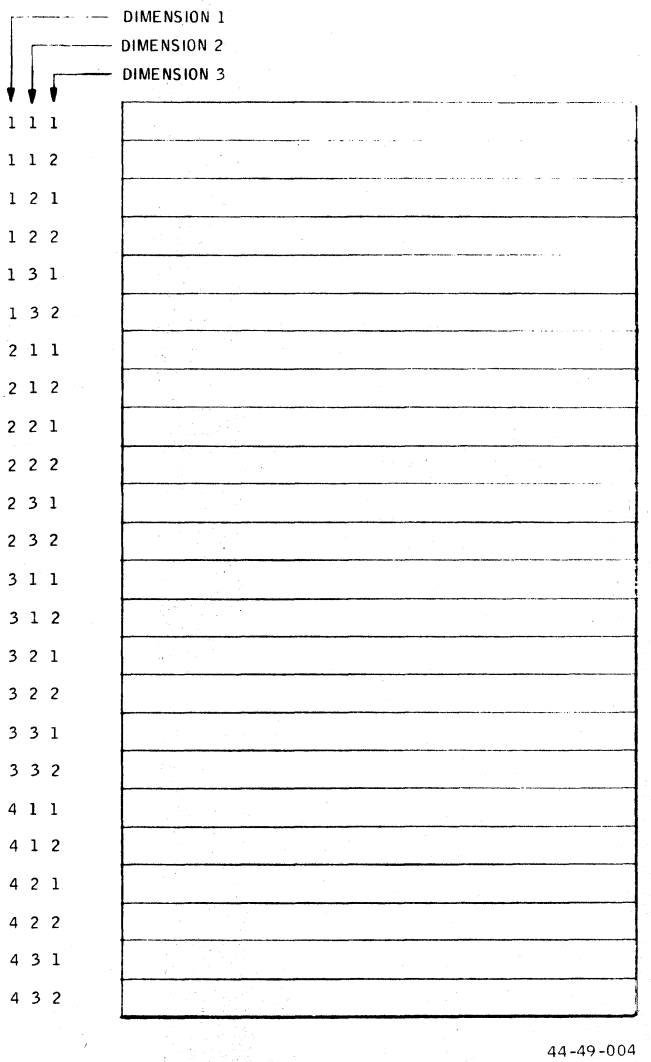


Figure 4-2. Three Dimensional Array

Section II. GROUP, TABLE AND CELL DECLARATIONS

4-4. General

Collections of quantities may be declared by group, table or cell declarations. Quantities may be grouped and declared under a single operation in a group declaration. Identical structures, with dimension, may be declared in a table declaration. Cell declarations are used to declare different quantities which occupy the same storage

areas to conserve storage. Group, table and cell declarations are made using two or more levels. The levels serve to define the name of the group, table or cell and then to define the specific quantities within the group, table or cell.

4-5. Group Declarations

Group declarations make it possible to define a group of names under the heading of a single name. The declaration contains two levels. The first level is the definition of the group name. The second level contains a list of names (identifiers) all of which will belong to the group.

a. Group Definition. In the example below, the first level is specified by the number one and the group name, OBB, is declared. The second level is specified by the number 2 and on this level all quantities of the group are declared. Since level two is considered to consist of a list of quantities, the list must be enclosed in parentheses. The list may contain all scalars, all arrays or a mixture of both (the example shows two scalars, one one dimensional array and one two dimensional array). Attributes may also be mixed within the list (the example shows three BIT attributes and one short numeric). The scalars and/or arrays declared in the list are separated from one another by a comma. The group declaration is terminated by closing level 2 (right parenthesis) followed by a semi-colon.

```
EXAMPLE:   DCL   1 OBB,
                 2(P23 BIT(32),
                   LLI BIN FIXED,
                   Z15(5) BIT(32),
                   Z24(4, 2) BIT(32));
```

In the group declaration five declarations have actually been made. The group name OBB has been declared, the two scalars P23 and LL1, and the two arrays, Z15 and Z24. Figure 4-3 illustrates how the group will be allocated in internal memory. Note that storage is assigned in the order in which the declarations have been made. Later on in a program, when TACPOL statements make references to the declared quantities, the entire group can be referenced by using the group name, the scalars can be referenced by using the scalar names and the arrays can be referenced by using the array names with appropriate subscripts. When using the group name a single subscript may follow the name to denote an individual quantity, i.e. OBB(2), OBB(6), etc. The scalar names may not be subscripted (they do not have dimension).

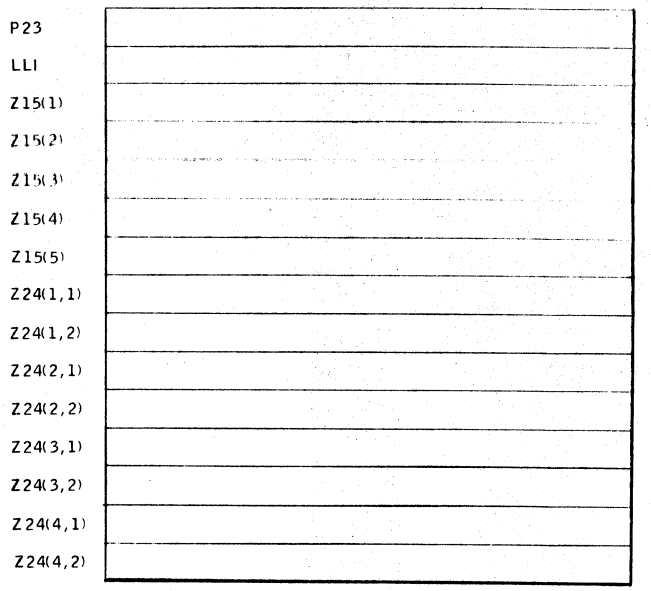


Figure 4-3. Group Layout

b. Group Allocation. Group declarations normally have an allocation of packed. All level 2 quantities assume the allocation of the level 1 identifier. Thus, all the quantities within the previous example have an allocation of packed even though arrays are normally aligned (See Chapter 3). It is possible to change the allocation for a group declaration by using the ALIGNED option.

```
EXAMPLE:   DCL   1 METAL ALIGNED,
                 2 (STEEL CHAR (6),
                    IRON CHAR (2),
                    TIN (10) BIT (32));
```

By using the available ALIGNED option in the above example all level 2 quantities are now assigned an allocation of aligned. The option must be specified on level 1 immediately following the group name. The option may never be specified with any level 2 declaration. The limit of the quantities that can be included in a group declaration is the computer storage physically available to hold the information.

4-6. Table Declarations

Table declarations are similar to group declarations when it comes to coding but table declarations use a repeat factor which gives the table dimension.

a. Levels. Table declarations contain two levels; the first level declaring the table with the repeat factor and the second containing a list of names, all of which will belong to the table. The list of names may contain scalars, arrays, or both.

```
EXAMPLE:   DCL  1 EMPL(3),
                2 (ARGU BIN FIXED,
                   STAT BIN FIXED,
                   PI2(4) CHAR(4));
```

On the first level, specified by the number 1, table EMPL is declared along with a repeat factor. The repeat factor stipulates how many times the scalars and/or arrays within the table declaration are to be repeated. In the example the two scalars will each be repeated three times and the array will also be repeated three times. Figure 4-4 illustrates how this table will be allocated in internal memory. Note that the table will be comprised of three parts referred to as EMPL(1), EMPL(2) and EMPL(3). Each part is identical in makeup consisting of the scalar ARGU (ARGU(1), ARGU(2) and ARGU(3)), the scalar STAT (STAT(1), STAT(2) and STAT(3)) and the array P12. When an array appears in a table declaration the repeat factor adds a dimension to the array with the repeat factor becoming dimension one. Thus, when an array appears in a table declaration a one dimensional array becomes a two dimensional array (as in this example), a two dimensional array becomes a three dimensional array and a three dimensional array becomes a four dimensional array. Later on in a program, when TACPOL statements reference the declared quantities, the entire table can be referenced by the table name, a part of the table can be referenced by subscripting the table name, or individual scalars and arrays can be referenced by their names with appropriate subscripts. Note that attributes can be mixed within the declaration (short numeric and character are illustrated) and that the coding of the table declaration follows the same rules as for group declarations.

b. Table Allocation. A table normally has an allocation of packed. All level 2 quantities assume the allocation of the table. ARGU, STAT, and PI2 in the previous example all have an allocation of packed. This allocation can be changed by using the ALIGNED option which is available to table declarations.

```
EXAMPLE: DCL  1 TABLESOF (2) ALIGNED,
              2 (LOGS (10) BIN FIXED,
                 SINES (10) BIN FIXED,
                 COSINES (10) BIN FIXED,
                 TANGENTS (20) BIN
                 FIXED);
```

In the above example the ALIGNED option appears on level 1 immediately following the repeat factor. All level 2 quantities now assume the allocation specified on level 1. An option may never be specified with any level 2 declaration. The limit of the quantities that can be included in a table declaration is the computer storage physically available to hold the information.

4-7. Cell Declarations

Cell declarations contain lists of quantities, in scalar, array, group or table form, which are allocated common storage. In effect, cell declarations are used to overlay areas of storage to conserve space. This common memory storage factor is unique to the cell declaration.

a. Levels. A cell declaration consists of 2 or 3 levels. The first level specifies the cell name and the cell declaration. The second level specifies an array, scalar, table or group declaration. The third level specifies the quantities of a table or group if level two is a table or group declaration.

```
EXAMPLE: DCL 1 MEMBR CELL,
             2 (ARMM(4),
             3 (GEL BIT(32),
               EETO CHAR(4),
               WARN(3), BIN FIXED(31,10)),
               DISCO,
             3 (LIGN BIN FIXED(62,8),
               PANEL(2,4) BIT(32)));
```

In the above example, the first level (specified by the number one) declares a cell by the name of MEMBR. The particle CELL must always immediately follow the cell name. Level two (specified by the number two) starts a list of names so a parentheses must be opened. In the above example the table ARMM is declared on level 2. A table can consist of scalars and/or arrays; so on level three the constituents of the table are listed. The number 3 specifies the third level and a parentheses is opened to start the list. The table consists of the scalars GEL and EETO and the array WARN. Following the definition of WARN the third level is closed by a right parenthesis which puts the declaration back on level 2. The group DISCO is now declared on the second level. Following this declaration are the constituents of the group which are declared on level three. Since level three was closed, the number three must appear again denoting the start of level three and the parentheses must be opened. The scalar LIGN and the array PANEL are declared on this

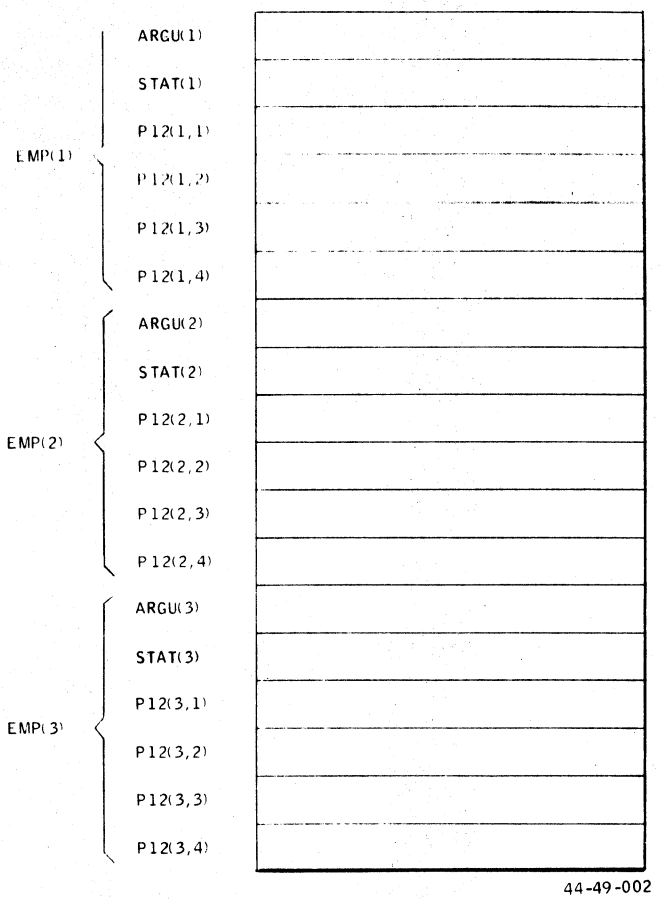


Figure 4-4. Table Layout

level as being part of group DISCO. When the declaration is concluded, both levels three and two must be closed by right parentheses. The effect of the cell declaration is to overlay all level two quantities. Thus, the table ARMM and the group DISCO share the same starting internal memory address. An illustration of the overlay is presented in figure 4-5. Table ARMM occupies more space than group DISCO so only part of internal memory is actually overlayed. Later on in a program, when TACPOL statements reference the declared quantities, the table MEMBR, or any part of MEMBR, the scalars GEL and ETOO, and the array WARN may be referenced by name and an appropriate subscript. The group DISCO, or any part of DISCO, and the array PANEL may be referenced by name and appropriate subscript, and the scalar LIGN may be referenced by name. The cell name is *never* referenced by name in any statement. The cell name is used by the compiler as a definition of a common storage area. A cell declaration need not contain three levels.

The effect of the above example is to overlay a single word of internal memory with three scalars of different attributes. All the scalars are level two declarations and, as such, are overlayed. WATER can be used for arithmetic operations, COFFEE for character string operations and TEA for bit string manipulations.

```
EXAMPLE:   DCL   1 LIQUID CELL,
                 2 (COFFEE BIN FIXED,
                    TEA CHAR (4),
                    MILK BIT(32));
```

b. Cell Allocation. Cells normally have an allocation of packed. All level 2 and level 3 declarations assume the allocation of level 1. The allocation may be changed by using the ALIGNED option (see Chapter 3).

```
EXAMPLE: DCL   1 LIVE CELL ALIGNED,
               2 (REAL,
               3 (ACTUAL (7, 5) BIN
                  FIXED (62,0)),
                  SUBSIST (5),
               3 (DWELL BIN FIXED
                  (31,0),
                  RESIDE (8) BIN FIXED
                  (31,0)));
```

In the above example the group REAL and the table SUBSIST are overlayed. The quantities within the group and the table are aligned because the ALIGNED option appears on level 1 immediately following the particle CELL. The option may not appear next to any level 2 or level 3 declarations. The limit of the quantities that can be included in a cell declaration is the computer storage physically available to hold the information.

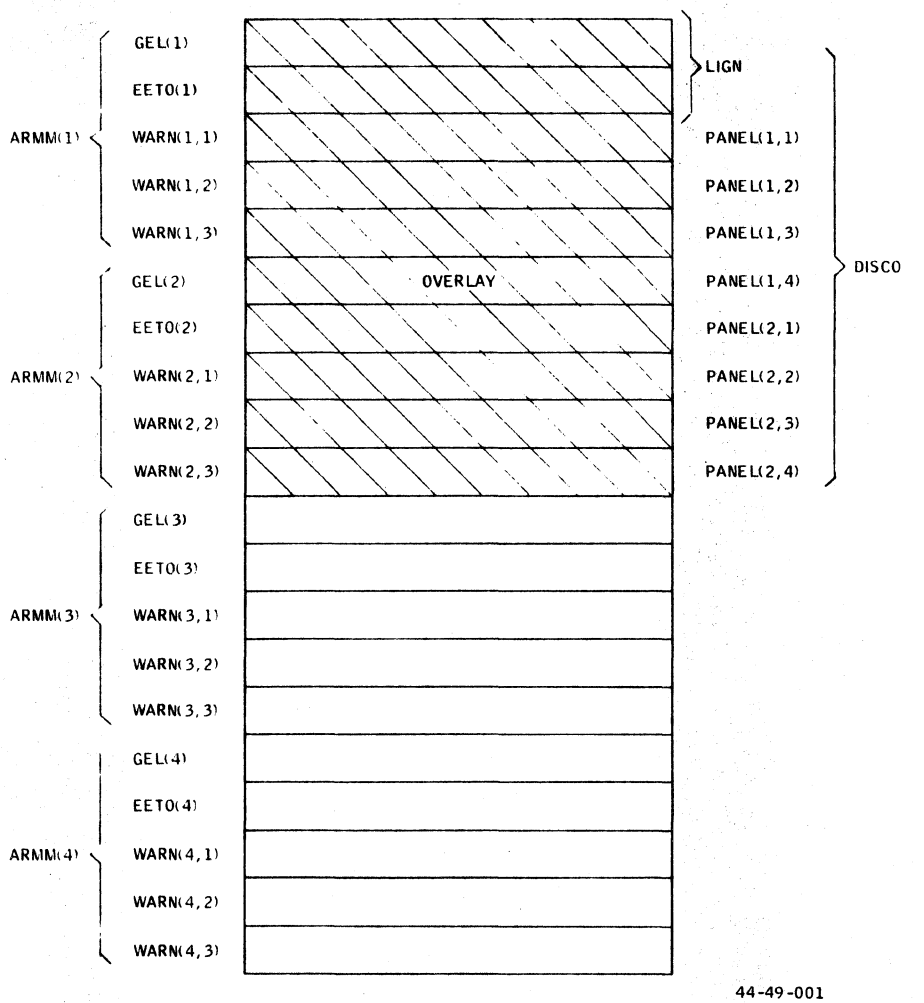


Figure 4-5. Cell Layout

Section III. VALUE DECLARATIONS

4-8. General

Scalar quantities may be assigned values at the same time as they are declared. When this occurs, the declaration is called a value declaration.

4-9. The value declaration requires the attribute INIT following the type attribute.

EXAMPLE: DCL APT2 BIN FIXED (15, 1) INIT (100.5);

Following the attribute INIT is the value, in correct literal form, enclosed in parentheses. In the above example, APT2 is not only declared as short numeric, but the value 100.5 is assigned to APT2. One very important rule of value declaration must always be followed. Once a name has been declared in a value declaration, the value assigned that name may never be changed.

a. Other examples of value declarations are:

(1) Long numeric:

DCL LNUM BIN FIXED (45, 5) INT (45007.125L);

(2) Character:

DCL CSTR CHAR (6) INIT ('X-Y473B'),

(3) BIT:

DCL BSTR BIT (7) INIT ('1110001'B);

b. Value declaration may not appear as part of group, table or cell declarations.

CHAPTER 5
ASSIGN STATEMENTS

5-1. General

Assign statements are used to assign values to identifiers which have been previously defined as a quantity other than a value quantity. The equal sign identifies an assign statement.

a. In the example below, A1 must have been defined previously as a short numeric quantity. In the assignment statement A1 is assigned the value of 50.

EXAMPLE: A1 = 50;

b. In the example below, AA and BB both must have been previously defined as the same quantity types. In the assign statement AA is assigned the value of BB.

EXAMPLE: AA = BB;

c. In the example below, GET4 must have been previously defined as a character quantity type. GET4 is assigned the character value AXEL. A value may be assigned to more than one quantity in an assign statement.

EXAMPLE: GET4 = 'AXEL';

d. In the example below AA, BB and CC are each assigned the value 200. The names are separated from one another by commas and they must have been previously defined as the same quantity types.

EXAMPLE: AA, BB, CC = 200;

5-2. Rules of Assignment

a. Short or Long Numeric Quantities. Short and long numeric values are right justified within quantities. Low order bits lost due to conforming to the quantity allocation of an identifier are truncated.

(1) In the example below, only one bit has been reserved for a fractional value. In the assign statement more than one fractional bit is specified in the value. The fractional bits that cannot fit into the allocated space of the quantity will be truncated and the value will actually become 203.5 when the assignment is made. A quantity which receives a value other than the desired value is said to contain an undefined value.

```
EXAMPLE:    DCL ALP BIN
            FIXED (10,1);
            ALP = 203.56;
```

(2) In the example below, the largest value that can be accommodated in 15 bits is +32,767. The assignment calls for the value of 40960 to be assigned to TEMP. Since the value is right justified the most significant bit of the value will be lost through truncation. The actual value of TEMP will be 8192. The value is said to be undefined.

```
EXAMPLE:    DCL TEMP BIN
            FIXED (15,0);
            TEMP = 40960;
```

(3) In the example below, no fractional bits have been reserved for P2A. The assignment calls for a mixed number. The integer will be right justified and the entire fraction will be lost through truncation.

```
EXAMPLE     DCL P2A BIN
            FIXED (35,0);
            P2A = 17295.763L;
```

(4) When values smaller than the allocation given a quantity are assigned, ZEROS are appended to the high and low order bit positions. In the example below, FLIP can hold a value much larger than the assigned value of 20.5. Leading ZEROS will be appended to the integer portion of the value and trailing ZEROS will be added to the fractional part. In binary FLIP would be assigned the value 00010100.10000.

```
EXAMPLE:    DCL FLIP BIN
            FIXED (12,4);
            FLIP = 20.5;
```

b. Character Quantities. Character strings are left justified within quantities. If the value assigned a character quantity is shorter than the declared length, space characters are appended to the right.

(1) In the example below, ALPHA can equal as much as seven characters. Only three characters have been assigned in the statement.

As a result ALPHA will be equal to BCD followed by the ASCII code for four blanks (ALPHA = BCDbbbb where b is an ASCII blank).

```
EXAMPLE:    DCL ALPHA CHAR (7);
            ALPHA = 'BCD';
```

(2) If the value assigned a character quantity is greater than the declared length, characters will be truncated from the right. In the example below, BETIC can be equated to a maximum of six characters. Method would be equated to BETIC and the characters ICAL would be truncated.

```
EXAMPLE:    DCL BETIC CHAR (6);
            BETIC = 'METHODICAL';
```

c. Bit Quantities. Bit strings are left justified with quantities. If the value assigned a bit quantity is shorter than the declared length, ZERO bits are appended to the right.

(1) In the example below, NUME is declared as eight bits in length and a five bit string is assigned. ZERO bits will be appended to the right to fill in the remaining bit positions. NUME has the following value after the assignment is made: 10011000.

```
EXAMPLE:    DCL NUME Bit (8);
            NUME = '10011'B;
```

(2) If a value assigned a bit quantity is greater than the declared length of the quantity, bits are truncated from the right. In the example below, RIC can be equated to a maximum of five bits in a string. The assignment calls for ten bits. The string 11011 is assigned to RIC and the least significant five bits, 11001, are truncated.

```
EXAMPLE:    DCL RIC BIT (5);
            RIC = '1101111001'B;
```

d. Value Declarations in Assignment Statements. A quantity that has been declared in a value declaration may never appear on the left side of an assign statement.

(1) The statement below, is in error and will so be indicated by the compiler. QUAN has had a value assigned through a value declaration. As such, the value will always remain with QUAN and cannot be changed through an assign statement.

```
EXAMPLE:    DCL QUAN BIT (4) INIT
            ('1000'B);
            QUAN = '0101'B;
```

(2) The quantity declared in a value declaration can appear on the right side of an assign statement. In the example below, DRAM is assigned the value of VALU. A blank character is appended to the right of DRAM since DRAM is one character longer than VALU.

```
EXAMPLE:    DCL VALU CHAR (3)
            INIT ('ZyX')
            DCL DRAM CHAR (4);
            DRAM = VALU;
```

CHAPTER 6
OPERATORS AND EXPRESSIONS

6-1. Operators

There are four types of operators in TACPOL: arithmetic, relational, string and logical. An operator that precedes an operand is a prefix operator, i.e., -A or +A. An operator that appears between operands is an infix operator, i.e., A+B or A-B.

a. Arithmetic Operators. The arithmetic operators are addition, subtraction, multiplication, division and exponentiation. The symbols for these operators are shown in table 6-1.

Table 6-1. Arithmetic Operators

SYMBOL	OPERATION
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

20083-4

When exponentiation is specified as the operator, it must be followed by an unsigned decimal number. Exponentiation only by positive integers is permitted.

```
EXAMPLE:    DCL DD BIN FIXED;
            DCL YY BIN FIXED;
            DD = 20;
            YY = DD**3;
```

In the above example, YY will contain 20^3 or 8000 as a result of exponentiation. Note that exponentiation must be described in consecutive characters without any imbedded blanks. The operators for addition and subtraction may be prefix or infix operators.

b. Relational Operators. The relational operators indicate a comparison of two values of the same type. That is, an arithmetic value may only be compared to another arithmetic value, a character string value may only be compared to another character string value and a bit string value may only be compared to another bit string value. The symbols for these operators are shown in table 6-2.

Table 6-2. Relational Operators

SYMBOL	OPERATION
EQ or =	equal
NE or ¬ =	not equal
LT or <	less than
LE or < =	less than or equal
GT or >	greater than
GE or > =	greater than or equal

20083-5

(1) The comparison of arithmetic values means a comparison of signed arithmetic values. To compensate for the fact that arithmetic values have different scale factors, the value having the smaller scale factor will have its binary point aligned with the value having the larger scale factor.

```
EXAMPLE:    DCL BBA BIN FIXED
            (9,1);
            DCL DDA BIN FIXED
            (9,3);
            BBA = 15.5;
            DDA = 10.125;
```

In the example BBA and DDA will each have the following binary values:

```
BBA = 00001111.1
DDA = 001010.001
```

BBA has the smaller scale factor and will be aligned with the scale factor of DDA. This involves shifting the value of BBA two binary positions to the left. The shifting for alignment of scale factors is an automatic feature of the TACPOL language. The comparison of the two aligned values is then made as follows:

BBA = 001111.100

DDA = 001010.001

Note the possibility of some significant bits being lost on the left if a large shift is involved in the operation. A short numeric operand may only be compared with another long numeric operand.

(2) When a comparison is made between character string values, the comparison is made, character-by-character, going from left to right.

EXAMPLE:
```
DCL EGO CHAR(4);
DCL ERGO CHAR (5);
EGO = 'SELF';
ERGO = 'HENCE';
```

In the example, if EGO and ERGO are compared, the first character of EGO (S) is compared with the first character of ERGO (H), the second character of EGO (E) is compared with the second character of ERGO (E), etc. Note the difference in length of the character string values given in the example. When this situation occurs, the shorter value is automatically extended on the right with character designations for a blank to the size of the larger value. The actual comparison made between EGO and ERGO is illustrated below.

EGO = SELFb (b = blank)

ERGO = HENCE

(3) When a comparison is made between bit string values, the comparison is made, bit-by-bit, going from left to right.

EXAMPLE:
```
DCL NTT BIT(6);
DCL PTT BIT(8);
NTT = '111001'B;
PTT = '10100111'B;
```

If NTT and PTT are compared, the first bit of NTT (1) will be compared with the first bit of PTT (1), the second bit of NTT (1) will be compared with the second bit of PTT (0), etc. When bit strings differ in length, as illustrated in the example, the shorter bit string value is automatically extended on the right with binary ZEROS to the size of the larger value. The actual comparison that would be made between NTT and PTT is illustrated below.

NTT = 11100100

PTT = 10100111

The result of any comparison operation is a bit string value, one bit in length, where the value of the bit will be ONE if the comparison is true or the value of the bit will be ZERO if the comparison is false.

c. String Operators. The string operators perform catenation or substring operations upon character strings and bit strings. The symbols for these operators are shown in table 6-3.

Table 6-3. String Operators

SYMBOL	OPERATION
CAT or 11	catenation
SUBSTR or $	substring

20083-6

(1) Catenation is the joining, or chaining, of strings into a single string. A bit string may only be catenated with another bit string and a character string may only be catenated with another character string.

EXAMPLE:
```
DCL (EBC,FAB) BIT(4);
DCL RECV BIT(8);
EBC = '1001'B;
FAB = '1100'B;
RECV = EBC CAT FAB;
```

In the example, the bit string specified by EBC is joined with the bit string specified by FAB and the result is left justified in RECV. RECV, as a result of the catenation, contains 10011100. If catenation produces a result less than the number of bits assigned the receiver, ZEROS are appended to the right of the result.

EXAMPLE:
```
DCL (BLUE,RED) BIT(3);
DCL GREEN BIT(8);
BLUE = '011'B;
RED = '101'B;
GREEN = BLUE CAT RED;
```

When BLUE is catenated with RED the result is 011101. However, GREEN is eight bits in length. The result is left justified in GREEN with two ZEROS appended to the extra bit positions yielding a final result of 01110100. If catenation pro-

duces a result greater than the number of bits assigned the receiver, bits are truncated from the right most bit positions.

```
EXAMPLE:    DCL DOCU BIT(4);
            DCL PAMP BIT(5);
            DCL BASK BIT(6);
            DOCU = '1001'B;
            PAMP = '11101'B;
            BASK = DOCU CAT
            PAMP;
```

When DOCU is catenated with PAMP the result, in nine bits, is 100111101. However, BASK is only six bits in length. The last three bits of the catenation will be truncated so the result will fit the assigned length of BASK. BASK will contain 100111. If the result of catenation of bit strings yields a length greater than 32 bits, bits will be truncated from the right of the result. No bit string may be longer than 32 bits in length. The catenation of character strings follows the same principles as the catenation of bit strings. The result is left justified within the receiver. If the result is less than the length of the receiver, character blanks are appended to the extra character positions.

```
EXAMPLE:    DCL (ETA,ZETA)
            CHAR(2);
            DCL RHO CHAR(6);
            ETA = 'AB';
            ZETA = 'CD';
            RHO = ETA CAT ZETA;
```

The result, left justified within RHO, is ABCDbb where bb are character blanks. If the result of character string catenation exceeds the character length of the receiver, the right most characters are truncated. Also, if the result of character string catenation exceeds 512 characters, characters will be truncated from the right of the result. No character string may exceed 512 characters in length.

(2) A substring operation designates a portion of a character string or a portion of a bit string. The operator may appear on the left of an assignment statement, the operator may appear on the right of an assignment statement, or the operator may appear on the left and the right of an assignment statement. The specification of a substring operation contains several parts.

```
EXAMPLE:    SUBSTR(ALPHA,2,3)
```

The substring operator is SUBSTR or the symbol $. Following the operator is an identifier, a first element and an element count, all enclosed in parentheses. The identifier (ALPHA in the example) must have been previously defined with an attribute of either CHAR or BIT. The identifier is separated from the first element by a comma. The first element specifies the left most character in a character string or the left most bit in a bit string which is to be used in the operation. The first element may be any character or any bit within a string and the value of the first element must lie between one and the length of the designated quantity (ALPHA). When determining the character position or bit position within a string, the left most character or bit is one, the next is two, etc. The first element defines a starting point within a string. The element count specifies the number of contiguous characters or bits, starting from the first element, for the operation. The sum of the first element and the element count must not exceed the declared length plus one of the designated quantity (ALPHA). Thus, in the example, the substring operation specifies the second character or bit of ALPHA to be the first element and, starting from the first element, three contiguous characters or bits are to be used. The actual character or bit positions of ALPHA entering into the operation are 2, 3 and 4.

```
EXAMPLE:    $(ALPHA, 2)
```

In the above example the element count is missing. Should the element count not be present for the operation it is understood to be one. The identifier and the first element must always be present.

(a) Substring operation on the right of an assign statement

```
EXAMPLE:    DCL LY CHAR(4);
            DCL LA CHAR(6);
            LA = 'ABCDEF';
            LY = SUBSTR(LA,3,2);
```

In the above example, LY is to be set to a portion of LA. The operation states, starting at character C (first element specifies the third character of LA as the left most character) obtain two contiguous characters (element count is two) and assign them to LY. The characters obtained will be C and D. Since the operation does not specify where in LY to place the characters, the characters will be left justified within the receiver (LY) and blanks will

be appended to the extra character positions. At the conclusion of the operation LY will be made equal to CDbb (where bb are character blanks). In this type of substring operation blank fill will always be used when necessary.

(b) Substring operation on the left of an assign statement

```
EXAMPLE:    DCL PRE CHAR(6);
            DCL PRO CHAR(8);
            PRE = 'ZYXWVU';
            PRO = 'AAAAAAAA';
            $(PRO,2,4) = PRE;
```

In the above example, a portion of PRO is to be set to a portion of PRE. The portion of PRO to be set is specified by the substring operation, character positions 2, 3, 4 and 5 (first element specifies character position two, element count is four). However, the operation does not indicate the character positions in PRE to be used. In this case, the starting character position of PRE will be the left most character (Z) and four contiguous characters will be used; Z, Y, X and W. This type of operation specifically states where to place the characters in the receiver (PRO). The operation becomes a true insert and there is no blank fill to the left or to the right of the inserted characters. At the conclusion of the operation PRO is assigned the value of AAZYXWAA. Note that character positions 1, 2, 7 and 8 or PRO are not disturbed.

(c) Substring operation on the left and on the right of an assignment statement

```
EXAMPLE:    DCL GRAPE CHAR(8);
            DCL APPLE CHAR(6)
            INIT
            ('123456');
            GRAPE =

            'ABCDEFGH';
            $(GRAPE,1,3) =
            $(APPLE,3,3);
```

In the above example, the substring operator appears on both sides of an assign statement. The statement explicitly states where the portion of a character string is to come from (third, fourth and fifth characters of APPLE) and where that portion is to be placed (first, second and third character positions of GRAPE). This operation is a true insert, there is never a blank fill in the receiver. At the conclusion of the operation the value of GRAPE will be 345 DEFGH characters 3, 4 and 5 of APPLE (first element is 3, the element count is 3) have been placed in character positions one, two and three of GRAPE (first element is 1, the element count is 3). Substring operations for bit strings follow the same principles as the substring operations for character strings. The only difference is when extra bit positions have to be filled they are ZERO filled.

```
EXAMPLE:    DCL (FOG, LOG) BIT(8);
            FOG = '11100111'B;
            LOG =
            SUBSTR(FOG,2,5);
```

In the example, bit positions 2, 3, 4, 5 and 6 of FOG are left justified within LOG. (The first element specifies the second bit position of the string and the element count is five.) LOG has been defined as eight bits in length. Three ZEROS will be appended to the extra bit positions of LOG. At the conclusion of the operation the value assigned to LOG will be 11001000.

d. Logical operators. The logical operators are the boolean operators AND, OR and NOT. The symbols for these operators are shown in table 6-4.

Table 6-4. Logical Operator

SYMBOL	OPERATION
AND or &	Logical AND
OR or |	Logical Inclusive OR
NOT or ¬	Logical NOT

20083-7

The NOT operator is always a prefix operator while the AND and OR are always infix operators. The logical operators may only be used with bit strings.

(1) The AND operation, as with all logical operations, is performed on a bit by bit basis from left to right. An AND operation performed between two bit strings yields a result for each bit position of the strings. For each bit position, a ONE and a ONE yields a result of ONE. All other combinations yield a ZERO.

```
EXAMPLE:    DCL (AA,DD,HH)
            BIT(6);
            AA = '111000'B;
            DD = '001110'B;
            HH = AA AND DD;
```

As a result of the above operation HH will have a value of 001000. Only in the third bit position of the two strings is there a ONE and a ONE combination which yields a ONE. The ONE-ZERO, ZERO-ONE and ZERO-ZERO combination all yield ZERO results. There is never a carry from one bit position to another in logical operations. It is possible to perform a logical operation between two bit strings of unequal length.

```
EXAMPLE:    DCL FIX BIT(5);
            DCL (COMB,POS)
            BIT(7);
            FIX = '10011'B;
            POS = '0111111'B;
            COMB = FIX AND POS;
```

In this example POS is two bits longer than FIX. During the AND operation ZEROS are appended to the shorter of the two values to make them equal in length. After the ZEROS are appended the AND operation is completed. As a result of the above operation COMB will have a value of 0001100.

If the receiving quantity is longer than the result of a logical operation, the result is left justified within the receiving quantity and ZEROS are appended to the extra bit positions.

```
EXAMPLE:    DCL(TRA,NGY) BIT(6);
            DCL GIV BIT(10);
            TRA = '101010'B;
            NGY '011011'B;
            GIV = TRA AND NGY;
```

In this example the AND is performed between TRA and NGY with the result left justified within GIV. The six bit result of the operation is 001010. GIV is 10 bits in length, so four ZEROS are appended to the right of the six bit answer to provide a ten bit result of 0010100000.

(2) The principles of the OR operation are the same as for the AND operation. The difference in the operators is the result they yield. A logical inclusive OR yields a result of ZERO for a ZERO-ZERO combination. All other combinations yield a result of one. Otherwise, all the rules of operation are the same.

```
EXAMPLE:    DCL MIKE BIT(4);
            DCL LARRY BIT(7);
            DCL SAM BIT(9);
            MIKE = '1100'B;
            LARRY = '1010110'B;
            SAM = MIKE OR
            LARRY;
```

In this example MIKE, four bits in length, is appended with three ZEROS to be of equal length with LARRY which is seven bits in length. The logical inclusive OR between the two values yields a seven bit result of 1110110 which is left justified within SAM. SAM is nine bits in length so two ZEROS are appended to the right of the seven bit answer which yields a nine bit result of 111011000.

(3) The NOT operator performs a ONE's complement of a bit string. This changes all ONE bits to ZERO bits and all ZERO bits to ONE bits.

```
EXAMPLE:    DCL (OPE,COM) BIT(6);
            OPE = '110110'B;
            COM = NOT OPE;
```

As a result of the NOT operation in the example, COM will have a value of 001001 which is the ONE's complement of OPE. If the receiving quantity is longer than the result of a NOT operation zeros are appended to the extra bit positions.

```
EXAMPLE:    DCL KVAL BIT(5);
            DCL PVAL BIT(8);
            KVAL = '00110'B;
            PVAL = NOT KVAL;
```

In the example, the NOT operation is performed on the value of KVAL producing a result of 11001. This result is left justified within PVAL and three ZEROS are appended to the extra bit positions. The NOT is always performed before the ZEROS are appended. Table 6-5 illustrates the result of bit by bit operations for all logical operators.

e. Rules of Operators

(1) Short and long numeric quantities may use only arithmetic and relational operators.

(2) Character quantities may use only relational and string operators.

Table 6-5. Results of Logical Operations

Contents of:		Operation Results			
				A	A
		NOT	NOT	AND	OR
A	B	A	B	B	B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

20083-8

(3) Bit quantities may use only relational, string and logical operators.

6-2. Expressions

There are four types of expressions in TACPOL: short numeric, long numeric, bit string, and character string. The type of data in expressions may not be mixed except in special cases for long numeric expressions.

a. Short Numeric Expressions. If the quantity receiving the result of the expression has been declared as short numeric, then all quantities within the expression must be short numeric. The expressions are evaluated from left to right according to the priorities of the operators. Operator priorities are illustrated in table 6-6.

EXAMPLE: DD = AA+BB*CC/DD;

Table 6-6. Short Numeric Operator Priorities

PRIORITY	SYMBOLS	OPERATIONS
1	+ and -	Prefix addition and subtraction
2	**	Exponentiation
3	* and /	Multiplication and division
4	+ and -	Infix addition and subtraction

20083-9

In the above example, the evaluation of the expression is left to right according to the priority of the operator. Thus, BB is multiplied by CC, the result is divided by DD, and AA is then added to provide the final result. Multiplication and division have higher priorities than addition. The order of priority may be changed by enclosing any part of the expression in parentheses. This raises that portion of the expression to the highest priority.

EXAMPLE: DD = (AA+BB)*CC/DD;

In this example, AA is first added to BB, the result is multiplied by CC and finally the division by DD is made. The inclusion of parentheses changes the evaluation of the expression and the result of this expression would be different from the result of the expression in the previous example. During the evaluation of expressions, the scale factors of the values are automatically adjusted to the scale factor of the value containing the largest fractional part. The result of expression evaluation is then automatically adjusted to the scale factor of the receiving quantity.

EXAMPLE:

```
DCL FIN BIN
FIXED (31,2);
DCL APT BIN
FIXED (31,5);
DCL LIBR BIN
FIXED (31,1);
FIN = 12.5;
APT = 37.125;
LIBR = FIN+APT;
```

In this example, FIN and APT have initial values (shown in binary) as follows:

FIN = 00000000000000000000000001100.10

APT = 00000000000000000000100101.00100

APT has the larger fractional part so FIN is adjusted to the scale factor of APT before the addition is applied. This means moving the value of FIN three binary positions to the left. The three most significant bits of FIN are lost (exclusive of the sign bit) and three ZEROS are appended to the right. The addition occurs with the adjusted values as illustrated below.

FIN = 00000000000000000000001100.10000

APT = 00000000000000000000100101.00100

The intermediate result of the above operation is:

00000000000000000000110001.10100
(49.625_{10})

The intermediate result is now automatically adjusted to the scale factor of the receiving quantity,

LIBR. The intermediate result is moved four binary positions to the right. The four least significant bits of the intermediate result are lost, and likenesses of the sign bit are appended to the left. The value of LIBR as a result of the operation is:

00000000000000000000000110001.1
(49.5_{10})

b. Long Numeric Expressions. The basic rule for long numeric expressions is, if the quantity receiving the result of the expression has been declared as long numeric, then at least one of the quantities within the expression must be long numeric. However, there are two exceptions to this rule due to two special operators for long numeric expressions.

Long numeric expressions are evaluated in the same manner as short expressions with the same automatic adjustment for scale factors. The order of priority is illustrated in table 6-7.

Table 6-7. Long Numeric Operator Priorities

PRIORITY	SYMBOLS	OPERATIONS
1	+ and -	Prefix addition and subtraction
2	** or (**)	Exponentiation
3	* or (*) and /	Multiplication and division
4	+ and -	Infix addition and subtraction

20083-10

Under normal circumstances, when a long numeric quantity is to receive the result of exponentiation, the quantity in the expression must be long numeric. However, there is a special long numeric operator which allows a long numeric quantity to receive the result of exponentiation when the quantity in the expression is short numeric. This is accomplished by enclosing the exponentiation operator in parentheses as illustrated in the example below.

```
EXAMPLE:    DCL FAVT BIN
            FIXED (31,0);
            DCL RSLT BIN
            FIXED (62,0);
            FAVT = 20;
            RSLT = FAVT(**)4;
```

If the exponentiation operator in the above example were not enclosed in parentheses the compiler would output an error condition because FAVT was declared short numeric and RSLT was declared long numeric. A similar situation exists with the multiplication operator. Normally, when a long numeric quantity is to receive the results of multiplication, one of the quantities within the expression must be long numeric. However, there is a special long numeric operator which allows a long numeric quantity to receive the results of multiplication between short numeric quantities. Enclosing the multiplication operator within parentheses allows this to occur as illustrated in the following example.

```
EXAMPLE:    DC L DECIM BIN
            FIXED (31);
            DCL OCT BIN
            FIXED (31);
            DCL HEXA BIN
            FIXED (62);
            DECIM = 37;
            OCT = 112;
            HEXA = DECIM(*)OCT;
```

If the multiplication operator in the above example were not enclosed in parentheses the compiler would detect the expression as being in error.

c. Exponent and Scale Factor for Long Numeric and Short Numeric Literals. A short or long numeric literal may contain an exponent and a scale factor. The exponent is used for raising or lowering the literal by a power of ten. The scale factor is used to scale the literal after the exponent has been applied.

```
EXAMPLE:    DCL MICRO BIN
            FIXED (31,0);
            MICRO = 8E 2S 1;
```

In the example the literal is 8. The E stands for exponent and is followed by an optionally signed number. If a sign (plus or minus) does not follow the E, a blank space must be left between the E and the number. The blank means +. The optionally signed number is the power of 10 which, in the example, would be 10^2. The S stands for scale factor and is followed by an optionally signed number. If the sign (plus or minus) does not follow S, a blank space must be left between the S and the number. The blank means +. The result of the above example is 800 (8 raised by 10^2) scaled one binary bit position to the left for a final result of 1600. MICRO is assigned the value 1600. The literal in the example could also have been coded as follows:

```
            MICRO = 8E+2S+1;
```

The use of a negative exponent lowers a number by a power of 10.

EXAMPLE:
```
DCL LIK BIN FIXED;
LIK = 800E-2S+1;
```

The example uses an exponent of 10^2. As a result, 800 lowered by 10^2 is 8 scaled one binary position to the left for a final result of 16. LIK is assigned the value 16. The use of a negative scale factor scales the result to the right.

EXAMPLE:
```
DCL MIMAT BIN FIXED;
MIMAT = 8E+2S-2;
```

Raising 8 by 10^2 yields a result of 800. This result is scaled two binary positions to the right for a final of 200. MIMAT is assigned the value 200.

d. Bit String Expressions. A bit string expression yields a bit string result. The expression is evaluated from left to right according to the priorities of the operators. Operator priorities are illustrated in table 6-8.

EXAMPLE:
```
BYTE = NOT TMM OR GRGE AND HRY;
```

Table 6-8. Bit String Operator Priorities

PRIORITY	SYMBOLS	OPERATIONS
1	NOT or ¬	Logical NOT
2	AND or &	Logical AND
3	OR or |	Logical Inclusive OR
4	CAT or ||	Catenation

20083-11

Following the order of priority evaluation in the example, the first operation performed is the NOT on quantity TMM; then a logical AND is performed between quantities GRGE and HRY and finally, a logical inclusive OR is performed between the result of the NOT operation and the result of the AND operation. The order of priority may be changed by enclosing part of the expression in parentheses. This is illustrated in the following example.

EXAMPLE:
```
QUO = NOT (STATUS CAT VADIS);
```

Catenation has a lower priority than the NOT operator. However, the catenation operation in the example is enclosed in parentheses so the quantities STATUS and QUO are catenated before the NOT operation is performed.

e. Character String Expressions. A character string expression yields a character string result. The expression is evaluated from left to right without any priority considerations of the operators. Only relational and string operators may appear within the expression.

f. Repeat Factor for Bit String and Character String Literals. A repeat factor may be used for bit string and character string literals. The effect is to catenate the literal to itself the number of times specified by the repeat factor.

EXAMPLE:
```
DCL SPEC CHAR (16);
SPEC = (4) 'KMPC';
```

The repeat factor precedes the literal and is an unsigned decimal number enclosed in parentheses. The example illustrates the use of the repeat factor for a character string. The literal KMPC is to be repeated four times assigning a value to SPEC of KMPCKMPCKMPCKMPC. The same function can be applied to bit string literals.

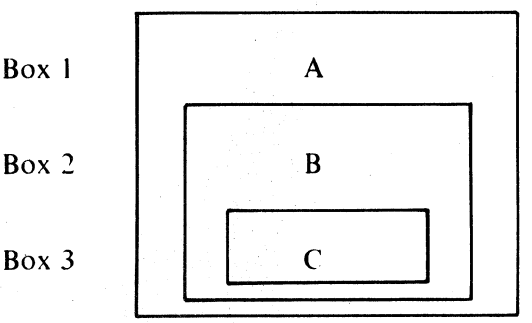
EXAMPLE:
```
DCL FUNCT BIT (9);
FUNCT = (3) '101'B;
```

As a result of the operation in the example, FUNCT will be assigned a value of 101101101.

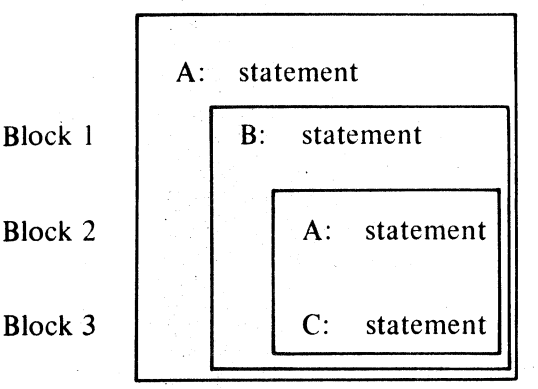
CHAPTER 7
BLOCKS

7-1. Block Structure

A TACPOL program is organized into blocks. A block consists of a collection of statements. Within the blocks, value names, quantities, point names, and procedure names are defined. The purpose of the block structure is to define the scope of names. To attain a clearer understanding of the block concept, consider blocks as boxes. These boxes can be nested (contained) one within another. For example:



The names defined in any one box are automatically 'known' throughout that box and all boxes contained within that box. In the example, A is defined in Box 1 and is thus known in Box 1 and the boxes it contains - Boxes 2 and 3. Similarly, B, which is defined in Box 2 is known in both Boxes 2 and 3. Finally, C is defined and known only in Box 3. The 'scope' of a name is the area of a program within which it is known. Thus, the scope of name A extends throughout Boxes 1, 2 and 3, where it is known. Likewise the scope of B extends throughout Boxes 2 and 3, and the scope of C extends throughout Box 3. All names used in a TACPOL program must be defined. Therefore, in selecting the proper definition for a name the program is searched starting with the innermost block designating the name and working outwards. If the name is defined within the smallest block that encloses that designation, this definition is selected as valid. Otherwise, the search is reapplied to the next outer-most block. Each successively larger block is searched until the definition is found. If the definition is not found, then the name is considered undefined and in error. The objective in using blocks in a program is to define the scope of a name. The scope of a name extends throughout the block in which it is defined and every contained block, except any contained block where it is redefined. Consider the following example:



The name B is defined and may be designated in block 2 and its contained block, 3. The name C is defined and may be designated only in block 3. The name A is defined first in block 1 and may be designated in blocks 1 and 2. Block 3 is indeed contained in block 1; however, because A is redefined in block 3, the scope of the outer A does not extend into block 3. When the block structure is searched starting from block 3, the inner definition of A will be found first. Thus any designation of A within block 3 will use this inner definition from block 3. When name A is referred to outside of block 3, specifically in either block 1 or 2, the block structure again is searched. For example, if A is designated in block 2 then this block is searched for the definition of A, where it is not found. The search is continued in the next containing block, 1, where the definition is found. Thus, any designation of A within block 2 will use this outer definition. Likewise, designations of A in block 1 will also use the outer definition. Thus, the programmer is free to devise any block structure for his program that is convenient to the solution of his problem. He may use the same name, if appropriate, for different entities defined in his program, as long as they are defined in different blocks. There are four types of blocks: BEGIN, DO, PROC, and CODE.

7-2. Begin Block

A BEGIN block consists of statements bounded by BEGIN and END.

```
BEGIN;
    statement-1;
    statement-2;
    statement-3;
END;
```

The BEGIN block may also contain declarations. A declaration must always precede its use in a STATEMENT. Because the BEGIN block is treated like a statement, it may appear anywhere a statement may appear. For example:

```
IF (A = B) THEN
    BEGIN;
      statement-1;
      statement-2;
      statement-3;
    END;

ELSE
  BEGIN;
    statement-4
    statement-5
    statement-6
  END;
```

7-3. DO Block

A DO block consists only of statements bounded by DO and END. No declaration may appear in a DO block.

```
DO I = 1 BY 1 TO 10;
    statement-1;
    statement-2;
    statement-3;
END;
```

The DO block, like the BEGIN block, is treated like a statement.

7-4. PROC Block

A PROC block consists of the procedure statements from (but not including) the procedure name to (and including) the END.

```
A:  PROC;
    statement-1;
    statement-2;
    statement-3;
    END;
```

Unlike BEGIN and DO blocks, the PROC block is treated like a declaration. The procedure name in the example A, is known in the block immediately containing the procedure. Consider the nested procedure blocks:

```
A:  PROC;
    B:  PROC;
          statement-a;
          statement-b;
          statement-c;
        END; /* B */
    statement-1;
    statement-2;
    statement-3;
    END; /* A */
```

Procedure blocks may contain declarations in addition to statements. Like BEGIN blocks, any declaration must precede its use in a statement. Therefore a nested procedure must appear among such declarations (in a PROC or BEGIN block) as procedure B appears preceding the statements in A, as shown above.

7-5. CODE Block

A block whose first constituent is the particle CODE specifies a text not written in the TACPOL language. Following the code block specification is a list of AN/GYK-12 Machine Language instructions. Thus, machine language code may be imbedded within a TACPOL language program.

a. A CODE block consists of the CODE block specification, AN/GYK-12 Machine Language instructions and an END statement which designates the end of the CODE block. (See chart below.)

```
CODE;

   MOL   statement-1
   MOL   statement-2
   MOL   statement-3
   MOL   statement-4
   MOL   statement-5
   MOL   statement-6
END;
```

CODE blocks may be nested within TACPOL blocks. However, names defined in TACPOL blocks are not normally known to CODE blocks. Therefore, the use of names in a code block that were defined in declarations would be undefined in the CODE block.

EXAMPLE:

```
CODE USES (LPRA,
PONN, GGUY);
```

b. There is a method of getting around the problem of undefined names in a CODE block. A modified CODE block containing a list of names, previously defined in TACPOL blocks, makes the names in the list known in the CODE block.

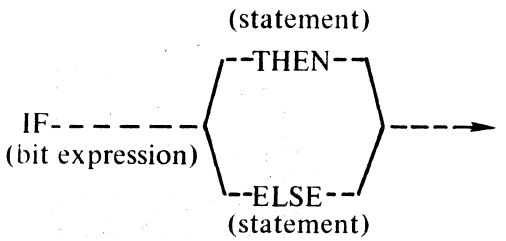
In the above example, the names in the list have been previously defined in outer or parallel blocks. The particle USES must follow the specifier CODE when a list of names is used in the CODE block heading.

CHAPTER 8
CONTROL STATEMENTS

8-1. IF Statement

A logical diagram of the IF statement would look like this:

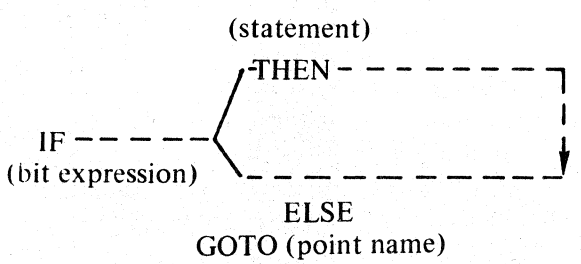
A logical diagram of the IF statement would look like this:



An example of such an IF statement to produce a positive difference between A and B would be:

IF (A LT B) THEN C=B-A; ELSE C=A-B;

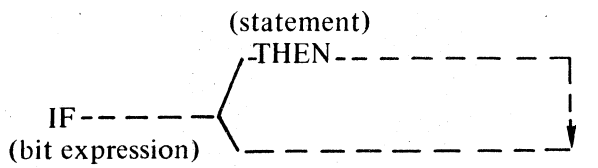
The two paths diverge for the execution of one statement (or block of statements) and merge again into a single path of execution. Either the THEN clause or the ELSE clause is executed, and the other is skipped. No matter which alternative is chosen as a result of the test, execution continues with the next sequential statement that appears in the program immediately following the ELSE clause, provided no transfer of control statement was executed in the selected clause. If the result of the bit expression in the IF clause is all ZEROS, this is treated as a FALSE condition and the ELSE clause will be executed. If the evaluation of the bit expression contains at least one bit then this is treated as a TRUE condition and the THEN clause will be executed. A variation of this type of IF statement is represented by the following diagram:



An example of such an IF statement would be:

IF (A = B) THEN A = -B; ELSE GOTO EQUAL;

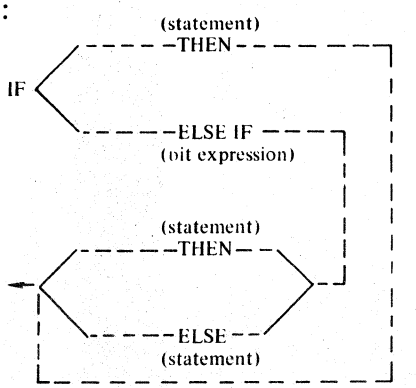
In this example an alternative (the ELSE clause) causes a transfer of control to some other point in the program. Sequential execution does not continue. Another kind of IF statement is represented by a third diagram:



A statement representing the above kind of IF statement could be as follows:

IF (A NE B) THEN A=A-1;

In the type of IF statement illustrated above, the alternatives are 'execute the THEN clause' or 'do not execute the THEN clause.' In either case, the next sequential statement is executed. If the expression tested is not true, control continues through the logical flow of execution. If the expression is true, the THEN clause is executed, and execution continues with the logical flow. Although the ELSE clause if sometimes omitted, as in this case, the THEN clause must appear in every IF statement. The statement in the THEN clause may be any statement and the statement in the ELSE clause may be any statement. For example:



IF (A=B) THEN C=0;

ELSE IF (A GT B) THEN C=+1; ELSE C=-1;

In this example, C is assigned the value 0 if A is equal to B; otherwise, if A is greater than B, C is assigned the value +1: otherwise (if A is less than B), C is assigned the value -1. The programmer may, of course, include further complexities as required. It should be noted that BEGIN and DO blocks (described earlier in chapter 7) qualify as statements and may appear in THEN and ELSE clauses.

8-2. NULL Statement

The NULL statement, as its name implies, is an empty statement: the only portion of a statement that appears is the terminating semicolon. Such a statement gives no direction to the computer; it may appear anywhere any other statement may appear, and is most often used in an IF statement in the THEN clause, or where the ELSE clause is specified and no action for the THEN clause is desired.

8-3. DO Statement

The DO statement is used to define and specify control for a block of statements to be used in a loop. Looping consists of a series of statements executed and repeated one or more times before control continues to the statement following the block. Every DO statement must have an associated END statement to define the end of the DO block.

a. The DO statement itself consists of the DO particle followed by the DO quantity identifier and an = sign. The initial value of the DO quantity appears next, followed by the BY clause and the TO clause. The initial value of the DO quantity and the numeric specifications in the BY clause and TO clause may be signed literals or any other short numeric expressions. The TO clause may be omitted; however, if it is, execution could continue indefinitely. Consider the following example:

```
DO COUNTER = 1
BY 1 TO 10;
statement-1;
statement-2;
statement-3;
END;
statement-4;
```

Statements 1, 2 and 3 constitute the DO block and are delimited by DO and END. The DO statement specifies that these statements are to be executed, as a block, ten times before control is transferred to statement-4. The quantity COUNTER is used to control the number of times the block is executed. When the DO statement is executed for the first time, COUNTER is assigned the value 1. Statements 1, 2 and 3 are then executed. When the END statement is reached, COUNTER is incremented by one, and control is transferred back to the beginning of the block where COUNTER is tested to see that it is no larger than 10. This looping continues until the value of COUNTER exceeds 10, at which point control passes on to statement-4. The above example is equivalent to the following:

```
          BEGIN;
          DCL COUNTER BIN
          FIXED (15);
          COUNTER = 1;
LOOP:     IF (COUNTER GT 10)
          THEN GO TO NEXT;
          statement-1
          statement-2
          statement-3
          COUNTER =
          COUNTER + 1;
          GOTO LOOP;
          END;
NEXT:     statement-4
```

An increment other than 1 can be stipulated. For example:

```
DO COUNTER = 1 BY 2
TO 10;
```

This DO statement causes the initial value of COUNTER to be set to one. Each time the DO statement is executed, the value is incremented by two. Thus, the statements of the DO block would be executed five times, the final time with COUNTER equal to 9.

b. The maximum value allowed for the DO quantity (COUNTER) in any DO statement is 32,767. The DO quantity may also be used in an expression within the DO block. For example, the following DO block could be used to compute the sum (in cubic inches) of the volumes of each of a series of circular ponds. Assume that every pond is 12 inches deep and that the diameters range from 18 inches to 10 feet, using six inch increments from size to size.

```
VOL = 0;PI =
2.1415926;
DO I = 9 BY 3 TO 60;
DO I = 9 BY 3 TO 60;
VOL = VOL + 12**
(PI * I * * 2);
END;
```

The initial value assigned to I is nine, which represents the radius of the smallest pond. Each increment of three makes I equal to the radius of the next larger size. The volume is computed for each size, and the result is summed in the quantity VOL.

c. The DO statement may be written without the TO phrase. In this case, looping will continue until some GOTO statement within the loop transfers control out of the loop. For example:

```
VOL = 0;PI =
3.1415926;
DO I = 9 BY 3;
IF (9I GT 60)
THEN GOTO X;
VOL = VOL + 12 +
(PI * I * * 2);
END;
x:          statement;
```

d. The DO statement may also be written without both the BY and TO phrases. In this case, the loop will be executed just once. An example of using short numeric expressions to define initial incremental and final values for the DO variable is:

```
DO I = (A+B/C) BY
(A+1)
TO (A*B*C);
```

e. Care must be taken in the use of control expressions so that the final value is exceeded (or surpassed) from the initial value by successive increments (or decrements). The expressions are evaluated only once at the initial entry to the DO block. The value of I may not be changed by statements written within the DO loop. There is a method of loop control which allows looping to continue as long as a certain condition exists. This method involves the WHILE clause as follows:

```
DO I=1 BY 1 WHILE (A
LT B);
```

The values of A and B are compared each time control reaches the DO statement. The computer continues executing the statements in the DO block until the value of A becomes equal to or greater than the value of A or B. Only bit expressions are allowed in the WHILE clause. In addition, care must be taken to insure that the conditions of the WHILE clause are reasonably attainable or the loops will be unending. It is advisable to never use the WHILE clause without a specific counter (TO clause) also defined for the loop. The counter ensures against excessive execution in case the WHILE condition proves to be unattainable. Note also that while any expressions in the TO or BY clause are evaluated only once on entry to the DO block, the comparison indicated in the WHILE clause is made each time the block is executed. Combining the preceding features there is the following form of DO statement:

```
DO I=1 BY 1 TO 10 WHILE (A LT B);
```

This control expression causes repeated execution of the group either until the tenth execution is completed or until A no longer is less than B. As soon as either condition is satisfied, execution ceases, no matter what the status of the other. DO blocks may be nested. Consider this example:

```
DO I=1 BY 1 TO 10;
statement-1
statement-2
statement-3
    DO J=1 BY 1 TO 10;
    statement-1A
    statement-2A
    statement-3A
    END;
statement-4
statement-5
statement-6
END;
```

f. The statements of the outer DO block (the other DO through END and statements 1 through 6) are executed ten times. The statements of the inner DO block (the inner DO through END and statements 1a through 3a are executed 100 times, ten times for each execution of the outer DO group. When the first DO statement is executed the first time, counter I is assigned the value 1.

Then statements 1 through 3 are executed. When control reaches the second DO statement, counter J is assigned the value 1, and the inner loop is executed until the value of J exceeds 10. Control then passes on to the first DO statement. The counter I is incremented by 1, and execution proceeds through statements 1 through 3. When the second DO statement is reached for the second time, J is reset to 1, and the inner DO block again is executed ten times before control passes to statement 4 for its second execution. The process is repeated until the outer DO block has been executed ten times. The inner DO block goes through its entire looping process immediately following each execution of statement 3. The example shows nesting only to the second level. Whatever the number of nested blocks, each contained block will be executed to completion for every single execution of its containing block. Control may not be passed to a statement within a DO block from outside of the DO block. Control may, however, be transferred out of a DO loop terminating execution of the DO block. For example, a GOTO statement might appear within a THEN or ELSE clause of an IF statement in the loop.

CHAPTER 9
PROCEDURES

Section I. INTRODUCTION

9-1. General

A program is a procedure that is not contained in any other procedure. A program consists of this single procedure block and possibly several nested procedure blocks. At execution time, the program is invoked automatically. During execution of the program, control can go from one procedure to another and return.

9-2. Proper Procedures

There are several different types of procedures. Proper procedures are procedures which are invoked by a CALL statement. Function procedures are invoked in an assignment statement (see Chapter 5). Function procedures have a RETURN statement and the name of a function procedure may be used to make the value in the expression in the RETURN statement available to the assignment statement which invokes the function procedure. Proper procedures do not have this capability and do not have RETURN statements. Explicit procedures are procedures (proper or function) which are coded by the user, whereas intrinsic procedures are available to all programs through the TACPOL compiler. This chapter discusses proper procedures but the concepts presented here for proper procedures also hold true for the other types of procedures. A proper procedure is headed by a PROC (procedure) statement and ended by an END statement, as follows (the dots represent the statement in the procedure.

```
EASTER:    PROC;

             .
             .
             .

           END;
```

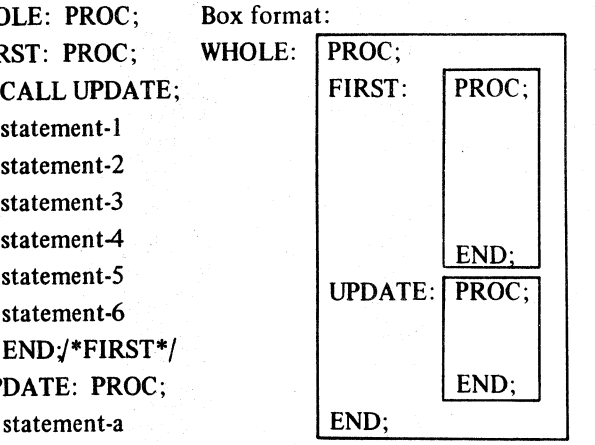
Each procedure must have a name such as EASTER in the example. (The format for definition of a procedure name NAME: should not be confused with the similar format for definition of point names.) Control does not pass automatically from one procedure to the next. Each procedure, except the first, must be invoked, or called separately from some other procedure. This usually occurs with the execution of a CALL statement, for example:

```
CALL EASTER;
```

Execution of this statement in another procedure would transfer control to the first executable statement of the procedure named EASTER. The different procedures contained within a program may be entirely separate from one another, or some may be nested. Consider the example:

```
WHOLE: PROC;
  FIRST: PROC;
    CALL UPDATE;
    statement-1
    statement-2
    statement-3
    statement-4
    statement-5
    statement-6
    END;/*FIRST*/
  UPDATE: PROC;
    statement-a
    statement-b
    statement-c
    END;/*UPDATE*/
  CALL FIRST;
  END;/*WHOLE*/
```

Box format:



WHOLE is a program which contains the two procedure blocks. FIRST and UPDATE. Following the rules of scope, the name WHOLE is known throughout the program. The procedure names, FIRST and UPDATE, are known throughout their procedures as well as known to each other.

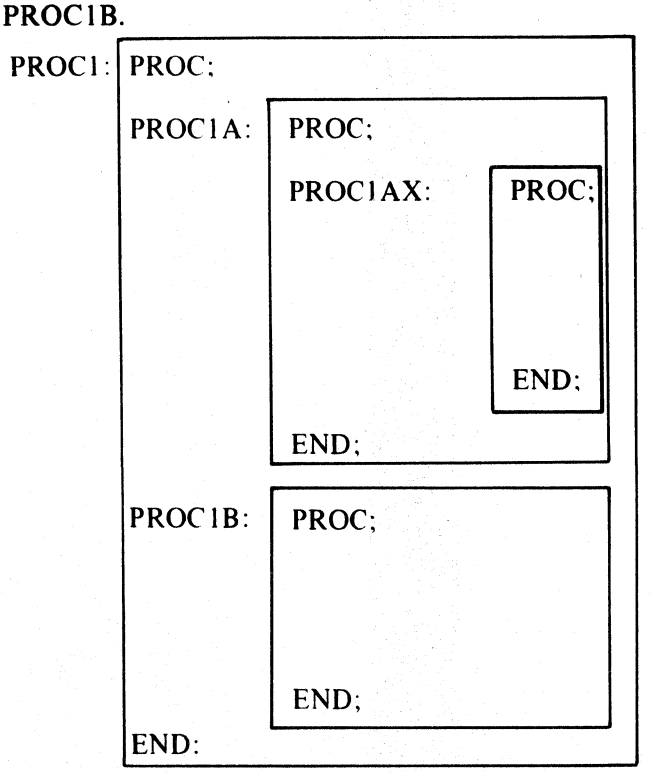
a. CALL Statement. The CALL statement defines:

(1) the point of invocation, which is the CALL statement itself;

(2) the invoking procedure, within which the CALL statement is contained, and

(3) the invoked procedure, which is the procedure referred to in the CALL statement.

Thus, in the example, the point ov invocation of FIRST is the statement CALL FIRST: , the invoking procedure is WHOLE and the invoked procedure is FIRST. Any procedure invoked by WHOLE might in turn invoke other procedures. For example, procedure FIRST invokes procedure UPDATE in the example. But control eventually returns to the statement in the invoking procedure WHOLE that immediately follows the point of invocation; in this case the END statement for the procedure WHOLE. More than one procedure may be contained within a single procedure either as separate procedures or as nested procedures. (See diagram on next page.) Consider procedures PROC1A, PROC1AX, and PROC1B all contained in PROC1. PROC1AX is contained in PROC1A. In this situation PROC1 can invoke either PROC1A or PROC1B; PROC1A or PROC1B can invoke one another but only PROC1A can invoke PROC1AX. In addition PROC1AX could invoke PROC1B.



b. GOTO Statement. A GOTO statement may be used in any block to transfer control to a point within the block itself or to a point in any containing block. However a GOTO statement may not be used to transfer control to a point in a different block on the same level or a separate block contained within the block executing the GOTO. A GOTO statement in PROC1A may not transfer control to a point in PROC1B or in PROC1AX. However a GOTO statement in PROC1AX may transfer control to a point in any procedure (including itself) except in PROC1B. This follows from the rules of scope. Control returns to an invoking procedure when the END statement of an invoked procedure is reached. Often there are reasons why a programmer wants control to return before the END statement would normally be reached. The example below illustrates such a situation where the GOTO statement is used to transfer control to a point preceding the END statement.

```
PROGRAM: PROC;
        statement-1
        statement-2

TEST: IF (DISTANCE=0) THEN GOTO
STOP;

        statement-3
        statement-4
        GOTO TEST;
STOP: END;
```

The execution of the procedure PROGRAM will end when the IF condition is satisfied, which will occur when DISTANCE = 0. The statement GOTO STOP; will then be executed, control transferred to the END statement following the point name STOP, and control passed through the END statement back to the procedure which invoked PROGRAM. The END statement must physically be the last statement in the procedure. A procedure can be terminated only when its END statement is executed, or when a GOTO statement transfers control to a containing block.

Section II. FUNCTION AND INTRINSIC PROCEDURES

9-3. Function Procedures

The proper procedure, as discussed in the previous section, is invoked by a CALL statement, and terminated by an END statement. A function procedure, on the other hand, is invoked by the appearance of its name in an expression. In addition the value of the function is made available to the invoking expression through the function name of the function procedure itself by the RETURN statement. The proper procedure does not have a RETURN statement and the name of a proper procedure may not be used as a quantity for data. The function is terminated by the END statement. An example of a function procedure is:

```
A:   PROC BIN FIXED (15, 0);

          RETURN 6 + 3);

          END;

     Z = A + 13;
```

A is the name of the function procedure as well as being the quantity through which the returned value of the function will be made available to the expression in the assignment statement which invoked the function. BIN FIXED (15,0) is the type specification of A. When the function A is invoked by the expression in the assignment statement, the value (9) of the expression which appears in the RETURN statement will be made available to the expression in the assignment statement by means of the function name A. This value (9) will then be added to 13, and assigned to Z.

9-4. Intrinsic Procedures

Explicit procedures are procedures (proper or function) which are written by the programmer who intends to use them. An intrinsic procedure, however, is a proper or function procedure which is available to all programs through the TACPOL compiler. Intrinsic procedures may be invoked by a CALL statement or by an expression. For example,

```
A = SIN(X) + .5;
```

SIN is a short numeric intrinsic procedure (see Appendix I). This assignment statement invokes the intrinsic procedure SIN. SIN computes the trigonometric sine of the value of the expression X. The result will then be passed back, added to .5, and assigned to the quantity A. The intrinsic procedures available to the TACPOL user are presented in Appendix A along with a brief description of the purpose of each.

CHAPTER 10
NAMES

10-1. General

All names must be declared. Specifically, data names are declared in data declarations:

```
DCL A BIN FIXED (7,2);
```

Procedure names are declared by their appearance in procedure declarations:

```
A: PROC;
```

Point names are declared by their appearance preceding statements in the text:

```
A: statement;
```

10-2. In data declarations common attributes can be specified for more than one name by enclosing the names in parentheses and specifying the common attributes following the closing parentheses. When more than one name is used in this manner in a scalar declaration, the group of names is referred to as an identifier list. For example:

```
DCL (A,B) BIN FIXED (15,0);
```

The BIN FIXED (15,0) attributes are specified for the identifier list containing A and B. Identifier lists cannot be used in value declarations, and may only be used in scalar declarations.

a. When a single procedure that has no contained blocks, an identifier cannot be declared more than one time. However, the same identifier can be declared more than once in separate blocks. This redefining process is known as redeclaration. As previously discussed, a name is known through the procedure in which it is declared and throughout all the contained procedures where the same identifier is not redeclared. When a name is redeclared the scope directed by its original declaration is discontinued and a new scope is set up by the redeclaration. The new scope is effective throughout the procedure in which the name is redeclared and throughout all contained procedures (in which it is not again redeclared). Care should be taken not to inadvertently redeclare names of intrinsic procedures, file names or other Compool data (see Chapter 14). For example, if MOVE (an intrinsic structure procedure, see Appendix A) were used unintentionally as a point name, MOVE would no longer be accessible as an intrinsic procedure in that block or in any blocks contained within that block. Note the example on the following page. Examine the example below, which consists of a program FIRST, containing the single procedure SECOND, and the related chart below.

```
1   FIRST:  PROC;
2           DCL (M,N) BIN FIXED (15,0);
3           DCL ALPHA BIN FIXED (6,2);
4           SECOND:  PROC;
5                    DCL M CHAR (5);
6                    DCL TITLE CHAR (8)
                              .
                              .
                              .
7                    END;
8           CALL SECOND;
9           A = 3;
                     .
                     .
                     .
10          END
```

b. The chart below defines the use and scope of each name that appears in the previous example.

LINE NUMBER	NAME	USE	SCOPE
1	FIRST	procedure name	entire program
2	M	short numeric quantity	FIRST, but not SECOND because M is redeclared in SECOND
2	N	short numeric quantity	all of FIRST
3	ALPHA	short numeric quantity	all of FIRST
4	SECOND	procedure name	all of FIRST
5	M	character string quantity	SECOND
6	TITLE	character string quantity	SECOND
9	A	short numeric quantity	assumed declared in COMPOOL; hence all of FIRST and any other programs in which it is not redeclared.

CHAPTER 11
ARGUMENTS AND PARAMETERS

11-1. Introduction

It is often desirable to provide values or quantities to a procedure, when the procedure is invoked. This is accomplished by the use of arguments and parameters. Arguments and parameters are the tools used to establish communiation between the invoking statement and the invoked procedure. The parameter is a name used in the invoked procedure to represent an argument. The argument is a name or expression provided to the parameter by an invoking statement. The point of invocation is a CALL statement in the case of proper procedures, or an expression in the case of function procedures.

a. Correspondence is established between the arguments and parameters as follows: The argument list appearing in the invoking statement or expression must have the same number of arguments as there are identifiers in the parameter list of the invoked procedure. Communication is obtained by the exact correspondence of the members of these two lists, as the members are paired in order.

b. Each identifier in the parameter list must be defined in a parameter declaration. Parameter declarations must appear immediately after the procedure declaration. There are four types of parameters and corresponding arguments; quantity, value, procedure, and point.

11-2. Quantity Arguments and Parameters

For each quantity parameter there must be a declaration defining its attributes following the procedure head. The quantity parameter declaration may be any legal data declaration (except a value declaration), such as:

```
DCL QPARAM BIN FIXED (15, 1);
```

QPARAM is the identifier of the quantity parameter. It is understood that the set of quantities thereby defined will not have an identity of its own, but will assume the identity of the set of quantities designated by the corresponding argument, established at the point of invocation. In other words, a quantity argument denotes the location of data in storage. It is this data which is provided to the quantity parameter. Consider the following example:

```
EXAMPLE:  PROC;

          DCL X(10, 10) BIN FIXED;

          CALL DIAG (X);
             /*X(1, 1). X(2, 2), . . .X(10, 10)
             = 1 */

          END; /*EXAMPLE*/

          DIAG:  PROC (A);
                 DCL A (10, 10) BIN FIXED;
                 DO I = BY 1 TO 10;
                      A(I, I) = 1;
                      END;/*DIAG*/
```

Within the containing procedure EXAMPLE, the CALL statements invoke DIAG, a procedure which sets all diagonal elements of the parameter array A to 1. The quantity A in the procedure DIAG is contained in the parameter list of the procedure DIAG. The quantity X is contained in the argument list of the first CALL statement which invokes the procedure DIAG. When the CALL statement invokes the procedure DIAG, the test dimensional array X (defined in the invoking procedure EXAMPLE) will be made available to the procedure DIAG through the parameter A.

11-3. Value Arguments And Parameters

For each value parameter there must be a declaration defining its attributes following the procedure head. The value parameter declaration consists of only simple scalar definitions in the following format:

```
DCL VPARAM BIN FIXED (15, 1) VALUE;
```

VPARAM is the identifier of the value parameter. The parameter attributes must correspond to the attributes of the argument by type (BIN FIXED) but not necessarily by size or scale (15, 1). As previously discussed, a quantity parameter requires that the information at the location addressed by the corresponding argument by used. In contrast, a value parameter declaration requires thtat the value of the corresponding argu-

ment (some expression) be used. The effect of this is that any changes to a quantity parameter in the invoked procedure will also affect the quantity in the invoking procedure. However the value parameter specifies that a 'snapshop' of the value of the quantity is to be taken when the procedure is invoked. Therefore any changes to a value parameter in the invoked procedure do not affect the value in the invoking procedure. In summary the value parameter causes the specified quantity to be redefined and the quantity parameter causes the current definition to be used in the invoked procedure. Consider the following example:

```
EXAMPLE:
   PROC;
   DCL A BIN FIXED (15.7);
   SIGN:   PROC (X) BIN FIXED (31, 0);
           DCL X BIN FIXED (31, 0) VALUE;

           IF (X GT O) THEN X = +1;
           ELSE IF (X LT O) THEN X = -1;
            /* Note that setting X does not change the
            value of the */
            /* argument in the invoking procedure */
           RETURN (X);
           END; /*SIGN*/

   A = -48, 25;
   A = SIGN(A);/*first invocation, where A = SIGN(48.25)
                = -1 */
   A = 48.25;
   A = SIGN(2*A+3.5); /*second invocation, where   */
                      /* A = SIGN (2*48.25+3.5)  */
                      /*   = SIGN (100)          */
                      /*   = +1                  */

   A = -48.25;
   A = SIGN(-2*SIGN(A);
    /*third invocation, where A= SIGN(-2*SIGN(-48.25)) */
    /*                         = SIGN(-2* -1)          */
    /*                         = SIGN(+2)              */
    /*                         = 1                     */
   END;/*EXAMPLE*/
```

Within the containing procedure EXAMPLE, expressions in various assignment statements invoke SIGN, a procedure which determines the sign of the value of its argument. During the first invocation of SIGN, the value of the expression -48.25 (where A = -48.25) is passed and assigned to X in the SIGN procedure. In this case, X is less than 0 and so is assigned -1. Therefore, as stated in the procedure comment, the SIGN of the expression is -1. During the second invocation of SIGN, the value of the more complex expression, 2*A+3.5 (where A = 48.25) is passed and assigned to X in SIGN. X is equal to 100, is greater than 0, and thus is assigned +1. Therefore, the SIGN of the expression, 2*A+3.5 , is +1. During the final invocation of SIGN, the value of an even more complex expression, -2*SIGN(A) (where A = -48.25) is passed and assigned to X in SIGN. In this more intricate case, the inner-most portion of the expression, SIGN(A), is evaluated. The result of this evaluation is -1. Then the SIGN of the entire expression, -2*(-1), is evaluated and the result is +1.

11-4. Procedure Arguments And Parameters

For each procedure parameter there must be a declaration defining its attributes, following the procedure head. The procedure parameter declaration has the following format:

```
DCL PPARAM ENTRY;
```

PPARAM is the identifier of the procedure parameter. The procedure parameter may only designate an argument that refers to a parameterless procedure name, defined within the program. Consider the following example:

```
EXAMPLE: PROC;
         DCL (A, B, C) BIN FIXED;

         ADD:  PROC;
               C=A+B;
               END; /* ADD*/

         SUB:  PROC;
               C=A-B;
               END; / *SUB*/

         COMPUTE: PROC(ARITH);
                  DCL ARITH ENTRY;

                  B=A;

                  CALL ARITH;
                  END; /*COMPUTE*/

         A=5;
         CALL COMPUTE (ADD);
         /*C=A+B=A+A
            =5+5=10 */

         CALL COMPUTE (SUB);
         /*C=A-B=A-A
            =5-5=0 */
         END; /*EXAMPLE*/
```

Within the containing procedure EXAMPLE, the CALL statements invoke COMPUTE, which in turn invokes the procedure ARITH. ADD and SUB are the procedure arguments provided to the procedure parameter ARITH at invocation. The quantity A is initially assigned 5. During the first invocation of COMPUTE, the parameter ARITH is replaced by the address of argument ADD. Thus in effect, CALL ARITH invoked ADD, where A+B is computed and assigned to C (see first comment). Control passes out of ADD. and then out of COMPUTE to the next statement. During the second invocation, the parameter ARITH is replaced by the address of argument SUB. In this case, CALL ARITH invokes SUB, where A-B is computed and assigned to C.

11-5. Point Arguments And Parameters

For each point parameter there must be a declaration defining its attributes, following the procedure head. The point parameter declaration has the following format:

```
DCL PNPARAM LABEL;
```

PNPARAM is the identifier of the point parameter. The point parameter may only designate an argument that refers to a point name. Consider the following example:

```
EXAMPLE: PROC;
         DCL A BIN FIXED;
         A = -5;
         CALL TEST (A, L1, L3, L2);

         L1: A = A * 0;
             CALL TEST (A, L3, L2, L1);
         L2: CALL TEST (A + 1, L2, L1, L3);
         L3: END; /*EXAMPLE*/

TEST: PROC (X, JLT, JEQ, JGT);
      DCL X BIN FIXED VALUE;
      DCL (JLT, JEQ, JGT) LABEL;
      IF (X LT 0) THEN GOTO JLT;
      ELSE IF (X GT 0) THEN
      GOTO JGT;
      ELSE GOTO JEQ;
      END; /*TEST*/
```

Within the procedure EXAMPLE, the CALL statements invoke TEST, a procedure which tests a value as to whether it is less than, greater than, or equal to 0. X is a value parameter which receives the value of its corresponding argument expression. JLT, JEQ, JGT are the point parameters which represent the point arguments L1, L2, and L3. During the first invocation of TEST, the value of the expression A (equal to -5) is passed and assigned to X. The addresses of the points L1, L3 and L2 are provided to JLT, JEQ and JGT respectively. Since X is equal to -5, the statement, GOTO JLT, passes control through JLT to its corresponding argument, point name L1. During the second invocation of TEST, the value of the expression A (now equal to 0) is passed and assigned to X. The address of the points L3, L2, and L1 are provided to JLT, JEQ and JGT respectively. Since X is now 0, the statement GOTO JEQ, passes control through JEQ to its corresponding argument, point name L2. During the final invocation of TEST, the value of the expression A + 1 (now equal + 1) is passed and assigned to X. The address of the points L2, L1, and L3 are provided to JLT, JEQ and JGT respectively. Since X is now equal to 1, the statement GOTO JGT, passes control through JGT to its corresponding argument, point name L3. Here the program EXAMPLE terminates.

CHAPTER 12
CONDITION DECLARATION

12-1. Condition

The condition declaration is available to the TACPOL user as a debugging aid. It specifies whether or not a snap procedure is to be called whenever a particular condition arises during execution of the block in which the declaration is contained. A snap procedure is a trace of the block in which the condition was detected.

a. The condition declaration is specified by the particular CHECK or IGNORE. The CHECK particle specifies that the snap procedure is to be invoked. The IGNORE particle specifies that the snap procedure is not to be invoked. The IGNORE particle is used to negate a condition which is invoked by the CHECK particle. The conditions which can be checked by a condition declaration are ZERO divide (ZDIV), fixed overflow (FOFL) and the USAGE particle which encompasses the checking of many conditions.

EXAMPLE: CHECK ZDIV;

b. Should a ZERO divide occur within the block in which the above condition declaration appears, a snap procedure is invoked.

EXAMPLE: IGNORE ZDIV;

c. Should a ZERO divide occur within the block in which the above condition declaration appears, a snap procedure is not invoked. The same coding techniques apply to FOFL.

12-2. USAGE

The USAGE particle requires a check name list. The list contains the names of quantities, procedures or points (in any kind of mixture) which have been defined within the program. A snap procedure is invoked whenever any of the following operations is performed:

a. A value is assigned by means of an assignment statement to a quantity identified by a simple, group or table scalar or array name contained in the associated name list.

b. A proper procedure is invoked by means of a CALL statement, where the proper procedure name is contained in the associated name list.

c. A function procedure is invoked by means of an expression evaluation, where the function procedure name is contained in the associated name list.

d. The sequence of execution is changed by means of a GOTO statement to a point identified by a point name contained in the associated name list.

```
EXAMPLE:    CHECK USAGE (TTY,
            BNT,
            APO, SICC);
```

If any of the names in the list are accessed or changed as specified by the rules of the USAGE particle, then the snap procedure would be invoked. The snap procedure can be cancelled for one or more names in the list by a subsequent IGNORE declaration.

```
EXAMPLE:    IGNORE USAGE (BNT,
            SICC);
```

CHAPTER 13
INPUT/OUTPUT

Section I. FUNCTION OF INPUT/OUTPUT

13-1. General

The basic function of input and output is data transmission: getting the data to be processed and returning the results of the processing. A programmer normally need write only the operation (e.g., READ, WRITE), the file name (see below), and a data name, that specifies where the data is to be stored or where the data to be written can be found.

13-2. Files

Data on an external medium is collected in a file. Files are defined by file declarations in the Compool. A file name can be declared in a program as a temporary device to avoid syntax errors. See Chapter 14 for actual syntax. A file name is declared for each file, and the file name is given file attributes that describe the data in the file and the manner in which it will be handled.

a. A file consists of one or more records, where a record is a set of quantities accessed in a single input/output operation. Files of quantities consist of storage allocated external to the primary memory of the computer. A file is either a partitioned file or a nonpartitioned file. A partitioned file consists of one or more partitions, each of which is a set of records within the file that may be accessed independently of records in other partitions of that file, as though they constituted a separate file. A partition of a file is accessed by means of a character string key, which specifies the partition currently to be accessed. Each partition within a file must have an unique key. Nonpartitioned files consist of no partitions and cannot be accessed by partition keys. The term partition can be substituted for file in the following text.

b. Two types of files are available in TACPOL. They are the serial file and the direct file.

(1) A serial file consists of records organized on the basis of their successive physical locations within the file. The records appear sequentially within the space allocated for the file and they are read or written sequentially. Serial files can exist on either a sequential or direct access storage device.

(2) A direct file contains records organized on the basis of a character string value (a 'key') associated with each record. This value has a limit of eight characters and is stored with the record. Records can be accessed directly by this value without regard to the actual position of the file. Direct files can exist only on a direct storage device.

c. A file may be accessed in one of three modes at any given time: INPUT, OUTPUT, or UPDATE. A file accessed for INPUT must be an existing file which is to be read but not written. A file accessed for OUTPUT must be a file which is to be written but not read. A file accessed for UPDATE must be a direct file and may be read or written. A file processing operation may transmit values to or from a file either before continuing execution of the program requesting such transmissions. or concurrently with the continued execution of the program. Normally the values will be transmitted before continuing execution of the program. However, by specification of a RETURN attribute for certain operations, the transmission occurs concurrently. If the transmission occurs concurrently, values involved in the transmission cannot be accessed by the program until the transmission is completed. A concurrent transmission is certain to be completed only at the point at which a 'wait' operation is executed for the file. For a transmission which is to be completed before continued execution of the program (RETURN attribute not specified), a 'wait' operation is understood to be executed immediately following the operation requesting the transmission.

Section II. INPUT/OUTPUT PROCESSING STATEMENTS

13-3. Processing Statements

The following file processing statements comprise the input/output operations in TACPOL.

13-4. OPEN Statements

OPEN statements are used to connect files to user programs so that file is available for processing. No data transfers take place as a result of this statement but the necessary linkage between the file and the user program is established. An OPEN statement consists of the following parts, some of which are optional in use:

a. OPEN. The particle which identifies an OPEN statement.

b. File Designation. The name of the file which is to be opened, mandatory.

c. Mode. Input, Output or Update, mandatory.

d. Origination. Either not used which specifies a new file or OLD which specifies the file has been previously created.

e. Disposition. Either KEEP which specifies that the file is to be kept after it is closed or, PASS which specifies that the file is to be kept and is to remain immediately available after being closed. If either of these two particles are not used (the particle being omitted from the statement) then the file is not kept after it is closed.

(1) In the example below, file SERIN is opened for input, it is an already existing file (required for the input mode) and it will be kept after the file is closed.

```
EXAMPLE:    OPEN SERIN INPUT
            OLD KEEP;
```

(2) In the example below, file DIRIN is opened for output, it is a new file (the particle OLD is absent from the statement) and the file will not be kept after the file is closed (the particles KEEP or PASS are absent from the statement. Only one file may be opened per OPEN statement. If several files are to be opened for program use each file requires a separate OPEN statement.

```
EXAMPLE:    OPEN DIRIN OUTPUT;
```

13-5. CLOSE Statements

A CLOSE statement serves to disconnect a file from a user program making that file unavailable for processing. No operation can be performed on a closed file except to open it. The CLOSE statement consists of the particle CLOSE followed by the file designation (name).

```
EXAMPLE:    CLOSE XYZ;
```

Only one file can be closed by a CLOSE statement. To close more than one file successive CLOSE statements must be used.

13-6. READ Statements

READ statements transmit values of a record in a designated file to a designated set of quantities. If the file is a serial file, it is repositioned after transmission so the next record to be accessed is the next record in the file. If the file is a direct file, the value yielded by the character string expression is the value of the key of the record to be read. Only files which have been opened for input or update can be read. The READ statement consists of the following parts, some of which are optional in use:

a. READ. Specifies the statement is a READ statement.

b. File Designation. The name of the file that is to be read.

c. Key Option. A character string expression which specifies the record to be read. The key option is used for direct files only.

d. INTO. A statement particle which is always present.

e. Quantity Designation. Specifies the quantity in memory into which the data will be transmitted.

f. Return Option. If the RETURN option is present it specifies concurrent operations. If the RETURN option is absent, noncurrent transmission is specified.

(1) In the example below serial file DESTIN is read (the key option is absent which specifies a direct file) into quantity TAB1. The RETURN option is specified for concurrent operations.

```
EXAMPLE:    READ DESTIN INTO
            TAB1 RETURN;
```

(2) The example below illustrates a read function for a direct file. Record ABC of file STREAM is read into quantity CELL. Nonconcurrent operation is specified by the absence of the RETURN option. For direct files the particle KEY must immediately precede the character string expression which denotes the record to be read. The example shows the character string expression in literal character format. Any expression may be used which yields a character string value for the record key.

EXAMPLE: READ STREAM KEY 'ABC' INTO CELL;

13-7. WRITE Statements

WRITE statements transmit values from a designated set of quantities to a new record added to the designated file. If the designated file is a serial file, it is repositioned after the transmission so that the next record to be accessed is the next record to be accessed. If the designated file is a direct file, the value yielded by the character string expression is the value of the key of the record to be written. Only files opened for output or update can be written. The WRITE statement consists of the following parts, some of which are optional in use:

a. WRITE. Specifies the statement is a WRITE statement.

b. File Designation. The name of the file that is to be written.

c. Key Option. A character string expression which specifies the record to be written. The key option is used for direct files only.

d. FROM. A statement particle which is present in most WRITE statements.

e. Quantity Designation. Specifies the quantity in memory from which the data will be transmitted.

f. Return Option. If present, specifies concurrent operations. If not present, specifies nonconcurrent operations.

g. ENDFILE. In a special write statement causes the terminal boundary of the file to be placed at the current position of the file. To use the ENDFILE particle the file must be a serial file which has been opened for output only.

(1) In the example below, the serial file ZETA (the key option is absent which specifies a direct file) is written into from quantity DELTA. The RETURN option is specified for concurrent operations.

EXAMPLE: WRITE ZETA FROM DELTA RETURN;

(2) The example below illustrates a write function for a direct file. Record ZZZ of file GAMMA is written from quantity IOTA. Nonconcurrent operations are specified by the absence of the RETURN option. The particle KEY must precede the character string expression. Any expression may be used for the record key which yields a character string value.

EXAMPLE: WRITE GAMMA KEY 'ZZZ' FROM IOTA;

(3) The special WRITE statement below places the terminal boundary of file LAMDA at the current position of the file.

EXAMPLE: WRITE LAMDA ENDFILE;

13-8. REWRITE Statements

REWRITE statements transmit values from a designated set of quantities to an already existing record in a designated file. Only files which are direct files that have been opened for update can be used in a REWRITE statement. REWRITE statements are coded exactly as WRITE statements are coded for direct files. The ENDFILE option is not available. The example below illustrates the rewriting of the already existing file SAM. Record MAX within the file is rewritten from quantity GEORGE. The RETURN option specifies concurrent operation. For nonconcurrent operation the RETURN option is not specified.

EXAMPLE: REWRITE SAM KEY 'MAX' FROM GEORGE RETURN;

13-9. DELETE Statements

DELETE statements cause already existing records in designated files to be removed from the files. The designated files must be direct files that have been opened for update. The DELETE statement consists of the following parts:

a. DELETE. Specifies the DELETE statement.

b. File Designation. The name of the file that contains the record to be deleted.

c. Record Key. The particle KEY followed by a character string expression which specifies the record to be deleted.

The example below illustrates the deletion of record KILO from file FOXTROT. No options are available for this statement.

```
EXAMPLE:    DELETE FOXTROT KEY
            'KILO';
```

13-10. SPACE Statements

The SPACE statement causes the designated file to be repositioned so that the next record to be accessed is either one record forward or one record backward from the current record position. The SPACE statement consists of the following parts:

a. SPACE. Specifies the SPACE statement.

b. File Designation. Specifies the file that is to be spaced.

c. Direction. If the particle BACK is present the designated file is positioned one record backward. If the particle BACK is absent the designated file is positioned one record forward.

(1) The example below spaces file ALLREC backward one record.

```
EXAMPLE:    SPACE ALLREC BACK;
```

(2) The example below spaces file ALLREC forward one record.

```
EXAMPLE:    SPACE ALLREC;
```

(3) If spacing of more than one record forward or backward is desired, it requires the use of more than one SPACE statement for the file.

13-11. REWIND Statements

The REWIND statement causes the designated file to be repositioned to the initial boundary of the file. The statement requires the particle REWIND and a file designation.

```
EXAMPLE:    REWIND EBCDIC;
```

13-12. UNWIND Statements

The UNWIND statement causes the designated file to be repositioned to the terminal boundary of the file. The statement requires the particle UNWIND and a file designation.

```
EXAMPLE:    UNWIND ASCII;
```

13-13. ON Statements

ON statements, like IF statements, specify the conditional execution of a constituent statement. The conditions to be met to execute the constituent statement are specified by particles ENDFILE, NOKEY or NOPART.

a. ENDFILE. If the particle ENDFILE is present, the file is examined to determine whether or not an attempt was made to reposition or to transmit a record from the file at a point beyond a boundary of that file. If the ENDFILE particle is used the file to be examined must be a serial file which was opened for input.

b. NOKEY. If the particle NOKEY is present, the file is examined to determine whether or not an attempt was made to read, rewrite or delete a record in the file when no record with the specified key exists; or, to write a record with a specified key which already exists. If the NOKEY particle is used the file to be examined must be a direct file.

c. NOPART. If the particle NOPART is present, the file is examined to determine whether or not an attempt was made to open an old partition in a file when that partition no longer exists. If the NOPART particle is used the file must be a partitioned file. The ON statement consists of the following parts:

(1) *ON.* Specifies the ON statement.

(2) *File Designation.* Specifies the file which is to be examined.

(3) *File Condition.* Specifies ENDFILE, NOKEY or NOPART.

(4) *THEN.* A particle which precedes the simple constituent statement.

(5) *Statement.* The statement to be executed if the stated condition in the examined file exists (GOTO, DO, BEGIN, etc.).

(6) *ELSE.* An arbitrarily used particle. If present, specifies the execution of an alternative statement if the ON condition is not satisfied.

(7) *Statement.* The statement to be executed if the ELSE alternative is used.

(a) In the example below, file TANGENT is examined for the ENDFILE condition. If the condition exists, the statement following the THEN particle is executed. If the condition does not exist the next statement in sequence is executed.

```
EXAMPLE:    ON TANGENT END-
            FILE THEN GOTO
            PART2;
```

(b) In the example below, file COSINE is examined for the NOKEY condition. If the condition exists, the statement following the THEN

particle is executed. If the condition does not exist, the statement following the ELSE particle is executed.

EXAMPLE: ON COSINE NOKEY
THEN SPACE COSINE;
ELSE GOTO PART3;

13-14. WAIT Statements

A WAIT statement causes a wait operation to be performed for the designated file. The continued execution of the program is delayed until all operations requested pertaining to the designated file have been completed. The statement requires the particle WAIT followed by a file designation.

EXAMPLE: WAIT GRP;

13-15. LOAD Statements

The LOAD statement causes the designated program to be made available for execution (loaded into memory). If an attempt is made to invoke a program which has not been previously loaded into memory, a load operation is performed before the program is executed. The statement requires the particle LOAD followed by a program name (procedure name).

EXAMPLE: LOAD JOE;

13-16. Permissible File Processing Operations

As a summary, table 13-1 on the following page lists the permissible file processing operations in TACPOL. The table is organized by origination (NEW or OLD), OPEN mode (INPUT, OUTPUT or UPDATE) and virtual organization (SERIAL or DIRECT). Missing from the list of operations are the WAIT statement and the LOAD statement. The WAIT statement is permissible for all files therefore it was not necessary to list the operation. The LOAD operation is not pertinent to files and is not included.

Table 13-1. Permissible File Processing Operations

Origination	Open Mode	Virtual[1] Organization	Read	Write	Write End File	Rewrite	Delete	Space	Rewind	Unwind	On Endfile	On Nokey	On Nopart
New	Output	Serial		X	X			X	X				
New	Output	Direct		X								X[3]	
New	Update	Direct	X	X		X	X					X[4]	
Old	Input	Serial	X					X	X	X	X[2]		X[6]
Old	Input	Direct	X									X[5]	X[6]
Old	Output	Serial		X	X			X	X	X			X[6]
Old	Output	Direct		X								X[3]	X[6]
Old	Update	Direct	X	X		X	X					X[4]	X[6]

NOTES:
1. Direct Files treated as Serial Files are included opposite SERIAL.
2. Arising from a READ or SPACE.
3. Arising from a WRITE.
4. Arising from a READ, WRITE, REWRITE or DELETE.
5. Arising from a READ.
6. Arising from a partition OPEN.

Each permissible operation for a given file is noted by an "X."

20083-12

CHAPTER 14
COMPOOL AND FILE DECLARATIONS

14-1. General

The name Compool ('Communication POOL') refers to a collection of names or quantities and programs that are commonly used by many different programs in a system. Placing these names in a central pool saves having to redeclare them each time that they are used in a new program. Procedures, declarations, data declarations, and file declarations may be included in the Compool. However, file declarations may not appear outside the Compool.

14-2. TACPOL Interface with the Compool

A Compool Generator will be used to generate the Compool tables to be made available to the TACPOL Compiler. Input to the Compool Generator consists of TACPOL like declarations and the output consists of binary and symbolic data. The TACPOL Compiler will refer to the Compool to define quantities and names which are not defined in a given program. Therefore the Compool is like a block in which all programs are contained.

14-3. File Declarations

As has been mentioned in earlier chapters, files, with the exception previously noted, can be declared only in the Compool. Before a file can be used by a program it must have been defined by input to the Compool Generator. The format for this input is a source language declaration as shown on the following page.

DCL identifier FILE	PARTS (partitions)	file type
1	2	3

RECORDS (No. of records) record type (No. of words)
4

LABELLED	BLOCKED (No. of words)
5	6

BUFFERED (No. of buffers)	media	classification
7	8	9

AUTH (authorization list and access)
10

a. This part of the declaration (1) specifies the file name.

EXAMPLE: DCL OPTIM FILE

b. The use of this part of the declaration (2) is not mandatory. It is only used if the file being declared is to be a partitioned file. The number of partitions in the file is specified by a number enclosed in parentheses.

EXAMPLE: PARTS (5)

c. The file type (3) is either SERIAL or DIRECT.

d. The number of records (4) in each file or partition is specified by a number enclosed in parentheses. The record type is either FIXED, VARIABLE or FREE. The maximum number of words in a record is specified by a number enclosed in parentheses.

EXAMPLE: RECORDS (4)
FIXED (32)

The particle LABELLED (5) is not mandatory. When used, it specifies the file is to be processed with standard header and trailer labels.

e. The use of this part (6) of the declaration is not mandatory. If used it specifies that logical records are grouped into physical blocks for actual I/O processing. The number of words in the block must be declared by specifying a number enclosed in parentheses.

EXAMPLE: BLOCKED (144)

f. The use of this part (7) of the declaration is not mandatory. If used it specifies the number of buffers to be allocated for I/O processing. The number is enclosed in parentheses.

EXAMPLE: BUFFERED (3)

g. The media (8) specifies the type of device that the file will be allocated to. One of the following may appear: TAPE, PRINTER, READER, PUNCH, DISPLAY, PLOTTER, CONSOLE, DASD, TERMINAL. In the case of TAPE, PRINTER and TERMINAL it is possible for a system to have more than one of these devices. To specify which device in a group is desired a number, enclosed in parentheses, follows the device name.

EXAMPLE: TERMINAL (3)

h. The classification (9) information is used to control the security of the file's contents. One of the following four classifications is assigned each file: UNCL (unclassified), CONF (confidential), SECR (secret), TOPS (top secret).

i. The authorization (10) specifies the program(s) which are authorized to open a file and the access which they are allowed. If more than one program is granted authorization in the declaration, the names of the programs appear in a list, separated from each other by commas, and enclosed in parentheses. The access to the file will be one of the following: INPUT, OUTPUT, UPDATE.

```
EXAMPLE:    AUTH (PPP,QQQ)
            OUTPUT;
```

(1) An example of a file declaration, not using all the options available, is illustrated below:

```
EXAMPLE:    DCL HIPT FILE
            DIRECT RECORDS
            (10) VARIABLE
            (200) TAPE (2)
            UNCL AUTH TPYO
            INPUT;
```

(2) The declaration specifies the nonpartitioned direct file HIPT with a maximum of 10 variable length records, a maximum of 200 words per record, which is allocated to tape 2. The file is unclassified and program TPYO has authorization for the file for input.

14-4. Special Notes On Files and File Declarations

The partitions of a partitioned file are not specified by name in the file declaration. The names of the partitions are specified in an OPEN statement, following the file name and enclosed in parentheses, when a partitioned file is opened.

```
EXAMPLE:    OPEN KKLO ('ABC')
            INPUT OLD KEEP;
```

a. A character expression, following the file name, indicates a partition of a file in an open statement.

b. Each partition in a file must have an unique key. Each partition in a file contains the same attributes as the file itself.

c. Record type can be either FIXED, VARIABLE or FREE. Even though FREE is accepted as a record type in a TACPOL statement, it is currently not implemented and has no meaning. Therefore, FIXED or VARIABLE only should be used for record types.

d. A device type available for file declarations is DASD. This stands for Direct Access Storage Device and means the drum or RAM (Random Access Memory) in the system. If the system has no drums for auxiliary storage then DASD should not be used.

APPENDIX A
INTRINSIC PROCEDURES

A-1. General

Certain proper and function procedures are understood to be defined in a text in which all TACPOL programs (and any Compool) are embedded. These procedures are described under seven headings: short numeric procedures, long numeric procedures, character string procedures, bit string procedures, structure procedures, point procedures, and conversion procedures.

A-2. Short Numeric Procedures

Short numeric procedures are all function procedures which yield short numeric values. In the following trigonometric short numeric procedures SIN and COS, X is in units of X and Y coordinates or Binary angular measurement units (BAMS) such that one complete revolution about 360 degrees is equal to one BAM with zero BAMS at true north and ±0.5 BAMS at 180 degrees (negative values to the left of north and positive values to the right). The result of the ASIN, ACOS and ATAN routines will be in these same units.

SIN (X) — This procedure computes the sine function of a short numeric value. The value argument X must be provided in units of angular measurement, such that $0 \leq X < 1$, with any scaling. The function SIN (X) will then be provided such that $-1 < SIN(X) < +1$, with the scaling of X.

COS (X) — This procedure computes the cosine function of a short numeric value. The value argument X must be provided in units of angular measurement, such that $\leq X < 1$, with any scaling. The function COS (X) will then be provided such that $-1 < COS(X) < +1$, with the scaling of X.

ASIN (X) — This procedure computes the arcsine function of a short numeric value. The value argument X must be provided such that $-1 < X < +1$, with any scaling. The function ASIN (X) will then be provided in units of angular measurement, such that $-0.25 \leq ASIN(X) \leq +0.25$, with the scaling of X.

ACOS (X) — This procedure computes the arccosine function of a short numeric value argument X must be provided such that $-1 < X < +1$, with any scaling. The function ACOS (X) will then be provided in units of angular measurement, such that $0 \leq ACOS(X) \leq 0.5$, with the scaling of X.

ATAN (X) — This procedure computes the arctangent of a short numeric value. The value argument X must be provided such that $-1 < ATAN(X) < +1$, with any scaling. The function ATAN (X) will then be provided in units of angular measurement, such that with $-0.125 \leq ATAN(X) \leq +0.125$, the scaling of X.

LN (X) — This procedure computes the natural logarithm of a short numeric value. The value argument X must be provided such that $0 < X$, with any scaling. The function LN (X) will then be provided, with the scaling of X. X must be such that $2^{22} < |LN(X)| < 2^{+9}$.

LOG (X) — This procedure computes the common logarithm of

a short numeric value. The value argument X must be provided such that $0<X$, with any scaling. The function LOG (X) will then be provided, with the scaling of X. X must be such that $2^{22}<|LOG(X)|2^{+9}$.

EXP (X) — This procedure computes the exponential function of a short numeric value. The value argument X must be provided such that $2^{22}<|X|2^{+9}$, with any scaling. The function EXP (X) will then be provided with the scaling of X.

SQRT (X) — This procedure computes the square root of a short numeric value. The value argument X must be provided such that $0\leq X$, with any scaling. The function SQRT (X) will then be provided, with the scaling of X.

REM (X,Y) — This procedure computes the remainder of the division of two short numeric values.

The value arguments X and Y may be provided with any scaling. The function REM (X, Y) will then be provided with scaling as follows:*

$$s_r = s_x$$

MAX (X, Y) — This procedure computes the larger of two short numeric values. The value arguments X and Y may be provided with any scaling. The function MAX (X, Y) will then be provided with scaling as follows:

$$s_r = \max(s_x, s_y)$$

If r requires greater than 31 significant bits of precision, the function is undefined.

MIN (X, Y) — This procedure computes the smaller of two short numeric values. The value arguments X and Y may be provided with any scaling. The function MIN (X, Y) will then be provided with scaling as follows:

$$s_r = \max(s_x, s_y)$$

If r requires greater than 31 significant bits of precision, the function is undefined.

ABS (X) — This procedure computes the absolute value of a short numeric value. The value argument X may be provided with any scaling. The function ABS (X) will then be provided with the scaling of X.

SIGN (X) — This procedure computes a numeric representation of the sign of a short numeric value. If $X<0$, then the function is -1, if $X = 0$, then the function is -1; if $X = 0$, then the function is 0; if $X>0$, then the function is +1. The value argument X may be provided with any scaling. The function SIGN (X) will then be provided with scaling as follows:

$$s_r = 0$$

SCALE (X, N) — This procedure rescales a short numeric value. The scale factor of the value of the first argument is changed to the number designated by the second argument. This results in no execute-time operations, only in compile-time considerations.
The value argument X may be provided with any scaling. The value argu-

**The notation uses s to represent the scale factor (s) of the result (r).*

ment N must be provided from an optionally signed number whose magnitude is greater than 0, but not greater than 127. The function SCALE (X, N) will then be provided with scaling as follows:

$$s_r = n$$

where

n is the value of N

TRUNC (X, N), TRUNC (X)

This procedure changes tne low-order precision of a short numeric value. Low-order bits are truncated (or added, zero-valued) to the value of the first argument, and the scale factor of the first argument changed, such that the value of the first argument retains the same magnitude but with a different low-order precision. The new scale factor is the number designated by the second argument. This results in binary shift operations. If no second argument occurs, the value of N is 0.

The value argument X may be provided with any scaling. The value argument Y must be provided from an optionally signed number whose magnitude is greater than 0, but not greater than 127. The function TRUNC (X, N) will then be provided with scaling as follows:

$$s_r = n$$

where

n is the value of N

If r requires greater than 31 significant bits of precision, the function is undefined.

ROUND (X, N), ROUND (X)

This procedure is exactly equivalent to TRUNC except that one-valued bit is added to the magnitude of the value of the argument (before any truncation) to the left-most low-order bit to be truncated, and that the scaling is as follows:

$$s_r = n$$

If r requires greater than 31 significant bits of precision, the function is undefined.

A-3. Long Numeric Procedures

The long numeric procedures are all function procedures which yield long numeric values.

REM (X, Y)

This procedure is entirely analogous to the short numeric function procedure of the same name, except that X and/or Y must be long numeric values.

MAX (X, Y)

This procedure is entirely analogous to the short numeric function procedure of the same name, except that X and/or Y must be long numeric values, and that the function is undefined only if r requires greater than 62 bits of precision.

MIN (X, Y)

This procedure is entirely analogous to the short numeric function procedure of the same name, except that X and/or Y must be long numeric values, and that the function is undefined only if r requires greater than 62 bits of precision.

ABS (X)

This procedure is entirely analogous to the Log Numeric Value X.

SHORT (X, N),

Short numeric function procedure of the same name, except that X must be a long numeric value.

SIGN (X) — This procedure is entirely analogous to the short numeric function procedure of the same name, except that X must be a long numeric value.

SCALE (X, N) — This procedure is entirely analogous to the short numeric function procedure of the same name, except that X must be a long numeric value.

TRUNC (X, N)
TRUNC (X) — This procedure entirely analogous to the short numeric function procedure of the same name, except that X must be a long numeric value, and that the function is undefined only if r requires greater than 62 bits of precision.

ROUND (X, N)
ROUND (X) — This procedure is entirely analogous to the short numeric function procedure of the same name, except that X must be a long numeric value, and that the function is undefined only if r requires greater than 62 bits of precision.

A-4. Character String Procedures

Character string procedures are all function procedures which yield character string values.

REP (X, N) — This procedure catenates one or more copies of a character string value. The second argument designates the number of copies of the first argument that are to be catenated. The value argument X may be any length. The value argument N must be greater than 0. The function REP (X, N) will then be provided with a length equal to the product of N and the length of X. If the length of the function would then be greater than 512 characters, sufficient characters are truncated from the right end of the function such that the length of the function is 512.

A-5. Bit String Procedures

The bit string procedures are all function procedures which yield bit string values.

REP (X, N) — This procedure catenates one or more copies of a bit string value. The second argument designates the number of copies of the first argument that are to be catenated.

The value argument X may be any length. The value N must be greater than 0. The function REP (X, N) will then be provided with a length equal to the product of N and the length of X. If the length of the function would then be greater than 32 bits, sufficient bits are truncated from the right end of the function such that the length of the function is 32.

BOOL (X, Y, N) — This procedure computes a Boolean result of the combination of two bit string values according to a truth table. The values of the first and second arguments are combined according to the truth table designated by the third argument. The third argument consists of four bits. The first bit denotes the value derived by the combination of two zero-valued bits; the second bit denotes that for a zero-valued bit and a one-valued bit; the third denotes that for a one-valued bit and a zero-valued bit; and the fourth denotes that for two one-valued

bits. The combination is determined for each successive pair of bits, one from each of the first two argument values, from right to left, starting with the left-most bit in each. If the lengths of the two arguments are not identical, sufficient zero-valued bits are appended before the combination onto the right end of the shorter value such that its length is identical to that of the longer. The value arguments X and Y may be provided with any length. The value argument N must be provided as a bit string literal of length 4. The function BOOL (X, Y, N) will then be provided with a length equal to the length of the longer of X and Y.

LETTER (X) — This procedure determines whether or not the left-most character in a character string value is a letter (A through Z). The value argument X may be provided as a character string of any length. The function LETTER (X) will then be provided as a bit string of length 1, zero-valued if the left-most character of X is not a letter, one-valued if it is.

DIGIT (X) — This procedure determines whether or not the left-most character in a character string is a digit (0 through 9). The value argument may be provided as a character string of any length. The function DIGIT (X) will then be provided as a bit string of length 1, zero-valued if the left-most character of X is not a digit, one-valued if it is.

A-6. Structure Procedures

Structure procedures manipulate the values of sets of quantities as single entities.

MOVE (X, Y) — This proper procedure assigns the values of the set of quantities designated by the first argument to the set of quantities designated by the second argument, as though both arguments designated bit string quantities whose lengths are the shorter of the two sets of quantities. X and Y must be quantity arguments.

CLEAR (X) — This proper procedure assigns a bit string value of indefinite length composed entirely of zero-valued bits to the set of quantities designated by the argument, as though the argument designated a bit string quantity whose length is the length of the set of quantities. X is a quantity argument.

A-7. Point Procedures

Point procedures transfer the sequence of execution according to specific rules.

SWITCH (X) P_1, P_2,..., P_n) — This proper procedure transfers the sequence of execution to the point designated by the point arugment P_i, where i is the largest integer not greater than the value of the short numeric argument X. If i is zero, the effect of the invocation of this procedure is exactly that of the execution of a null statement. The value argument X may be provided with any precision and scaling, but the value of the argument must not be less than 0 nor greater than n, where n is the number of point arguments in the argument list.

A-8. Redefinition Attribute Procedures

The redefinition attribute procedures are listed below.

SHORT (X, N),
SHORT (X)

Long Numeric Value X. This function procedure redefines the value of the long numeric argument X to a short numeric function value. The low-order N bits of precision of X are retained as the precision of the function. The scale factor of X is the scale factor of the function.

X must be a value argument of any precision and scaling. N must be a value argument which is a number greater than 0 and not greater than 31. If the argument N is not provided, N is understood to be 31. If X requires greater than N significant bits of precision, the function is undefined.

SHORT (X, N),
SHORT (X)

Character String Value X. This function procedure redefines the value of the character string argument X to a short numeric function value. The high-order N bits of X are retained as the precision of the function. If the length in bits of X is not N, sufficient bits are added (zero-valued) or truncated from the right such that its length in bits is N. The scale factor of the function is 0.

X must be a value argument of any length. N must be a value argument which is a number greater than 0 and not greater than 31. If the argument N is not provided, N is understood to be 8.

SHORT (X, N),
SHORT (X)

Bit String Value X. This function procedure redefines the value of the bit string argument X to a short numeric function value. The high-order N bits of X are retained as the precision of the function. If the length of X is not N, sufficient bits are added (zero-valued) or truncated from the right such that its length is N. The scale factor of the function is 0.

X must be a value argument of any length. N must be a value argument which is a number greater than 0 and not greater than 31. If the argument N is not provided, N is understood to be 1.

LONG (X, N),
LONG (X)

This function procedure redefines the value of the short numeric argument X to a long numeric function value. The low-order N bits of precision of X are retained as the precision of the function. The scale factor of X is the scale factor to the function.

X must be a value argument of any precision and scaling. N must be a value argument which is a number greater than 0 and not greater than 31.

If the argument N is not provided, N is understood to be 31. If X requires greater than N significant bits of precision, the function is undefined.

LONG (X, N),
LONG (X)

Character String Value X. This function procedure redefines the value of the character string argument X to a long numeric function value. The high-order N bits of X are retained as

the precision of the function. If the length in bits of X is not N, sufficient bits are added (zero-valued) or truncated from the right such that its length in bits is N. The scale factor of the function is 0.

X must be a value argument of any length. N must be a value argument which is a number greater than 0 and not greater than 62. If the argument N is not provided, N is understood to be 8.

LONG (X, N), LONG (X) Bit String Value X. This function procedure redefines the value of the bit string argument X to a long numeric function value. The high-order N bits of X are retained as the precision of the function. If the length of X is not N, sufficient bits are added (zero-valued) or truncated from the right such that its length is N. The scale factor of the function is 0. X must be a value argument of any length. N must be a value argument which is a number greater than 0 and not greater than 32. If the argument N is not provided, N is understood to be 1.

CHAR (X, N), CHAR (X) Short Numeric Value X. This function procedure redefines the magnitude of the value of the short numeric argument X to a character string function value. The low-order N bits of precision of X are retained, and sufficient bits added (zero-valued) to the right such that N is an even multiple of 8. This multiple is the length of the function. X must be a value argument of any precision and scaling. N must be a value argument which is a number greater than 0 and not greater than 31. If the argument N is not provided, N is understood to be 8.

CHAR (X, N), CHAR (X) Long Numeric Value X. This function procedure redefines the magnitude of the value of the long numeric argument X to a character string function value. The low-order N bits of precision of X are retained, and sufficient bits added (zero-value) to the right such that N is an even multiple of 8. This multiple is the length of the function.

X must be a value argument of any precision and scaling. N must be a value argument which is a number greater than 0 and not greater than 62. If the argument N is not provided, N is understood to be 8.

CHAR (X, N), CHAR (X) Bit String Value X. This function procedure redefines the value of the bit string argument X to a character string function value. The high-order N bits of X are retained as the length in bits of the function. If the length of X is not N, sufficient bits are added (zero-valued) or truncated from the right such that its length is N. If N is not then an even multiple of 8, sufficient bits are added (zero-valued) such that N is an even multiple of 8. This multiple is the length of the function.

X must be a value argument of any length. N must be a value argument

which is a number greater than 0 and not greater than 32. If the argument N is not provided, N is understood to be 8.

BIT (X, N), BIT (X)

Short Numeric Value X. This function procedure redefines the magnitude of the value of the short numeric argument X to a bit string function value. The low-order N bits of precision of X are retained as the length of the function. X must be a value argument of any precision and scaling. N must be a value argument which is a number greater than 0 and not greater than 31. If the argument N is not provided, N is understood to be 1.

BIT (X, N), BIT (X)

Long Numeric Value X. This function procedure redefines the magnitude of the value of the long numeric argument X to a bit string function value. The low-order N bits of precision of X are retained as the length of the function. X must be a value argument of any precision and scaling. N must be a value argument which is a number greater than 0 and not greater than 32. If the argument N is not provided, N is understood to be 1.

BIT (X, N), BIT (X)

Character String Value X. This function procedure redefines the value of the character string argument X to a bit string function value. The high-order N bits of X are retained as the length of the function. If the length in bits of X is not N, sufficient bits are added (zero-valued) or truncated from the right such that its length in bits is N.

X must be a value argument of any length. N must be a value argument which is a number greater than 0 and not greater than 32. If the argument N is not provided, N is understood to be 8.

APPENDIX B
PARTICLES AND WORD OPERATORS

B-1. General

Listed in this appendix are the particles, and operators constructed as words, comprising the TACPOL language. Word operators are reserved words which may not be utilized as programmer defined names. These operators are identified with an asterisk to the left of the word.

ALIGNED	attribute - specifies that storage for a declared set of quantity is to be allocated so as to minimize the time required for access.
AND	logical operator - indicates a logical 'and' of the bit string values immediately preceding and following the particle.
AUTH	file specification - specifies the programs (by name) which may access the file to perform read, write or both read and write operations.
B	bit string specifier - designate a bit string.
BACK	space statement specification - specifies that the designated serial file is to be spaced backward.
BEGIN	block delimiter - indicates the start of a BEGIN block.
BIN	attribute - specifies that values of the declared quantities are to be 'binary' representations of decimal numbers (must be used with FIXED: BIN FIXED).
BIT	attribute - specifies that the quantities in a declaration are 'bit' string quantities.
BLOCKED	file specification - specifies records to be grouped into physical blocks for actual I/O operations.
BUFFERED	file specification - specifies the number of buffers to be allocated for I/O processing.
BY	DO specifier - identifies the value immediately following the particle as the step value (increment) of the control quantity in the DO statement.
CALL	statement - specifies that the proper procedure whose name appears immediately following the particle is to be invoked.
CAT	string operator - specifies that the string values immediately preceding and following the operator are to be 'catenated'.
CELL	attribute - identifies a declaration as a CELL declaration.
CHAR	attribute - specifies that the quantities in a declaration are 'character' string quantities.
CHECK	statement - specifies the condition declaration which turns specified conditions on (ZDIV, FOFL, USAGE).
CLOSE	statement - disassociates a file from the program.
CODE	block delimiter - identifies the block following the particle as being statements in a non-TACPOL language (currently assembly language).

COMPOOL — attribute - used in a procedure head to designate that procedure as being in the common pool.

CONF — file attribute - specifies that the data on the designated file is to be classified confidential.

CONSOLE — file attribute - specifies that the designated file is to use the console.

DASD — file attribute - Direct Storage Access Device. Specifies that the designated file is to use the DASD as the storage medium. DASD is a drum.

DCL — statement - identifies the text following the particle and ending with a semicolon as being a 'declaration'.

DELETE — statement - specifies that the existing record in the file is to be removed.

DIRECT — file attribute - specifies that the records on the file are to be organized to be accessible by a key value; i.e., specific records may be accessed regardless of their position relative to other records in the file.

DISPLAY — file attribute - specifies that the designated file is to use the display system.

DO — statement - specifies the text following the DO particle and delimited by the associated END particle as being the body of the DO statement, to be executed a number of times as indicated by the control variable and the DO specifiers.

E — literal specification - specifies the precision of the number. (none)

*ELSE — ELSE clause specifier - identifies the start of the ELSE clause in an IF statement.

*ELSE — ELSE clause specifier - identifies the start of the ELSE clause in an ON statement.

END — block delimiter - indicates the 'and' of any block for which there is an associated particle indicating the start of that block. (See BEGIN, DO, PROC.)

ENDFILE — I/O specification - in a WRITE statement, causes the terminal boundary or the file to be placed at the current position of the file. The file must be a serial file opened for output.

ENDFILE — I/O specification - in an ON statement, the file is examined to determine whether or not an attempt was made to reposition or to transmit a record from a serial file, opened for input, at a point beyond the boundary of the file.

ENTRY — attribute - specifies that the names listed in the declaration are names of proper procedures which are parameters to the procedure in which the declaration appears.

FILE — I/O statement specifier - used in an input/output statement or declaration as part of the text.

FIXED — attribute - specifies the quantities in a declaration are 'fixed' point (must be used with BIN: BIN FIXED).

FIXED — file specification - specifies a file of fixed length.

FOFL — condition - specifies that whenever a 'fixed point overflow' occurs as the result of an arithmetic operation, the 'snap' procedure

	is to be implicitly invoked or not invoked (see SNAP).
FREE	attribute - a record type attribute for file declarations not implemented.
FROM	REWRITE statement specifier - specifies the set of quantities from which the record is to be rewritten.
FROM	WRITE statement specifier - specifies the set of quantities from which the record is to be written.
GE	relational operator - specifies that a determination is to be made as to whether the value preceding the particle is greater than or equal to the value following the particle.
GOTO	statement - specifies control is to be transferred to the indicated point.
GT	relational operator - specifies that a determination is to be made as to whether the value preceding the particle is greater than the value following the particle.
IF	statement - specifies conditional execution of the statement in the THEN clause or ELSE clause as controlled by the outcome ('true' or 'false') of the bit expression of the IF clause.
IGNORE	statement - specifies the condition declaration which turns specified conditions off (ZDIV, FOFL, USAGE).
INIT	attribute - defines the value following the particle as the value to be assigned to the quantity in the declaration. ('initialize').
INPUT	file attribute - used in an OPEN statement to specify that the file being opened is to be used for input only.
INTO	READ statement specifier - specifies the set of quantities into which the record is to be read.
KEEP	I/O specification - specifies that the file is to be kept after being closed.
KEY	input/output specification - specifies the character expression to be used as the key value record to be accessed from the file.
L	literal specification - specifies that the literal is to be treated as a long numeric literal.
LABEL	attribute - specifies that the names listed in a declaration are point names which are parameters to the procedure that contains the declaration.
LABELLED	file specification - specifies a file with standard header and trailer labels.
LE	relational operator - specifies that a determination is to be made as to whether the value preceding the particle is less than or equal to the value following the particle.
LOAD	statement - specifies that the designated programs are to be loaded into the computer and prepared for execution.
LT	relational operator - specifies that a determination is to be made as to whether the value preceding the particle is less than the value following the particle.

NE — relational operator - specifies that a determination is to be made as to whether the values preceding and following the particle are not equal.

NOKEY — I/O specification - in an ON statement, the file is examined to determine whether or not an attempt was made to read, rewrite or delete a record in a direct file when no record with the specified key exists.

NOPART — I/O specification - in an ON statement, a file is examined to determine whether or not an attempt was made to open an old partition in the file and the partition no longer exists.

*NOT — logical operator - indicates a logical 'not' of the bit string values immediately preceding and following the particle.

OLD — I/O specification - specifies that the file in the I/O operation has been created before being opened.

ON — statement - a file statement which specifies the conditional execution of a constituent statement based upon the results of specified file conditions (ENDFILE, NOKEY or NOPART).

OPEN — statement - specifies that the named files or file set members are to be opened for use. The process of opening a file involves associating a file with the program by name and specifying certain attributes for the file.

OR — logical operator - indicates a logical 'or' of the bit string values immediately preceding and following the particle.

OUTPUT — file attribute - used in an OPEN statement to specify that the file being opened is to be used for output only.

PACKED — attribute - specifies that storage for a declared set of quantities is to be allocated so as to minimize the total storage required.

PARTS — file specification - specifies a file which is to be a partitioned file.

PASS — I/O specification - specifies that the file is to be kept and is to remain immediately available after being closed.

PLOTTER — file attribute - specifies that the designated file is to use the plotter.

PRINTER — file attribute - specifies that the designated file is to use the line printer.

PROC — block delimiter - indicates the start of a 'procedure' block.

PUNCH — file attribute - specifies that the designated file is to use the card punch.

READ — statement - specifies that a record is to be read into the specified set of quantities from the specified file.

READER — file attribute - specifies that the designated file is to use the card reader.

RECORDS — file specification - specifies the number of records in a file in a file declaration.

RETURN — statement - indicates that the value following the particle is to be 'passed back to' the invoking function procedure.

REWIND — statement - specifies that the designated serial file is

	to be rewound so that it is in its starting position.
REWRITE	statement - specifies that the specified set of quantities is to be written onto a file as a record replacing an existing record on that file.
S	literal specification - specifies the scale factor to be used for the literal.
SECR	file attribute - specifies that the data on the designated file is to be classified secret.
SERIAL	file attribute - specifies that the data on the file is to be organized in a serial fashion, i.e., accessed sequentially one record following another, etc.
SPACE	statement - causes the designated serial file to be spaced a specified number of records forward or backward.
*SUBSTR	string operator - designates a substring of a string quantity (specified by the text enclosed in parenthesis following the operator).
TAPE	file attribute - specifies that the designated file is to use magnetic tape.
TERMINAL	file attribute - specifies that the designated file is to use a data terminal.
*THEN	THEN clause specifier - identifies the start of the THEN clause in an IF statement.
TO	DO specifier - identifies the value immediately following the particle as the maximum allowable value of the control quantity in a DO statement.
TOPS	file attribute - specifies that the data on the designated file is to be classified top secret.
UNCL	file attribute - specifies that the data on the designated file is to be unclassified.
UPDATE	file attribute - used in an OPEN statement to specify that the file being opened is to be used for input and/or output.
UNWIND	statement - specifies that the designated serial file is to be positioned at its end.
USAGE	condition - in the CONDITION declaration specifies that any use of the names listed following the particle is to cause the snap (trace) procedure to be invoked or not invoked. (See CHECK and IGNORE.)
USES	attribute - in a CODE block, specifies the names in the list that follows will be known to the assembly language code within the block.
VALUE	attribute - specifies that the names listed in the declaration are parameter quantities which are implicitly assigned the values of the corresponding expression arguments on invocation of the procedure containing the declaration.
VARIABLE	file specification - specifies a file of variable length.
WAIT	statement - causes all program execution to wait until all file processing operations on the designated file are completed.
WHILE	DO specifier - identifies a bit expression which follows the particle in a DO statement. The DO statement is executed until the bit expression is no longer 'true'.

WRITE — statement - specifies that a record is to be written from the specified set of quantities onto the specified file.

ZDIV — condition - specifies that whenever an attempt is made to 'divide by zero' the 'snap' procedure is to be implicitly invoked or not invoked (see SNAP).

APPENDIX C
TABLE OF INTEGER PRECISION

Table C-1. Table of Integer Precision

If the Decimal Integer is Greater Than	And Less Than or Equal To	Then the Number of Bits Required to Contain the Decimal Integer is
0	1	1
1	3	2
3	7	3
7	15	4
15	31	5
31	63	6
63	127	7
127	255	8
255	511	9
511	102 3	10
1 023	204 7	11
2 047	409 5	12
4 095	819 1	13
8 191	163 83	14
16 383	327 67	15
32 767	655 35	16
65 535	131 071	17
131 071	262 143	18
262 143	524 287	19
524 287	104 857 5	20
1 048 575	209 715 1	21
2 097 151	419 430 3	22
4 194 303	838 860 7	23
8 388 607	167 772 15	24
16 777 215	335 544 31	25
33 554 431	67 108 863	26
67 108 863	134 217 727	27
134 217 727	268 435 455	28
268 435 455	536 870 911	29
536 870 911	107 374 182 3	30
1 073 741 823	214 748 364 7	31
2 147 483 647	461 068 601 832 629 990 3	62

20083-13

APPENDIX D
SAMPLE COMPILER OUTPUTS

D-1. General

This appendix contains samples of compiler outputs such as TACPOL Source Listing (table D-1), Attribute and Reference List (table D-2), Machine Language Output (table D-3), and Cross Reference and Set Used Listing (table D-4).

Table D-1. TACPOL Source Listing

```
TACPOL SOURCE LISTING          PAGE   1

 1          MESSP: PROC;                                                    HLR00020
 2                 /*DECLARATIVES FOR REQUIRED DATA STRUCTURES*/            HLR00030
 3                 DCL 1 INPM CELL,                                        HLR00040
 4                     2 (COORD (15),          /*DESIGN OF INPUT MESSAGE TABLE*/ HLR00050
 5                     3 (DESIG CHAR (4),                                  HLR00060
 6                        XCOORD BIN FIXED (15),                           HLR00070
 7                        YCOORD BIN FIXED (15)),                          HLR00080
 8                       NUMERIC (1),                                      HLR00090
 9                     3 (DESIGN BIN FIXED (31))));   /*OVERLAY OF DESIG*/  HLR00100
10                 DCL COUNT BIN FIXED;              /*ERROR MESSAGE COUNTER*/ HLR00110
11                 DCL 1 ERRM (15),                  /*ERROR MESSAGE AREA*/  HLR00120
12                     2 (ERRDES CHAR (4),                                 HLR00130
13                        ERRXCD BIN FIXED (15),                           HLR00140
14                        ERRYCD BIN FIXED (15));                          HLR00150
15                 DCL ACHAR CHAR (4) INIT ('AA  '); /* SOURCE DESIGNATOR*/ HLR00160
16                 DCL BCHAR CHAR (4) INIT ('BB  '); /*CONSTANTS*/         HLR00170
17                 DCL CCHAR CHAR (4) INIT ('CC  ');                       HLR00180
18                 DCL INDEX BIN FIXED (1 );         /*INDEX               /HLR00190
19                 /* START DYNAMIC  STATEMENTS*/
20                 COUNT = 0;                /* INITIALIZE ERROR COUNT */  HLR00230
21                 INDEX = 1;                /*INITIALIZE ERROR INDEX*/    HLR00240
22                 DO I=1 BY 1 TO 15;        /*BEGIN LOOP*/                HLR00250
23                   IF (DESIG(I) = ACHAR) THEN GOTO NEXT;  /*DESIGNATOR =S AA*/ HLR00260
24                   IF (DESIG(I) = BCHAR) THEN GOTO NEXT;  /*DESIGNATOR =S BB*/ HLR00270
25                   IF (DESIG(I) = CCHAR) THEN GOTO NEXT;  /*DESIGNATOR =S CC*/ HLR00280
26                   COUNT = COUNT+1;        /*START ERROR PROCESSING*/    HLR00290
27                   ERRDES(INDEX) = DESIG(I);    /*TRANSFER MESSAGE IN ERROR*/ HLR00300
28                   ERRXCD(INDEX) = XCOORD(I);                            HLR00310
29                   ERRYCD(INDEX) = YCOORD(I);                            HLR00320
30                   DESIGN(I), XCOORD(I), YCOORD(I) = 0; /*ZERO INPUT MESSAGE*/ HLR00330
31                   INDEX = INDEX+1;        /*SET INDEX FOR NEXT ENTRY*/  HLR00340
32          NEXT:  END;      /*DO LOOP*/                                   HLR00350
33                 END;      /*PROCEDURE*/                                 HLR00360
```

20083-14

Table D-2. Attribute and Reference List

```
ALPHAR      LITERAL VALUE  CHARACTER STRING(  4)
            DEFINED AT   15  REFERENCES AT   23

ALPHFAST    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHFIRE    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHFREL    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHP       GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHSLTR    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHSLOW    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHSYS     GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHXMT1    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHXMT2    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHXMT3    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPHXMT4    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALPLINGC    GROUP SCALAR  SHORT NUMERIC( 7,   0)  IN GROUP QTOP
            DEFINED IN COMPOOL  NO REFERENCES

ALR         CELL
            DEFINED IN COMPOOL  NO REFERENCES
```

20083-15

```
1/13/71          MESSP                                                    COMPOOL MSGPROC          PAGE    2

ADDR   OP M R S  D AW R P T  K  STMT  NAME   OPERATION  OPERAND            COMMENTS                 SEQUENCE

                                    3          CMP        MSGPROC                                  1   3
                                    4          SHIFT                                               1   4
                                    5 +        SHIFT      SET REG 15 FOR ILLEGAL OP PROCESSING                 +
0000   20 2 F 0  008E               6 +        LDF        9BADOP-$,15                               +
                 0000               7  9D0001  EQU        00000                                    3   1
                 0000               8  9D0002  EQU        00000                                    4   1
                 0000               9  9D0003  EQU        00000                                    5   1
                 0002              10  9D0004  EQU        00002                                    6   1
                 0003              11  9D0005  EQU        00003                                    7   1
                 0000              12  9D0006  EQU        00000                                    8   1
                 0000              13  9D0007  EQU        00000                                    9   1
                 003C              14  9D0008  EQU        00060                                   10   1
                 003E              15  9D0009  EQU        00062                                   11   1
                 003E              16  9D0010  EQU        00062                                   12   1
                 0040              17  9D0011  EQU        00064                                   13   1
                 0041              18  9D0012  EQU        00065                                   14   1
                 007A              19  9D0013  EQU        00122                                   18   1
0002   27 1 C 2  003C              20             MZF        9D0008+R'02'                            20   1
0004   20 0 7 0  0001              21             LDF        =00001,07                               21   1
0006   26 1 7 2  007A              22             SDF        9D0013+R'02',07                         21   2
                 007C              23  9D0014  EQU        00124                                   22   1
0008   20 0 7 0  0001              24             LDF        =00001,07                               22   2
000A   66 1 7 2  007C              25             SDH        9D0014+R'02',07                         22   3
000C              0000              26  9P0031  BSS        0                                       22   4
                                   27             DCHKP      9D0014,00015,SBF,9P0030                 22   5
                                   28 +           DCHK            DCHKP - BY IS POSITIVE LITNUM                +
                                   29 +           DCHK       FOR DCHKP AND DCHKS , OP IS 'SBE' FOR TO POSITIVE    +
                                   30 +           DCHK       AND IS 'ADF' FOR TO NEGATIVE                         +
                                   31 +           DCHK                                 (TO NOT LITNUM)         +
000C   6C 1 E 2  007C              32 +           LDH        9D0014+R'2',14                               +
000E   09 0 E 0  000F              33 +           SBF         =00015,14                                   +
0010   32 2 E 0  0002              34 +           XEF        $+2,14                                       +
0012   33 2 E 0  0076              35 +           XEF        9P0030-$,14                                  +
0014   60 1 7 2  007C              36             LDH        9D0014+R'02',07                         23   1
0016   0C 0 7 0  0004              37             MPF        =+00004,07                              23   2
0018   20 1 1 0  000E              38             LDF        R'07',01                                23   3
001A   20 2 E 2  FFFC              39             LDF        9D0003-00004+D'02',08                   23   4
001C   20 2 E 0  00BE              40             LDF        9L0016-$,06                             23   5
001E   12 1 8 0  000C              41             CLF        R'06',08                                23   6
0020   71 2 2 0  0062              42             XEQ        9P0024-$                                23   7
0022              0000              43  9P0032  BSS        0                                       24   1
0022   60 1 7 2  007C              44             LDH        9D0014+R'02',07                         24   2
0024   0C 0 7 0  0004              45             MPF        =+00004,07                              24   3
0026   20 1 1 0  000E              46             LDF        R'07',01                                24   4
0028   20 2 E 2  FFFC              47             LDF        9D0003-00004+D'02',08                   24   5
002A   20 2 E 0  00B2              48             LDF        9L0017-$,06                             24   6
002C   12 1 8 0  000C              49             CLF        R'06',08                                24   7
002E   71 2 2     0054              50             XEQ        9P0025-$                                24   8
0030              0000              51  9P0033  BSS        0                                       25   1
0030   60 1 7 2  007C              52             LDH        9D0014+R'02',07                         25   2
```

20083-16

Table D-4. Cross Reference and Set Usec Listing

CROSS REFERENCE AND SET-USED LISTING

SYMBOL	REF	VALUE	CROSS-REFERENCE							
9BADDP	USED	0090	0006							
9DD001	NONE	0000								
9DD002	NONE	0000								
9DD003	USED	0000	0039	0047	0055	0067				
9DD004	BOTH	0002	0081	*0093						
9DD005	BOTH	0003	0085	*0095						
9DD006	NONE	0000								
9DD007	SET	0000	*0091							
9DD008	SET	003C	*0020	*0061						
9DD009	NONE	003E								
9DD010	USED	003E	0073							
9DD011	SET	0040	*0083							
9DD012	SET	0041	*0087							
9DD013	BOTH	007A	*0022	0068	*0097					
9DD014	BOTH	007C	*0025	0032	0036	0044	0052	0062	0088	*0101
9EN	NONE	00DC								
9L0016	USED	00DC	0040							
9L0017	USED	00DE	0048							
9L0018	USED	00E0	0056							
9MB	BOTH	0080	*0119	0129	*0135	0150	0152	0153		
9MBK	NONE	00D8								
9MBLOP	USED	00BC	0151							
9MBYTE	USED	0092	0079							
9M01	NONE	00BE								
9P0015	USED	0084	0104	0105	0106					
9P0024	USED	0084	0042							
9P0025	USED	0084	0050							
9P0026	USED	0084	0058							
9P0030	USED	008A	0035							
9P0031	USED	00CC	0102							
9P0032	NONE	0022								
9P0033	NONE	0030								
9P0034	NONE	003E								
9RTHREE	NONE	007E								

20083-17

DATA PROCESSING
TACPOL KEY PUNCH TRANSMITTAL

PAGE ____ OF _____
DATE _____
PROGRAM _____

PROGRAMMER: EXT:

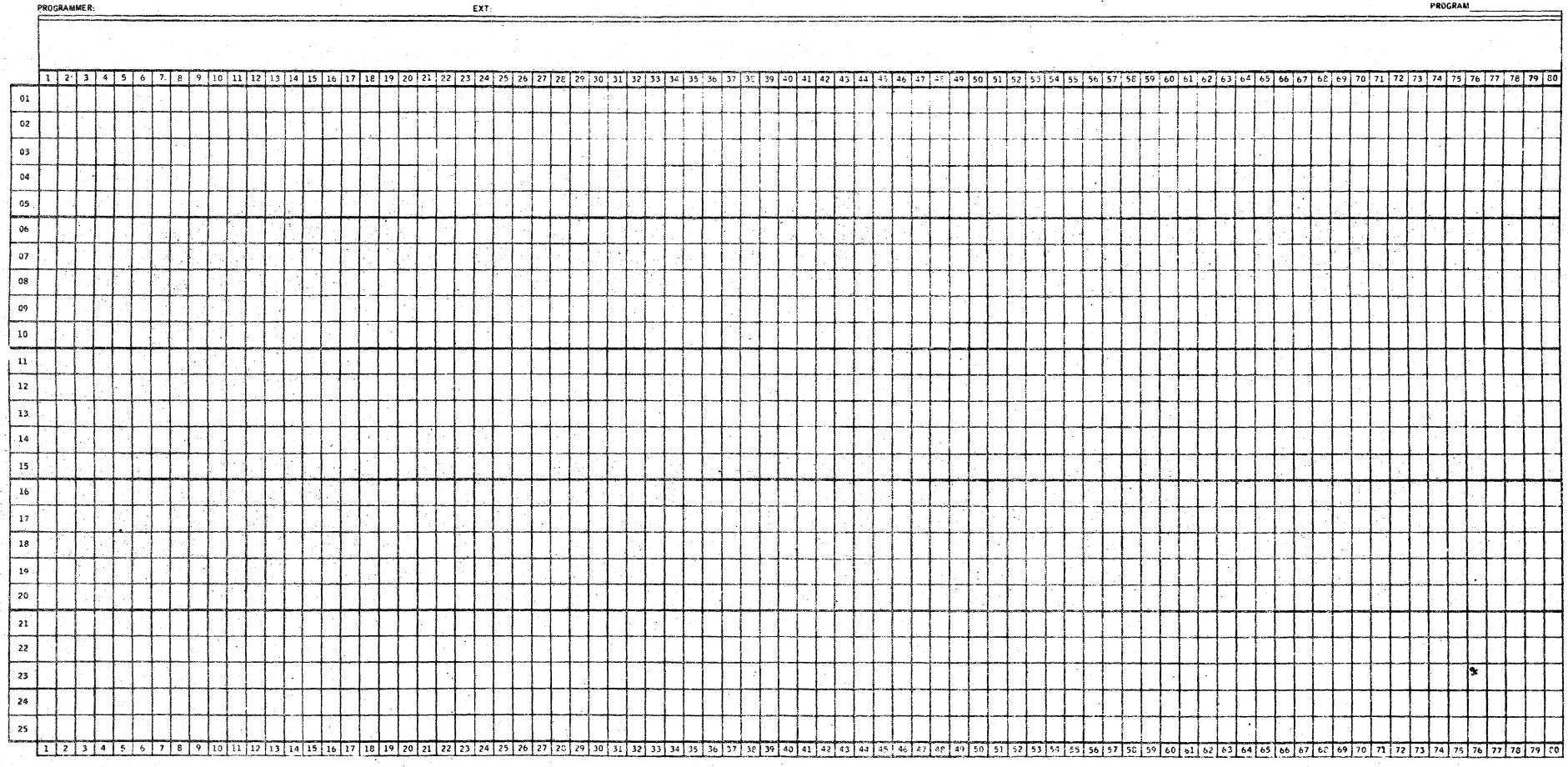


Figure FO-1. TACPOL Coding Form

44-49-005