PHILCO
A SUBSIDIARY OF Ford Motor Company
COMPUTER
DIVISION

ELECTRONIC
DATA PROCESSING
SYSTEMS

PHILCO 2000

ALGEBRAIC PROGRAMMING LANGUAGE

ALTAC III

TM-27

PHILCO 2000

ALGEBRAIC PROGRAMMING LANGUAGE

ALTAC III

January 1963

3/21/63 errata added 4/63

12/31/63 errata added

PHILCO CORPORATION
A SUBSIDIARY OF Ford Motor Company
Computer Division • 3900 Welsh Road
Willow Grove, Pennsylvania

This manual replaces manual TM-5C.

PREFACE

This manual is a comprehensive description of the Philco 2000 Algebraic Programming Language, ALTAC III (hereinafter referred to as ALTAC). It discusses the rules which must be followed when writing programs in ALTAC language.

No previous programming experience is assumed for an understanding of the material presented herein; however, a knowledge of the Philco 2000 TAC Language would be helpful.

CONTENTS

CONTENTS (Cont'd)

CONTENTS (Cont'd)

Page

Chapter

Appendices

INTRODUCTION

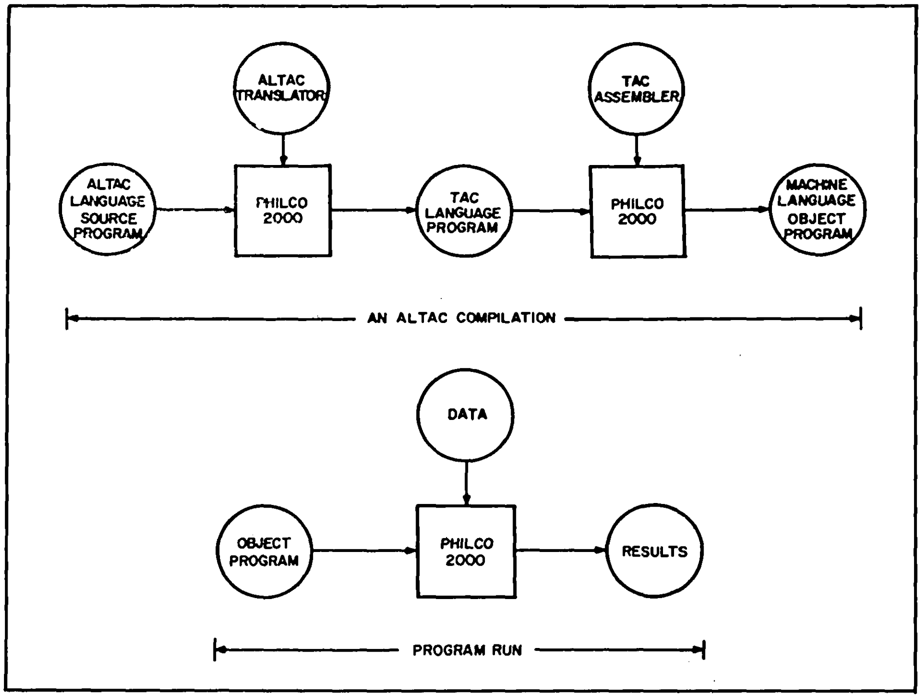
THE ALTAC LANGUAGE

The ALTAC Language is a scientific, problem-oriented, automatic programming language which may be used to express and solve many different kinds of problems. ALTAC is especially suited for solving scientific and technical problems, which usually contain a large number of algebraic expressions.

THE ALTAC TRANSLATOR

The ALTAC language program defining the operations to be performed by the computer is called the *source* program. In an ALTAC compilation, the ALTAC Translator accepts an ALTAC language *source* program and translates it into a TAC language program. The TAC Assembler then produces a machine language *object* program from the TAC language program. This *object* program may then be used in a program run, to process data and derive meaningful results.

The following diagram* shows the relation between Language and Translator:



* The diagram is not intended to show all inputs and outputs associated with the compilation and running of ALTAC programs.

The operations performed in the compilation process are continuous and require no operator intervention once the process has started.

ALTAC provides several important features not generally included in other algebraic compilers. Some of these features are:

- Programs in FORTRAN format are acceptable without modification.

- TAC language instructions may be included in the ALTAC *source* program.

- One-, two-, three-, and four-dimensional arrays can be represented.

- Any floating-point number in the range -10^{600} to 10^{600} can be accommodated.

- Symbolic addresses as well as statement numbers may be used.

- Statements may be written in compound form.

- Positive, negative, and zero subscripts are permitted. A subscript may be any fixed-point expression, including other subscripted variables.

- Mixed expressions (those containing both fixed- and floating-point values) are permitted.

Chapter I
FORMAT OF THE SOURCE PROGRAM

SOURCE PROGRAM FORMATS

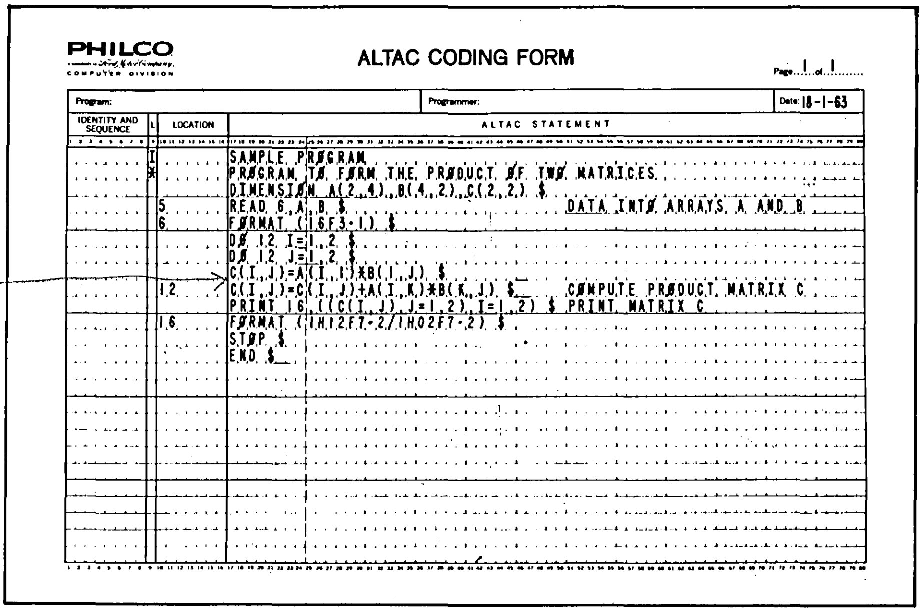
An ALTAC source program consists of a series of ALTAC language statements written in ALTAC format or FORTRAN format, or TAC instructions written in TAC instruction format.* The principal difference between ALTAC and FORTRAN formats is the location of fields on the program cards. This difference and other details of ALTAC and FORTRAN formats are presented below.

ALTAC FORMAT

When written in ALTAC format, each statement of the source program begins a new line of the ALTAC coding form (see below). Statements too long to fit on one line are continued on succeeding lines, starting after column 16.

After the program is written it is punched on cards, each line on the coding form corresponding to one card. Figure 1 shows the general appearance of an ALTAC source program as it is written on a standard ALTAC coding form. Figure 2 illustrates how this program appears on cards.

DØ 12 K=2,4$



PHILCO
COMPUTER DIVISION

ALTAC CODING FORM

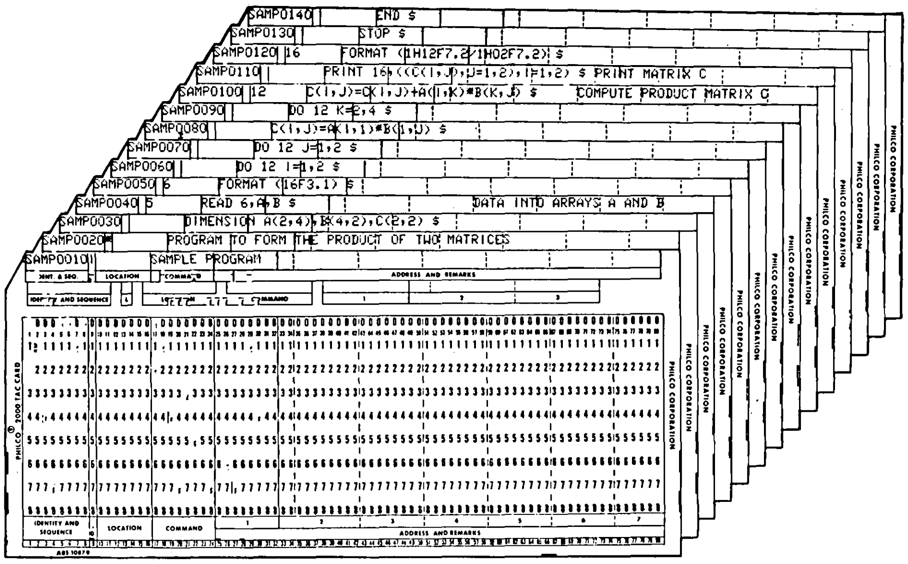
Page 1 of 1

Program: | Programmer: | Date: 18-1-63

```
IDENTITY AND  L  LOCATION  ALTAC STATEMENT
SEQUENCE
              I            SAMPLE PRØGRAM
              *            PRØGRAM TØ FØRM THE PRØDUCT ØF TWØ MATRICES
                           DIMENSIØN A(2,4),B(4,2),C(2,2) $
                 5         READ 6,A,B $                        DATA INTØ ARRAYS A AND B
                 6         FØRMAT (16F3·1) $
                           DØ 12 I=1,2 $
                           DØ 12 J=1,2 $
                           C(I,J)=A(I,1)*B(1,J) $
                 12        C(I,J)=C(I,J)+A(I,K)*B(K,J) $       CØMPUTE PRØDUCT MATRIX C
                           PRINT 16,((C(I,J),J=1,2),I=1,2) $ PRINT MATRIX C
                 16        FØRMAT (1H12F7·2/1H02F7·2) $
                           STØP $
                           END $
```

Figure 1 – An ALTAC Program

* Refer to the Philco 2000 TAC Manual, TM-11.



```
SAMP0140            END $
SAMP0130            STOP $
SAMP0120 16         FORMAT (1H12F7.2/1H02F7.2) $
SAMP0110            PRINT 16,((C(I,J),J=1,2),I=1,2) $ PRINT MATRIX C
SAMP0100 12         C(I,J)=C(I,J)+A(I,K)*B(K,J) $     COMPUTE PRODUCT MATRIX C
SAMP0090            DO 12 K=2,4 $
SAMP0080            C(I,J)=A(I,1)*B(1,J) $
SAMP0070            DO 12 J=1,2 $
SAMP0060            DO 12 I=1,2 $
SAMP0050 6          FORMAT (16F3.1) $
SAMP0040 5          READ 6,A,B $                      DATA INTO ARRAYS A AND B
SAMP0030            DIMENSION A(2,4),B(4,2),C(2,2) $
SAMP0020*           PROGRAM TO FORM THE PRODUCT OF TWO MATRICES
SAMP0010            SAMPLE PROGRAM
```

Figure 2 – An ALTAC Source Deck

The program is transferred from cards to magnetic tape before being read into the computer.

The ALTAC Coding Form

The following diagram shows the format of the ALTAC coding form and card:

IDENTITY AND SEQUENCE 1 8	L 9	LOCATION 10 16	ALTAC STATEMENT 17 80

An explanation of the contents of each field is presented below, together with the coding conventions which must be followed when writing ALTAC statements in ALTAC format.

COLUMNS	HEADING	CONTENTS
1-8	IDENTITY AND SEQUENCE	Any combination of characters to be used as identity and sequence numbers. The ALTAC Translator ignores any information in these columns.
9	L	A space, an asterisk (*), a T, or an I. A space signifies an ALTAC language card; an asterisk signifies a remarks card; a T signifies a TAC insert card; an I signifies the card containing the program identity. (See Chapter VIII.)
10-16	LOCATION	Statement numbers or symbolic addresses. A statement number may be any unsigned integer from 0 to 99999. A symbolic address may be any alphanumeric symbol from one to seven characters long, the first character of which must be alphabetic.
17-80	ALTAC STATEMENT	ALTAC language statements. Because remarks are permitted following an ALTAC statement, a dollar sign must be used to terminate the statement. The remarks are written on the same line as the statement, and may start anywhere after the dollar sign. ALTAC statements may be compounded and continued on succeeding cards (see page 14). Spaces appearing in an ALTAC statement are ignored.[†]

[†] Except spaces in Hollerith fields, and in columns 17-32 of the I Card. See pages 41, 54, and 59.

FORTRAN FORMAT

FORTRAN programs, or ALTAC programs written in FORTRAN format, are acceptable to the ALTAC Translator. The format is communicated to the ALTAC Translator by means of an IDENTIFY statement (see page 57).

Figure 3 is an illustration of the standard FORTRAN card format. The contents of each column is discussed below:

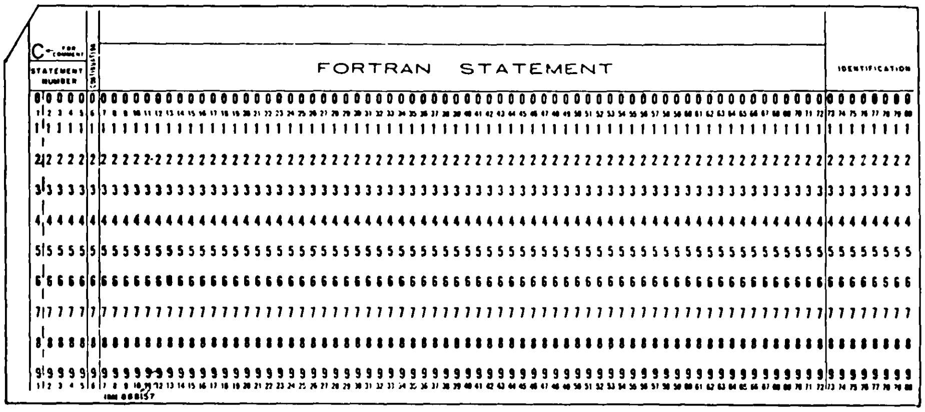


Figure 3 – Standard FORTRAN Card

COLUMNS	HEADING	CONTENTS
1	C	A "C", a "T", or a space. A "C" indicates a comments or remarks card; a "T" indicates a TAC insert; a space indicates a FORTRAN statement.
1-5	STATEMENT NUMBER	Same as for columns 10-16 of an ALTAC card, except that the first character of a symbolic address cannot be a "C" or a "T" in column 1. Because remarks are not permitted in FORTRAN statements the dollar sign is not needed to terminate the statement.
6	CONTINUATION	For a single statement: Blank For a continuation card: any non-blank character.
7-72	FORTRAN STATEMENT	FORTRAN statements
73-80	IDENTIFICATION	Identity and sequence numbers.

ALTAC CHARACTERS

The characters that are allowable in ALTAC statements[†] are:

- All decimal digits.
- All alphabetic characters.
- The twelve special characters + - * / ( ) , . ; = space (denoted by Δ) and $.

[†] Additional characters shown in Appendix A are also allowable, provided they appear as a Hollerith field in a FORMAT statement (see page 41).

Chapter II

BASIC ELEMENTS OF THE ALTAC LANGUAGE: CONSTANTS, VARIABLES, SUBSCRIPTS, AND EXPRESSIONS

In the ALTAC Language there are provisions for expressing constants, subscripted and non-subscripted variables, Hollerith fields, and arrays of up to four dimensions. When linked together with certain ALTAC "operators" (see page 10) these elements form expressions meaningful to the ALTAC Translator.

CONSTANTS

ALTAC constants are of two types - fixed point and floating point. These are defined as follows.

Fixed-Point Constants

Fixed-point constants are constants which are written in the following general form:

GENERAL FORM	EXAMPLES
Any decimal integer in the range −32767 to +32767.	7 +3895 −50

If the absolute value of a constant is greater than 32767, it is treated as a floating-point constant (see next page).

When used as a subscript, a fixed-point constant is treated modulo the size of core storage (the number of memory locations) in the object machine.

Unsigned fixed-point constants are regarded as positive. The fixed-point constants +0 and -0 are the same in the object program.

Floating-Point Constants

Floating-point constants are constants which are written in the following general form:

GENERAL FORM	EXAMPLES
Any decimal number whose absolute value is greater than 32767, or which is written either with a decimal point or with a decimal exponent preceded by an *E*. (The letter *E* means "times 10 to the power".) The *magnitude* of the number thus expressed must either be zero, or must lie in the range 10^{-600} to 10^{600}.	+1. 3.14 or .314E1 –.0062 or –6.2E–3 +101E5 98765

Note that the floating-point constants +0. and -0. are the same in the object program, while the fixed-point constant 0 and the floating-point constant 0. are *not* the same in the object program.

VARIABLES

The name of a variable may consist of from one to seven alphanumeric characters. The first character, which must be alphabetic, determines the mode (fixed- or floating-point) of the variable.

Fixed-Point Variables

Fixed-point variables are variables whose names are written in the following general form:

GENERAL FORM	EXAMPLES
Name contains 1-7 alphanumeric characters, the first of which is either *I*, *J*, *K*, *L*, *M*, or *N*.	I JOB38 KAPPA NUMBER5

Fixed-point variables can assume any integral value from -32768 to 32767 (except -0, since, as is the case with fixed-point constants, -0 and +0 are the same in the object program). If the value of a fixed-point variable lies outside this range, the value is reduced modulo 32768, or modulo the size of core storage of the object machine when used as a subscript.

Floating-Point Variables

Floating-point variables are variables whose names are written in the following general form:

GENERAL FORM	EXAMPLES
Name contains 1-7 alphanumeric characters, the first of which is alphabetic but not *I*, *J*, *K*, *L*, *M*, or *N*.	ALPHA E RHO7

Floating-point variables can assume the value 0., or any value not exceeding 10^{600} nor less than 10^{-600} *in magnitude*. (An assumed value of -0. is the same as +0. in the object program.)

SUBSCRIPTS

A subscript may be any fixed-point expression (see page 10). By subscripting a variable, it can be made to refer to any element of a one-, two-, three-, or four-dimensional array. The number of subscripts must always agree with the number of dimensions of the array.

Subscripted Variables

GENERAL FORM	EXAMPLES

A fixed-or floating-point variable, followed by parentheses enclosing 1, 2, 3, or 4 subscripts separated by commas.	C(I)
	ALPHA(I, J)
	BETA(I, J, K, L)
	GAMMA(2*I+3, J, K+5)

For each variable that appears in subscripted form, the size of the corresponding array (i.e., the subscripts of its last element) must be stated in a DIMENSION or TABLEDEF statement (see pages 25 and 28) preceding the first appearance of the variable.

Subscripted variables may appear in subscripts to any desired depth. For example, the subscripted variable

MATRIX(J(I),K)

will be read as

$$\mathrm{MATRIX}_{j_i,k} \ .$$

The variable J is also the name of an array and must appear in a DIMENSION or TABLEDEF statement. ALTAC will use the value of the i^{th} element of array J as the first subscript of MATRIX.

A general method for computing the effective address of a subscripted variable (i.e., the actual memory address represented by the variable) is presented on page 26.

Storage of Arrays

Arrays are stored forward in memory, in order of increasing absolute location, with the innermost subscript varying most rapidly. Thus, a two-dimensional array may be said to be stored "column-wise." For example, the elements of the 2x3 array

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

would be stored in the order a_{11}, a_{21}, a_{12}, a_{22}, a_{13}, a_{23}.

For three-dimensional arrays, elements of the first plane are stored before elements of the second plane, etc. This same method of storage is extended to four-dimensional arrays.

EXPRESSIONS

An expression is any sequence of constants, variables (subscripted or non-subscripted), and functions, separated by operation symbols and parentheses, so as to form a meaningful, unambiguous mathematical expression.

Operation Symbols

There are five "operators" or operation symbols in ALTAC Language. These are:

+	denoting *addition*	(binding strength 1)
-	denoting *subtraction*	(binding strength 1)
*	denoting *multiplication*	(binding strength 2)
/	denoting *division*	(binding strength 2)
**	denoting *exponentiation*	(binding strength 3)

Any of the above operators may be used in an expression, to define relationships between constants, variables, and functions. The effect of each of their binding strengths is discussed below.

Processing of Expressions

The efficiency of the instructions compiled from ALTAC expressions depends to some extent on the way the expressions are written. ALTAC processes an expression according to the following rules:

Rule 1 - In an expression of the form $A\,op_1\,B\,op_2\,C$, if the binding strengths of the operators op_1 and op_2 differ, the operations with the greater binding strength will be applied first. If the binding strengths of op_1 and op_2 are the same, then the operations in general will be performed from left to right. For example, the expression A*B**C will be computed as A*(B**C), and the expression A/B*C will be computed as (A/B)*C.

Rule 2 - ALTAC assumes that the entire expression is parenthesized. ALTAC scans from left to right until it encounters a right parenthesis; it then proceeds to evaluate the expression between this right parenthesis and its corresponding left parenthesis according to rule 1. After replacing the parenthetic expression by its value, ALTAC continues scanning (from left to right) until it encounters another right parenthesis, and proceeds as above, until all parenthetic expressions are evaluated.

In an ALTAC expression there must be a corresponding right parenthesis for each left parenthesis used, and vice versa. If this condition is not met, the statement is illegal.

An expression of the form A^{2^4} should be written as A** (2**4) or as (A**2)**4 depending on whether $A^{(2^4)}$ or $(A^2)^4$ is meant. A**2**4 is ambiguous and is therefore not a valid ALTAC expression.

No two operators are written consecutively. Negative exponents and fractional exponents of the form (x/y) should always be enclosed in parentheses, since exponentiation has the greatest binding strength.

Mixed Expressions

A special feature of the ALTAC system is that it permits the writing of mixed expressions. A mixed expression is one containing a combination of fixed- and floating-point variables or constants.

In mixed expressions the floating-point mode has precedence in specifying the mode of the value of the expression. For example, the expressions A**I and I**B would be converted to floating-point mode because of the floating-point variables A and B. *Values resulting from fixed-point operations are always truncated to an integer.*

Chapter III
ARITHMETIC STATEMENTS

The ALTAC Language comprises five types of statements:

- Arithmetic Statements
- Control Statements
- Specification Statements
- Input-Output Statements
- Subprogram Statements

Each type of statement performs a specific function. Arithmetic statements are discussed in this chapter; subsequent chapters are devoted to the discussion of the other statements.

THE ARITHMETIC STATEMENT

Arithmetic statements are written as equations. The equal sign signifies that the value of the variable on the left side is *to be replaced by* the value of the expression on the right side, not that the variable equals the expression. (See, in particular, the third example below.) The general form of an arithmetic statement is:

GENERAL FORM	EXAMPLES

$v = e$
where v is a variable (subscripted or non-subscripted), and e is an expression.
	Y = X
	A(I) = B(I)-C(I)
	SUM = SUM + X(I)
	KAPPA = A*(S-3.)/L

The value of the expression *(e)* is always converted to the mode of the variable *(v)*. Thus, in the last example above, the floating-point value of the mixed expression would be truncated to the integer (and reduced modulo 32768 if necessary) before being stored in the memory location represented by KAPPA. For example, if the value of the expression is 7.998, the value 7 will be stored, not 8.

The following are other examples of arithmetic statements:

STATEMENT	MEANING
Z=A+B	Add the quantity in A to the quantity in B and store the result in Z.
X=KAPPA	Convert the quantity in location KAPPA to floating-point and store the result in X.
Y=Y+X(I)	Add the quantity in the *i*th location of array X to the quantity in Y, and store the result in Y.
N(I)=BETA*7	Multiply the quantity in BETA by floating-point 7, convert the product to fixed-point and store the result in the *i*th location of array N.

COMPOUND STATEMENTS

A series of arithmetic statements may be compounded (written sequentially) by linking them with semicolons to form one or more consecutive lines of coding. Statements continued on succeeding lines must start after column 16. The last statement of the series must be terminated by a dollar sign.

The following is an example of a compound statement:

LOCATION	ALTAC STATEMENT
	BETA=3*Y; A=K-N/7; C=A+B $

Each statement is executed in the order in which it occurs in the program. Other examples of compound statements are presented on page 18.

If a statement number ~~of~~ or symbolic address is used with a compound statement, only the first statement in the compound statement will be identified by the statement number or symbolic address.

Chapter IV
CONTROL STATEMENTS

This chapter discusses the sixteen ALTAC statements which *control* the sequence of operations in a program. In general, these control statements may be used to:

- provide unconditional transfer of control to other statements in the program
- test variables and provide conditional transfer of control to other statements in the program
- set or test "program switches" to determine which of several paths a program may take
- execute a particular sequence of statements repeatedly a specified number of times

UNCONDITIONAL GO TO

The Unconditional GO TO statement is used to unconditionally transfer control to other statements in the program. The general form of this statement is:

GENERAL FORM	EXAMPLES
GO TO n where n is either a statement number or a symbolic address.	GO TO 9 GO TO ALPHA

This statement causes control to be transferred to the statement with symbolic address or statement number n.

ASSIGNED GO TO

A GO TO statement that is subject to modification by an ASSIGN statement (see next page) is called an Assigned GO TO. Assigned GO TO statements also provide unconditional transfer of control, and they may be written in either of two forms:

GENERAL FORMS	EXAMPLES
GO TO m or **GO TO** $m, (n_1, n_2, \ldots, n_k)$ where m is a non-subscripted variable appearing in a previously executed ASSIGN statement, and $n_1, n_2, \ldots, n_k$ are each either a statement number or a symbolic address.	GO TO Z GO TO Z, (7, K, 15)

The Assigned GO TO statement causes control to be transferred to the statement whose symbolic address or statement number is equal to the value last assigned to *m* by an ASSIGN statement. When the second form above is used, the variable *m* should not be assigned a value which does not appear in the parenthesized part of the statement.

ASSIGN

The ASSIGN statement is used to assign a value to a non-subscripted variable which appears in an Assigned GO TO statement. The general form of the ASSIGN statement is:

GENERAL FORMS	EXAMPLES
ASSIGN *n* to *m* or ASSIGN (*n*) to *m* where *n* represents a statement number, or a symbolic address if enclosed in parentheses, and *m* is a non-subscripted variable.	ASSIGN 7 to Z ASSIGN (K) to Z

When used with a subsequent GO TO statement, the ASSIGN statement causes the GO TO to transfer control to the statement whose symbolic address or statement number is *n*.

The statement ASSIGN *n* to *m* is not the same as the arithmetic statement $m = n$. A variable which has been assigned can be used only for an Assigned GO TO, until it is re-established as an ordinary variable.

COMPUTED GO TO

The Computed GO TO statement is used to transfer control to one of several statements in the program. The general form of this statement is:

GENERAL FORM	EXAMPLES
GO TO $(n_1, n_2, \ldots, n_m), i$ where $n_1, n_2, \ldots, n_m$ are each either a statement number or a symbolic address, and *i* is a non-subscripted fixed-point variable.	GO TO (10, 15, 20), J GO TO (BETA, 6, DEL), K

This statement functions as a program switch. It causes control to be transferred to the statement with symbolic address or statement number $n_1, n_2, \ldots$, or n_m , depending on whether the value of i at the time of execution of the statement is 1,2,...,or m, respectively.

Thus, if J in the first example above has the value 2 at the time of execution of the statement, control will be transferred to the statement with statement number 15.

IF

IF statements provide a means of making comparisons and conditionally transferring control. IF statements may be written in either of two forms:

GENERAL FORMS	EXAMPLES

IF $(e)n_1, n_2, n_3$
or
IF $(e_1) : (e_2), S$
where e, e_1 and e_2 represent expressions, : represents a comparison symbol, S represents a statement, and n_1, n_2, and n_3 are each either a statement number or a symbolic address.
	IF (X-Y)3, K, 6
	IF (X)GT(Y), GO TO 6

Any of the following comparison symbols may be used in an IF statement of the second form above:

SYMBOL	MEANING
E	Equal to
NE	Not equal to
LT	Less than
LTE	Less than or equal to
GT	Greater than
GTE	Greater than or equal to

In the first form above, control would be transferred to the statement with symbolic address or statement number n_1, n_2, or n_3, if the value of the expression denoted by e is less than, equal to, or greater than zero, respectively.

In the second form, control would be transferred to the statement represented by S, if the relationship (denoted by the comparison symbol) between the expressions e_1 and e_2 is met. If the relationship is not met, the next statement is executed.

If several statements are to be executed as a result of satisfying a single IF condition, the dependent statement *S* (in the second form above) may be written as a compound statement terminated by another IF statement (see Compound IF statements below), or by a dollar sign. For example, if the condition (X) E (Y) is satisfied in the following:

```
IF(X) E (Y),I=J+6;Z=A+BETA-2;GO TO KAPPA $
```

all three statements (making up the dependent statement *S*) following the IF condition would be executed. If the condition (X)E(Y) is not satisfied, all three statements will be ignored, and control will be transferred to the statement following the IF statement.

The expressions e_1 and e_2 in the second form need not be in the same mode; however, more efficient coding will result if they are.

COMPOUND IF STATEMENTS

A compound IF statement is composed of several IF statements separated by semi-colons. The following is an example of a compound IF statement:

```
IF(X) E (Y),I=J+1;IF(X)GT(Y),I=J-1;GO TO K $
```

The object program tests the conditions in sequence until it finds one condition that is satisfied. The dependent statement(s) following this satisfied IF condition are then executed. The remainder of the compound statement is ignored. The program then proceeds to the first statement which follows the compound IF statement.

SENSE LIGHT

The SENSE LIGHT statement is used to set a particular bit of a word in memory to 1, or all bits to zero, simulating an on or off condition respectively. The lights or bits are numbered from 1 to 48, and are referred to in the following manner:

GENERAL FORM	EXAMPLES
SENSE LIGHT *i* where *i* is any unsigned integer 0–48.	SENSE LIGHT 40

If *i* has the value zero, all lights are turned off. If *i* has any other value 1-48, then sense light *i* is turned on.

IF SENSE LIGHT

The IF SENSE LIGHT statement is used to test a sense light (set by a previous SENSE LIGHT statement) and to turn it off. The general form of this statement is:

GENERAL FORM	EXAMPLES

IF (SENSE LIGHT i) n_1, n_2
where i is any unsigned integer 1–48, and n_1 and n_2 are each either a statement number or a symbolic address.
	IF (SENSE LIGHT 7)5, 10
	IF (SENSE LIGHT 40)K, 7

This statement causes control to be transferred to the statement with symbolic address or statement number n_1 or n_2, if sense light i is on or off, respectively. If the sense light is on, it is turned off prior to transfer of control.

IF SENSE SWITCH

The IF SENSE SWITCH statement is used to test a sense switch. A sense switch is one of the forty-eight toggles numbered 0-47 on the Philco 2000 Console. (Reference to sense switch 48 is interpreted as a reference to sense switch 0.) This statement is written as follows:

GENERAL FORM	EXAMPLES

IF (SENSE SWITCH i) n_1, n_2
where i is any unsigned integer 0–48, and n_1 and n_2 are each either a statement number or a symbolic address.
	IF (SENSE SWITCH 9)15, 30
	IF (SENSE SWITCH 37)4, BETA

The IF SENSE SWITCH statement causes control to be transferred to the statement with symbolic address or statement number n_1 or n_2, if sense switch i is on or off, respectively.

IF SENSE BIT

The IF SENSE BIT statement is used to test a sense bit. A sense bit is one of forty-eight bits in a memory location* within an operating system. The general form of this statement is:

GENERAL FORM	EXAMPLES

IF (SENSE BIT i) n_1, n_2
where i is an unsigned integer 0–48, and n_1 and n_2 are each either a statement number or a symbolic address.
	IF (SENSE BIT 24)9, 12
	IF (SENSE BIT 40)B, KAPPA

* In the Philco Operating System SYS, the address of the memory location is 49.

This statement causes control to be transferred to the statement with symbolic address or statement number n_1 or n_2, if bit i of of the memory location is 1 or 0, respectively. Reference to sense bit 48 is interpreted as reference to sense bit 0.

IF OVERFLOW

The IF OVERFLOW statement is used to test an overflow indicator for floating-point *exponent* fault. The general form of this statement is:

GENERAL FORM	EXAMPLES

IF OVERFLOW n_1, n_2
where n_1 and n_2 are each either a statement number or a symbolic address.
	IF OVERFLOW 6, 25
	IF OVERFLOW D, LAMBDA

If floating-point exponent fault occurred, the overflow indicator is cleared to zero and control is transferred to the statement with symbolic address or statement number n_1. If overflow did not occur, control is transferred to the statement with symbolic address or statement number n_2.

The FORTRAN statements IF ACCUMULATOR OVERFLOW, IF QUOTIENT OVERFLOW, and IF DIVIDE CHECK are all treated as IF OVERFLOW statements by ALTAC.

DO

The DO statement is used to execute a series of instructions repeatedly a specified number of times. This statement may be written in either of two forms:

GENERAL FORMS	EXAMPLES
DO $n\ i = m_1, m_2, m_3$ or **DO** $(n)\ i = m_1, m_2, m_3$ where n is a statement number, or a symbolic address if enclosed in parentheses; i is a non-subscripted fixed-point variable; and m_1, m_2, and m_3 are each either an unsigned fixed-point constant or a non-subscripted fixed-point variable.	DO 7 I = 1, 9, 2 DO (K) J = 1, 16, 3 DO 5 K = 1, N DO 17 K = K ?

The DO statement causes repeated execution of all statements within its range. The *range* of a DO statement extends from the first statement following the DO statement up to and including the statement whose symbolic address or statement number is n.

The statements in the range are executed repeatedly, first for $i=m_1$, and each succeeding time for i incremented by m_3, until the value of i exceeds m_2. When the value of i exceeds m_2, the DO is said to be satisfied, and control is transferred to the first statement following the statement with symbolic address or statement number n.

The fixed-point variable i is called the *index* of the DO. m_1 represents the initial value of i , m_2 the limiting value, and m_3 the incrementing quantity. If m_3 is omitted, it is assumed to be 1; if m_2 is omitted, it is assumed to be m_1.

In the special case where both m_1 and m_2 are i, the DO is automatically satisfied at the end of its range and the value of i remains as m_1.

The following is an example of a DO statement:

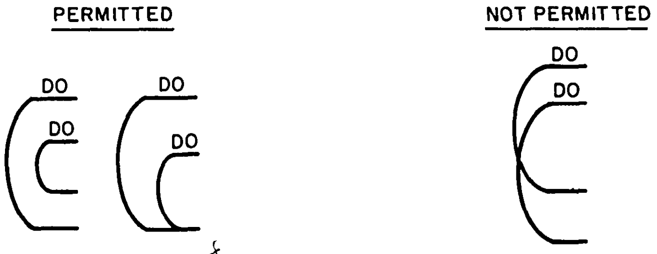
STATEMENT	MEANING
DO(ALPHA) I=1,5,2	Execute all statements immediately following, up to and including the statement with symbolic address ALPHA, first for I=1, next for I=3, and last for I=5; then transfer control to the statement following the statement whose symbolic address is ALPHA.

Statements in the range of a DO may themselves use the current value of the index, but are not permitted to redefine this value. This restriction, therefore, automatically excludes a DO in the range of another DO with the same index name.

A GO TO statement or an IF statement of the form IF *(e)* n_1,n_2,n_3 should not be the last statement in the range of a DO.

DO Nests. The range of a DO may include other DO statements, provided that the DO's are properly nested. A set of DO's is considered to be properly nested if the following rule is observed:

- If a DO statement is in the range of another DO, all statements in the range of the former DO must also be in the range of the latter. The following is an illustration of proper and improper nesting arrangements.

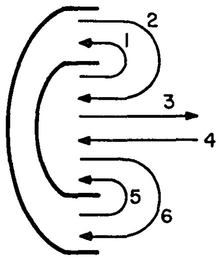


As many as 63 levels ~~or~~ of DO's are permitted in a nest.

Control cannot be transferred into the range of a DO from outside its range. The only exception to this rule is if control is being *returned* into the range of the DO, ~~and no~~ DO's or other ~~statements were executed outside its range which changed the value of the index or~~ any ~~of the indexing parameters of the DO. This exception makes~~ it ~~possible to exit temporarily from the range of a DO to execute a subroutine, if desired~~.

Control cannot be returned into the range of a DO if (1) a DO statement was executed outside of this range, or (2) a statement was executed outside of this range which changed the value of the index or an indexing parameter of the DO.

The following shows which transfers of control are permissible and which are not.



Transfers 1, 3, and 6 are permitted. Transfer 4 is permitted only if it adheres to the provisions stated in the exception above. Transfers 2 and 5 are *not* permitted.

If a DO has been satisfied and control transferred out of its range, the value of the index controlled by the DO is no longer defined, and must be redefined prior to its use again. If exit is made before the DO is satisfied, the current value of the index remains available for use.

In nested DO loops, the index value of one DO may be used by the other DO's as indexing parameters, or as subscripts or operands in other statements. For example:

```
|        .
|        .
| DO 15 J=1,N $
| DO 13 K=J,40 $
| SUM=A(J)-J $
|        .
|        .
```

CONTINUE

The CONTINUE statement is used primarily as the last statement in the range of a DO, and serves as a common point to which control is transferred. It generates no coding, other than the assignment of a statement number or symbolic address for purposes of modifying and testing the index. The general form of this statement is:

GENERAL FORM	EXAMPLE
CONTINUE	CONTINUE

At the end of the range of a DO, CONTINUE simply means "do nothing, but proceed to modify and test the index."

If the first executable statement in an ALTAC program is a TAC insert (see page 57), then a CONTINUE statement must precede the TAC insert (including the STARTTAC statement, if any).

PAUSE

The PAUSE statement is used to provide a temporary halt in a program. The general form of this statement is:

GENERAL FORM	EXAMPLES
PAUSE *n* where *n* is any unsigned fixed-point octal number of up to 5 digits. If *n* is omitted, zero is assumed.	PAUSE PAUSE 111 PAUSE 77777

Upon executing a PAUSE statement, the computer will halt displaying the octal number *n*. Pressing ADVANCE on the console will cause the program to resume operation, starting at the next sequential statement.

STOP

The STOP statement is used to signal the end of a program run. The general form of this statement is:

GENERAL FORM	EXAMPLE
STOP	STOP

When this statement is executed, all tapes will be run out (see page 36), and control transferred to the operating system used.

Chapter V
SPECIFICATION STATEMENTS

The ALTAC Language includes four Specification Statements: DIMENSION, EQUIVALENCE, COMMON, and TABLEDEF. These are non-executable statements; they provide the ALTAC Translator with information concerning the allocation of storage, and the arrangement of data in memory. The function of each of these statements is discussed below.

DIMENSION

The DIMENSION statement provides ALTAC with the information necessary to allocate storage for an array in the source program. The name of each array together with its dimensions *must* appear in a DIMENSION (or TABLEDEF) statement. The general form of the DIMENSION statement is:

GENERAL FORM	EXAMPLES
DIMENSION $v_1, v_2, v_3, \ldots$ where each v is a variable, subscripted with 1, 2, 3, or 4 unsigned fixed-point constants, representing the maximum dimensions (last element) of the corresponding array.	DIMENSION A(5), B(4, 7) DIMENSION KAPPA(3, 5, 7), RHO(2, 4, 6, 8)

In the second example above, KAPPA is shown to be the name of a 3-dimensional array for which 105 (3x5x7) locations are reserved; RHO is the name of a four dimensional array for which 384 (2x4x6x8) locations are reserved.

If the name of an array appearing in a DIMENSION (or TABLEDEF) statement also appears in an EQUIVALENCE and/or COMMON statement, the EQUIVALENCE statement must precede the COMMON, DIMENSION or TABLEDEF statements; the COMMON statement, in turn, must precede the DIMENSION or TABLEDEF statement.

There may be several DIMENSION statements in a program, each of which must precede the first appearance* of any of its variables in the program.

* Not considering appearances in EQUIVALENCE and COMMON statements.

A DIMENSION statement should not contain the names of functions or subroutines.

EQUIVALENCE

The EQUIVALENCE statement permits the programmer to conserve storage by specifying that storage locations are to be shared by two or more variables. EQUIVALENCE is also used in conjunction with the COMMON statement, to control the allocation of storage in the common storage area (see page 27). The general form of the EQUIVALENCE statement is:

GENERAL FORM	EXAMPLE
EQUIVALENCE $(v_1, v_2, v_3, \ldots)$, $(v_k, v_{k+1}, \ldots), \ldots$ where each v represents a variable. A single unsigned fixed-point constant in parentheses may follow a variable.	EQUIVALENCE (A, B(5), C), (BETA(10), RHO, X(2))

In the example shown above, arrays A, B, and C are to be assigned storage locations in such a way that the first element of array A, the fifth element of array B, and the first element of array C all occupy the same location. Similarly, the tenth element of array BETA occupies the same memory location as the first element of array RHO, and the second element of array X. A programmer can thus refer to the same memory location by different names. It is his responsibility, however, to insure that the appropriate values appear in these locations at the time of reference.

Computing Effective Addresses

It may be necessary for a programmer to know the effective address of a subscripted variable, for example, when planning to use the EQUIVALENCE statement. By means of the following information, he can calculate this address:

Assuming a subscripted variable of the form $A(J_1,J_2,J_3,J_4)$, with corresponding DIMENSION statement

$$\text{DIMENSION } A(N_1,N_2,N_3,N_4)$$

the general equation for computing the effective address is:

$$\text{The address of } A(J_1,J_2,J_3,J_4) = A+(J_1-1)+N_1(J_2-1)+N_1N_2(J_3-1)+N_1N_2N_3(J_4-1)$$

For a variable of less than four dimensions, substitute 1 for the unused subscripts in the general equation above. Thus, if A(2,1,2) is a subscripted variable with dimensions A(3,3,3), the effective address of A(2,1,2) = A+(2-1)+3(1-1)+9(2-1) = A+ 10, the eleventh element of array A.

COMMON

The COMMON statement is used to reserve areas of common storage which are equally accessible to different object programs in memory. The general form of this statement is:

GENERAL FORM	EXAMPLE
COMMON $v_1, v_2, v_3 \ldots$ where each v is the name of a variable or is a non-subscripted array name.	COMMON ZETA, B, TAU

The variables are placed in common storage in the order that they appear in the COMMON statement, provided none of them appear in an EQUIVALENCE statement. Variables which appear in both EQUIVALENCE and COMMON statements will be placed first in the common area, in the order that they appear in the EQUIVALENCE statement. For example, according to the statements:

```
EQUIVALENCE (D,H),(A,F)
COMMON A,B,C,D,E
DIMENSION B(3),C(2),E(2)
```

common storage would be assigned as follows:

D and H

A and F

B (1)

B (2)

B (3)

C (1)

C (2)

E (1)

E (2)

The size of the EQUIVALENCE storage within COMMON *plus* the total size of those variables which appear in COMMON and do *not* appear in EQUIVALENCE, is the size of the area of memory reserved for common storage.

TABLEDEF

The TABLEDEF statement is used to specify the dimensions of an array which has been defined by means of a TAC insert (see page 57). The general form of this statement is:

GENERAL FORM	EXAMPLE
TABLEDEF $v_1, v_2, v_3, \ldots$ where each v is a variable, subscripted with 1, 2, 3, or 4 unsigned fixed-point constants, representing the maximum dimensions (last element) of the corresponding array.	TABLEDEF A(10), B(5, 8), DELTA(3, 4, 5)

ALTAC does *not* reserve storage for an array which appears in a TABLEDEF statement, unless the array also appears in an EQUIVALENCE or COMMON statement, or appears as a formal parameter of a subprogram (see page 46).

Chapter VI
INPUT-OUTPUT STATEMENTS

There are two kinds of input-output statements in ALTAC: ORDER statements and FORMAT statements. These statements are used together to specify the transmission of information between core and magnetic tapes.

ORDER statements specify either an input or an output operation to be performed, or the manipulation of magnetic tapes. These statements may contain a tape reference, a FORMAT statement reference, and a list of the quantities to be transmitted.

FORMAT statements provide information about the form and arrangement of data, and the type of data conversion to be performed.

ORDER STATEMENTS

There are thirteen ORDER statements in ALTAC. These may be grouped as follows:

- Six statements which provide for the transfer of *binary coded* (6 bits per character) information:

  READ *n, List*
  PRINT *n, List*
  PUNCH *n, List*
  READ INPUT TAPE *i, n, List*
  WRITE OUTPUT TAPE *i, n, List*
  PUNCH OUTPUT TAPE *i, n, List*

- Two statements which provide for the transfer of *binary* information:

  READ TAPE *i, List*
  WRITE TAPE *i, List*

- Five statements which provide for the manipulation of magnetic tapes:

  END FILE *i*
  RUNOUT *i*
  BACKSPACE *i*
  REWIND *i*
  LOCKOUT *i*

The parameter *i* represents a magnetic tape unit, *n* represents the statement number or symbolic address of a FORMAT statement, and *List* represents a list of the variables and arrays that are to be transferred, and the order of transfer.

Magnetic Tape References

The parameter *i*, representing a magnetic tape unit, is a fixed-point variable or an unsigned fixed-point constant. If *i* is a fixed-point variable, it must be defined prior to its use. If *i* is a fixed-point constant, the following rules apply:

- If *i* consists of two digits or less, the digits will be interpreted as the tape number.
- If *i* consists of more than two digits, the last two digits will be interpreted as a data select* character, while the preceding digits will be interpreted as the tape number.

Both the data select character and the tape number are treated modulo 16.

Format Statement References

The parameter *n* is the statement number or symbolic address of the FORMAT statement that is associated with the ORDER statement. In a READ, PRINT, or PUNCH statement, a symbolic address *n* must always be enclosed in parentheses.

Lists

ORDER statements which call for the transfer of information ordinarily contain a *List* of the quantities to be transmitted. (Cases where the *List* parameter is omitted are discussed on page 31.) A *List* refers to specific locations in memory, and is represented by a series of subscripted or non-subscripted variables separated by commas. The following are some examples of *Lists:*

```
A
A,B
A,B,(C(I), I=1,10)
A,B,((C(I,J), I=3,10,2),D(J,7),J=1,5)
```

A *List* is read from left to right with repetition for variables enclosed in parentheses.

Information is transferred item by item in the order that the variables appear in the *List*. When no items remain, transmission ceases.

* Refer to the Philco 2000 Input-Output Systems Manual (TM-16).

The iterative action involved in assigning values to the elements of a parenthesized *List* is the same as that for a DO loop. For example, the order of operations for

```
A,C,(B(I), I=1,10),D
```

is the same as for the following:

```
  | A
  | C
  | DO 7 I=1,10
7 | B(I)
  | D
```

For a *List* of the form

```
A,X(5),((D(I,J), I=1,3,2),BETA(1,J),J=1,2)
```

the information would be processed as follows:

```
A
X(5)
D(1,1)
D(3,1)
BETA(1,1)
D(1,2)
D(3,2)
BETA(1,2)
```

Simplifying a List

An entire array may be transmitted by writing only the name of the array. For example, if the names ALPHA and BETA previously appear in a DIMENSION statement, the statement

READ INPUT TAPE *i*,*n*,ALPHA,BETA$

would cause the input of all the elements of arrays ALPHA and BETA, in the order in which they are stored in memory. Whenever possible the programmer should use this abbreviated notation, for its use will result in a more efficient object program.

Omission of The List Parameter

The *List* parameter is omitted in the following cases:

- To space forward over an input record.
- To write a blank record on tape. (On binary tapes, a blank record is interpreted as an end-of-file record.*)
- To read or write a record described by a FORMAT statement that contains Hollerith specifications only.

*See also the END FILE statement, page 35.

READ

The READ statement is used to read binary coded information from the system input tape*. The general form of this statement is:

GENERAL FORM	EXAMPLES
READ *n*, *List* where *n* and *List* are as described on page 30.	READ 11, A, (B(I), I = 1, 5) READ(K), (BETA(I), I = 1, 10)

This statement causes card after card of information to be read from the system input tape until the amount of information specified by the *List* is transmitted. The information that is read is converted according to the FORMAT statement whose symbolic address or statement number is *n*, and is stored in the memory locations specified in the *List*. If the FORMAT statement specifies more than 80 characters to be read from a card, an error will result and control will be transferred to the operating system.

READ INPUT TAPE

The READ INPUT TAPE statement is used to read binary coded information from a tape. The general form of this statement is:

GENERAL FORM	EXAMPLES
READ INPUT TAPE *i*, *n*, *List* where *i*, *n*, and *List* are as described on page 30.	READ INPUT TAPE 9, 30, (A(I), I = 1, 99, 2) READ INPUT TAPE 6, 25, ALPHA, BETA READ INPUT TAPE INAME, KAPPA, (A(J), J = 1, N)

This statement causes binary coded information to be read a card at a time from tape *i*. The information is converted according to the FORMAT statement whose symbolic address or statement number is *n*, and stored in the memory locations specified in the *List*.

If the FORMAT statement specifies more than 80 characters to be read from a card, an error will result and control will be transferred to the operating system.

* System input and output tapes are those input or output tapes that are defined and/or controlled by a particular operating system.

READ TAPE

The READ TAPE statement is used to read binary information from tape. The general form of this statement is:

GENERAL FORM	EXAMPLES
READ TAPE *i*, *List* where *i* and *List* are as described on page 30.	READ TAPE 10, (A(I), I = 1, 20) READ TAPE L, (ALPHA(J), J = 1, N, 2)

This statement causes binary information to be read from tape *i* into the memory locations specified in the *List*. No conversion is required; consequently there is no FORMAT reference *n* in this statement.

The binary information must have been written by a WRITE TAPE statement (see page 35), and will contain as many words as are specified in its *List*.

All or part of the record may be read by the READ TAPE statement, after which the tape is positioned at the beginning of the next record. Attempting to read more words than were written in the record will result in an error and control will be transferred to the operating system.

PRINT

The PRINT statement is used to write binary-coded information on the system output tape, edited for the off-line High Speed Printer. The general form of this statement is:

GENERAL FORM	EXAMPLES
PRINT *n*, *List* where *n* and *List* are as described on page 30.	PRINT 34, A, (B(I), I = 1, 7) PRINT(K), A, (BETA(I), I = 1, 16, 3)

This statement causes information specified in the *List* to be written on the system output tape (defined by the installation), edited for the High Speed Printer. The tape is then printed off-line.

As many as 120 characters may be printed on a line. Successive lines are printed in accordance with the FORMAT statement whose symbolic address or statement number is *n*, until the complete *List* has been satisfied. If the FORMAT statement specifies more than 120 characters to be printed on a line an error will result, and control will be transferred to the operating system.

PUNCH

The PUNCH statement is used to write binary-coded information on the system output tape, edited for the off-line punch. The general form of this statement is:

GENERAL FORM	EXAMPLES
PUNCH *n*, *List* where *n* and *List* are as described on page 30.	PUNCH 100, (ALPHA(I), I = 1, 30) PUNCH(KAPPA), A, B, (C(J), J = 1, 50)

This statement causes information specified in the *List* to be written on the system output tape, edited for the Card Punch. The tape is then punched off-line.

As many as 80 columns can be punched on a card. Successive cards are punched, according to the FORMAT statement with symbolic address or statement number *n*, until the *List* has been satisfied. If the FORMAT statement specifies more than 80 characters to be punched on a card, an error will result and control will be transferred to the operating system.

PUNCH OUTPUT TAPE

The PUNCH OUTPUT TAPE statement is used to write binary-coded information on a tape, edited for the off-line punch. The general form of this statement is:

GENERAL FORM	EXAMPLES
PUNCH OUTPUT TAPE *i*, *n*, *List* where *i*, *n*, and *List* are as described on page 30.	PUNCH OUTPUT TAPE 13, K, (BETA(I), I = 1, 40) PUNCH OUTPUT TAPE 1302, K, ALPHA, BETA PUNCH OUTPUT TAPE L, K, (GAMMA(I), I = 1, 10)

This statement causes binary coded information in the *List* to be written on tape *i*, edited for the Card Punch according to the FORMAT statement with symbolic address or statement number *n*. Tape *i* is then punched off-line.

Successive cards are punched in accordance with the FORMAT statement until the *List* is satisfied. If the FORMAT statement specifies more than 80 characters to be punched on a card, an error will result and control will be transferred to the operating system.

WRITE OUTPUT TAPE

The WRITE OUTPUT TAPE statement is used to write binary coded information on a tape edited for the off-line High Speed Printer. The general form of this statement is:

GENERAL FORM	EXAMPLES
WRITE OUTPUT TAPE *i*, *n*, *List* where *i*, *n*, and *List* are as described on page 30.	WRITE OUTPUT TAPE 12, K, (BETA(I), I = 1, 40) WRITE OUTPUT TAPE 1205, K, DELTA, GAMMA WRITE OUTPUT TAPE M, KAPPA, A, (B(I), I = 1, 30)

This statement causes binary coded information specified in the *List* to be written on magnetic tape *i*, edited for the High Speed Printer according to the FORMAT statement with symbolic address or statement ~~under~~ NUMBER *n*. The information is printed off-line. As many as 120 characters can be printed per line. Successive lines are printed in accordance with the FORMAT statement, until the *List* is satisfied. If the FORMAT statement specifies more than 120 characters to be printed on a line, an error will result and control will be transferred to the operating system.

WRITE TAPE

The WRITE TAPE statement is used to write binary information on tape. The general form of this statement is:

GENERAL FORM	EXAMPLES
WRITE TAPE *i*, *List* where *i* and *List* are as described on page 30.	WRITE TAPE 10, (A(I), I = 1, 20) WRITE TAPE L,(ALPHA(J), J=1, 25, 2)

This statement causes a record of binary information to be written on tape *i*. The record is written in a format that is acceptable to the READ TAPE statement and consists of all the words specified in the *List*.

END FILE

The END FILE statement is used to write an end-of-file mark on a tape. The general form of this statement is:

GENERAL FORM	EXAMPLES

END FILE *i*
where *i* is an unsigned fixed-point constant or fixed-point variable, as described on page 30.
	END FILE 12
	END FILE K
	END FILE 506

This statement causes an end-of-file indicator to be written on tape *i*. In the case where tape *i* was last used as an input data tape or is a system controlled tape, no end-of-file indicator is written.

RUNOUT

The RUNOUT statement may be used to position an input tape or to transmit the contents of an output buffer block.* The general form of this statement is:

GENERAL FORM	EXAMPLES
RUNOUT or **RUNOUT** *i* where *i* is an unsigned fixed-point constant or fixed-point variable, as described on page 30.	RUNOUT RUNOUT 9 RUNOUT K

The RUNOUT statement is interpreted as follows:

- If the last reference to tape *i* was made by an *input* statement, the RUNOUT *i* statement will position the tape at the end of the block which contains the last processed record.
- If the last reference to tape *i* was made by an *output* statement, the RUNOUT *i* statement will complete the editing and transmit the contents of the output buffer blocks to that tape, if necessary.

A RUNOUT statement without a tape reference is interpreted as a runout of all tapes used.

BACKSPACE

The BACKSPACE statement is used to backspace a binary tape. The general form of this statement is:

GENERAL FORM	EXAMPLES
BACKSPACE *i* where *i* is an unsigned fixed-point constant or fixed-point variable, referring to a binary tape.	BACKSPACE 10 BACKSPACE L

This statement causes binary tape *i* to be backspaced one record.

* 128 word area in memory or on magnetic tape.

REWIND

The REWIND statement is used to rewind a tape. The general form of this statement is:

GENERAL FORM	EXAMPLES
REWIND *i* where *i* is a fixed-point constant or fixed-point variable, as described on page 30.	REWIND 12 REWIND K

This statement causes tape *i* to be rewound. If the last reference to tape *i* was made by an output statement, the REWIND statement will complete the editing and transmit the contents of the output buffer blocks before the rewind occurs.

LOCKOUT

The LOCKOUT statement is used to rewind and lockout* a tape. The general form of this statement is:

GENERAL FORM	EXAMPLES
LOCKOUT *i* where *i* is a fixed-point constant or fixed-point variable, as described on page 30.	LOCKOUT 9 LOCKOUT K

This statement causes tape *i* to be rewound with lockout. If the last reference to tape *i* was made by an output statement, the LOCKOUT statement will complete the editing and transmit the contents of the output buffer blocks before the rewind with lockout occurs.

FORMAT STATEMENTS

The FORMAT statement is used to control the conversion of data to or from an internal form and an external form. FORMAT statements contain field descriptors which provide information about the external form of the data, and the type of data conversion to be performed.

FORMAT statements are of the following general form:

GENERAL FORM	EXAMPLES
FORMAT (d_1, . . ., d_n) where each *d* is a field descriptor.	FORMAT (2H10, I3, F5.2, E8.3, A8/)

* When a tape is "locked-out" it can no longer be referenced by the program, unless an operator intervenes and changes its lockout status.

Each unmodified* field descriptor describes one field. The leftmost descriptor describes the first field, the next descriptor describes the second field, and so on.

Each FORMAT statement must contain a statement reference in its location field. FORMAT statements are non-executable statements, and may therefore be placed anywhere in a program.**

FIELD DESCRIPTORS

ALTAC field descriptors comprise the *Iw, Fw.d, Ew.d, Ow,' Aw, nH* and *nX* descriptors. These descriptors may be used to describe numeric, alphanumeric, and blank fields.

Numerical Field Descriptors

Four forms of conversion of numerical data are available:

DESCRIPTOR	EXTERNAL FORM	INTERNAL FORM
Iw	Decimal Integer	Fixed-Point Binary
Fw.d	Fixed-Point Decimal	Floating-Point Binary
Ew.d	Floating-Point Decimal	Floating-Point Binary
Ow	Octal Integer	Binary representation of the octal integer

- *I, F, E* and *O* are control characters specifying the type of conversion.
- *w* is an unsigned fixed-point constant representing the width (number of characters) of the field in the external medium.
- *d* is an unsigned fixed-point integer representing the number of characters in the field which appear to the right of the decimal point.

For *F* and *E* conversions, *w* may represent an input field of as many as 80 characters, corresponding to the contents of an entire card. For *I* conversions, an input quantity should not be greater

* See Repetition of Similar Formats, page 41.

** If all FORMAT statements are placed before the first executable statement of the source program, a more efficient object program will result.

than 32767 in magnitude; if it is, it is reduced modulo 32768. For *O* input conversions, *w* should not exceed 16; if *w* exceeds 16, only the right-most 16 characters of the field are used.

An output field may contain as many as 80 characters on a card, or as many as 121* characters to a printed line. The output field always contains the right-most *w* characters of the output quantity, with leading spaces added to make up the *w* count where necessary.

A numerical field may contain decimal or octal digits, decimal point, plus and minus signs, spaces, and the letter *E* (in the case of *E* and *F* input conversions). On input, non-leading spaces are interpreted as zeros.

The character *d* represents the number of characters in the field which appear to the right of the decimal point. In the case of *E* output conversions, where the output quantities are ordinarily expressed in mantissa-exponent form (see below), *d* represents the number of fractional digits of the mantissa.

If *d* is greater than 10, it is assumed to be 10. If the decimal point is omitted from an *Fw.d* or an *Ew.d* descriptor, the descriptor assumes the *d* specified (or assumed) for the previous *F* or *E* descriptor.

A decimal point appearing in an input data field takes precedence over the *d* specification for that field.

The acceptable forms of *input* fields for the *E* conversion are:

±mantissa
±mantissa±exponent
±mantissaE±exponent
±mantissaEexponent

These forms may be written with or without a sign.

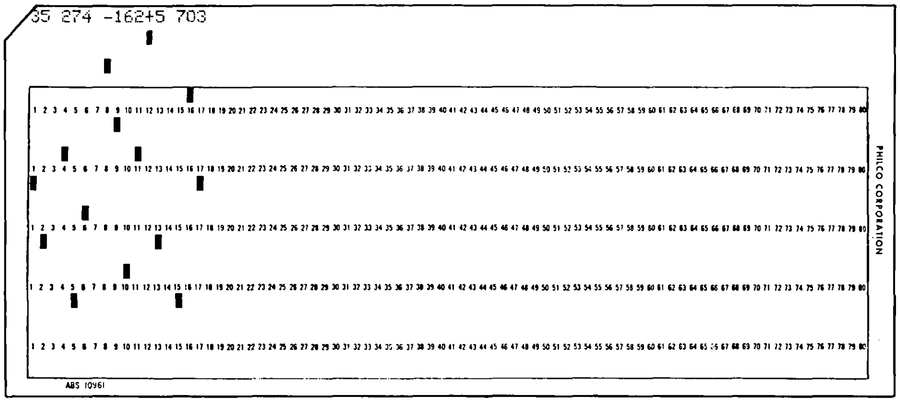
The mantissa may be of any magnitude; the exponent may be any integer in the range -600 to 600. The *output* form for the *E* conversion with no scale factor (see page 40) is:

$$\overbrace{\pm 0.\underbrace{\text{xxx}\cdots\text{xxx}}_{\text{d}}\pm\text{eee}}^{\text{w}}$$

* 120 printable characters *plus* a printer control character (see page 45).

Examples of Input Conversions

If the data punched on the following card



is described by the statement

FORMAT(I2,F4.1,E7.2,O4)$

the fields will be interpreted as 35,27.4,-1.62x10^5, and octal 703.

Examples of Output Conversions

If the internal quantities 417,-.329,+.538x10^3, and octal 627 are described by the statement:

FORMAT (1H1I3,F6.2,E10.3,O5)$

they will be represented externally as:

417Δ-0.33Δ0.538+003ΔΔ627

where Δ denotes a space, and the characters 1H1 in the FORMAT statement are printer control characters (see page 42), which cause the printer to skip to the top of the next page before printing the line.

Alphanumeric Field Descriptors

The descriptors *Aw* and *wH* are used to specify the form of alphanumeric fields. An alphanumeric field may contain any of the Philco characters shown in Appendix A.

The *Aw* descriptor may be used to describe a field of up to eight characters.

For A input: *w* should not exceed 8; if *w* exceeds 8, the rightmost 8 characters of the input field are used to fill the computer word. If *w* is less than 8, the *w* characters of the field are stored left justified with trailing blanks.

For A output: if *w* is greater than 8, the 8 characters of the output field are preceded by *w*-8 spaces. If *w* is less than 8, the leftmost *w* characters of the computer word are transmitted.

The alphanumeric field described by a *wH* descriptor, unlike that described by an *Aw* descriptor, is not limited to a single computer word. The *w* characters of the field are written following the *wH* specification in the FORMAT statement. For example,

```
41H ALPHANUMERIC=HOLLERITH=PHILCO CHARACTERS
```

Note that spaces are significant, and are included in the *w* count. *w* may be any value not exceeding the record size (i.e., 121 characters when specifying a printed line of 120 characters, or 80 characters per card).

For H input: *w* characters are extracted from the input record and they replace the *w* characters following the *wH* specification.

For H output: the *w* characters following the specification (or the characters which replace them, see *H input* above) are written as part of the output record.

Blank Field Descriptor

The descriptor *wX* may be used to skip *w* characters of an input record, or to insert *w* spaces in an output record.

If the descriptor *wX* (or *wH*) precedes another descriptor, the comma normally used to separate the two descriptors may be omitted. Only in the case of these two descriptors is this omission permitted.

REPETITION OF SIMILAR FORMATS

When successive fields within a record are to be of the same format, a single descriptor may be used to specify this common format. The number of fields affected by this single descriptor is indicated by a fixed-point constant, *n*, which is prefixed to the descriptor (*I,F,E,O,* or *A*). Thus,

```
FORMAT (F6.2,F6.2,F6.2)
```

and

```
FORMAT (3F6.2)
```

are equivalent.

If the format of a group of fields are to be repeated *n* times, the descriptors for the group may be enclosed in parentheses, preceded by the constant *n*. For example, the statement

```
FORMAT(F5.2,3XF5.2,3X)$
```

and

```
FORMAT(2(F5.2,3X))$
```

are equivalent.

SCALE FACTORS

To permit more general use of the *F* and *E* descriptors, a scale factor, *nP*, may precede the specification. *n* is a fixed-point constant, which may be negative or unsigned. (A plus sign is not a legitimate character in a FORMAT statement.) *P* is a control character.

For *F* input and output conversions, the scale factor is defined such that:

$$\text{External Number} = \text{Internal Number} \times 10^{n}$$

When *nP* is used with an *E output* descriptor, the mantissa of the output quantity is multiplied by 10^{n} and the exponent is reduced by *n*. Thus, if the quantities 536, 1624, .732x10^{5}, were described by the statement

```
FORMAT(I3,-1PF7.1,2PE10.1)$
```

the following would result:

```
536   162.4   73.2+003
```

The *E input* descriptor ignores the scale factor.

Once *nP* is specified for an *F* or *E* descriptor, it will apply to all succeeding *F* or *E* descriptors within the FORMAT statement until another *nP* is specified.

PRINTER OUTPUT CONTROL CHARACTERS

The first character of each record that is to be printed is treated as a vertical format character. Vertical format characters control the vertical spacing of the paper on the High Speed Printer, and are interpreted as follows:

CHARACTER	MEANING
1	skip to top of next page
0	double space
Δ (space)	single space
+	no space

Any other character used will be interpreted as single space.

MULTI-RECORD FORMATS

A single FORMAT statement may be used to describe several records. The descriptors of each record are separated by slashes. For example, if data are to be printed according to the statement

```
FORMAT(1H1I5,F8.2/1HΔE9.2)$
```

the first line would be printed according to descriptors I5 and F8.2, and the second line according to descriptor E9.2. If the second and all succeeding lines are to be printed according to descriptor E9.2, the specifications for these lines should be enclosed in another pair of parentheses, as follows:

```
FORMAT(1H1I5,F8.2/(1HΔE9.2))$
```

If the end of a format statement is reached before the List is satisfied, the format repeats from the last open (left) parenthesis.

Both the slash and the last right parenthesis of a FORMAT statement indicate the end of a record.

Consecutive slashes may be used in order to skip records; i.e., to skip an input card, or to produce a blank line or a blank card. *n+1* consecutive slashes causes *n* records to be skipped. For example,

/// would cause two records to be skipped.

FORMAT STATEMENT PROCESSING

FORMAT statements are translated and stored as one or more consecutive word (W/) constants* by ALTAC during compilation. The first word begins with the first left parenthesis that followed the word "FORMAT"; the last word ends with the last right parenthesis with trailing blanks if necessary. Interpretation of the FORMAT statement is made at run time.

FORMAT STATEMENTS READ IN DURING PROGRAM EXECUTION

Although FORMAT statements are usually written in the source program, they may also be read in during the execution of the object program. For example, according to the statements:

LOCATION	ALTAC STATEMENT
	DIMENSION SPEC (20), ALPHA(20)$
1	FORMAT (20A8) $
	READ 1, SPEC $
	READ (SPEC), ALPHA $

* Refer to the Philco 2000 TAC Manual, TM-11.

the alphanumeric data that is read into array SPEC by the first READ statement, is used as format specifications by the second READ statement. The format specifications (alphanumeric data) read into array SPEC must have been written as if they were appearing in a FORMAT statement in the source program, except that the word "FORMAT" is omitted (see preceding section).

Chapter VII

FUNCTIONS AND SUBROUTINE SUBPROGRAMS

A function or a subroutine is a pre-coded set of instructions for performing a particular operation.

There are three distinct types of functions in ALTAC: Arithmetic Statement Functions, Library Functions, and Function Subprograms.* There are also Subroutine Subprograms.

An Arithmetic Statement Function is a function which is defined by a single arithmetic statement in the source program. A Library Function is a function which is defined on the TAC library tape. A Function Subprogram is a function which is defined by a subprogram. A Subroutine Subprogram is a subroutine which is defined by a subprogram. Subroutine Subprograms differ from Functions in their output capacity and in the method in which they are referenced (see page 53).

Arithmetic Function Statements should precede all other statements in the source program, except IDENTIFY, FUNCTION, SUBROUTINE, or the I card.

Other details regarding Functions and Subroutine Subprograms are presented below.

FUNCTION NAMES

The name of a function may be composed of from one to seven alphanumeric characters. The first character, which must be alphabetic, determines the mode of the value of the function.

The following rules must be observed when naming functions:

Rule 1 - If the name of a function is four to seven characters long and the last character *is* an *F*, then the value of the function is in fixed-point mode only if the first character is *X*.

Rule 2 - If the name of a function is four to seven characters long and the last character *is not* an *F*, or if the name of

* A subprogram is a separately written source program designed to operate under the control of a main program. Subprograms may also call other subprograms.

the function is less than four characters long, then the value of the function is in fixed-point mode only if the first character is *I, J, K, L, M, or N.*

Rule 1 applies to Arithmetic Statement Functions and Library Functions; rule 2 applies to Function Subprograms.

SUBROUTINE NAMES

The name of a Subroutine Subprogram may be composed of from one to seven alphanumeric characters, the first character of which must be alphabetic. (Unlike function names, a subroutine name does not have any mode associated with it.)

ARGUMENTS

The arguments of a function or subroutine are written separated by commas, and enclosed in parentheses following the function or subroutine name. An argument of an Arithmetic Statement Function may be any expression. An argument of any other function or any subroutine may be an expression, the name of an array, or a Hollerith field.

The appearance of the name of a function in an expression, or the name of a subroutine in a CALL statement (see page 54), serves to call that function or subroutine. The function or subroutine is then computed using the arguments which appear after the function name in the expression, or which appear after the subroutine name in the CALL statement. The arguments which appear after the function or subroutine name in the statement defining or identifying the function or subroutine, are *formal parameters.* Each formal parameter is a single non-subscripted variable. These formal parameters are replaced by the corresponding arguments in the calling statement prior to the calculation of the function or subroutine.

The arguments of the function or subroutine in the calling statement must always agree in number, order, and mode, with the formal parameters in the statement defining or identifying the function or subroutine.

The number* of arguments following a function or subroutine name can be from 1 to 31 for an 8192 word source computer, or from 1 to 255 for a 16,384 or 32,768 word source computer.

* As shown on page 53, this number can also be zero for subroutines, since a subroutine can be without arguments.

ARITHMETIC STATEMENT FUNCTIONS

These are functions which are defined by a single arithmetic statement. The general form of this type of function is:

GENERAL FORM	EXAMPLES
$f(a_1, a_2, \ldots) = e$ where *f* is a function name that obeys rule 1 on page 45, each *a* is a formal parameter, and *e* is an expression not involving subscripted variables.	RATEF (A, B) = A/60*B XVALUEF(J, K) = J*K/N**2

The arithmetic statement defining the function must precede any statement calling the function and any EQUIVALENCE, COMMON, DIMENSION or TABLEDEF statement in the program.

The arguments which appear after a function name in the statement defining the function are formal parameters, and are replaced by the corresponding arguments in the calling statement prior to the calculation of the function. For example, according to the following statements

```
           |      .
           |      .
Defining → | RATIOF(X,Y)=X/Y
Statement  |
           |      .
           |      .
Calling  → | Z=10 * RATIOF (A+B, C**2)
Statement  |
           |      .
           |      .
```

the calling statement, *Z*, would be evaluated as if it were written

```
Z = 10*(A+B)/C**2
```

LIBRARY FUNCTIONS

Library functions are functions that are included on the TAC library tape because of their frequent use. Each installation may have its own set of ALTAC library functions. The following are some of the standard functions which are supplied with the ALTAC Translator. The appearance of the name of the function in an expression serves to call the function.

Function Name	Number of Arguments	Mode of		Operation Performed
		Arguments	Function	
ABSF	1	Floating	Floating	Computes $\mid arg \mid$
XABS F	1	Fixed	Fixed	Computes $\mid arg \mid$
FCABSF	2	Floating	Floating	Computes $\mid arg \mid$, where *arg* is a complex number
COSF	1	Floating	Floating	Computes COS(arg) in radians
COS1F	1	Floating	Floating	Same as COSF
ACOSF	1	Floating	Floating	Computes COS^{-1}arg
ACOS1F	1	Floating	Floating	Same as ACOSF
FCORF	7	One Fixed and Six Floating	Floating	Computes correlation coefficient of two variables
~~FCORMVF~~	~~4~~	~~One Fixed and Three Floating~~	~~Floating~~	~~Computes correlation coefficient, means, and variances of two varibles~~
DIMF	2	Floating	Floating	Produces a positive difference: DIMF (arg_1,arg_2) = arg_1 - MINF (arg_1,arg_2)
XDIMF	2	Fixed	Fixed	Same operation as above, using XMINF
EXPF	1	Floating	Floating	Computes the value e^{arg}
FGAMMAF	1	Floating	Floating	Computes $\Gamma(arg)$
~~FLECF~~	~~3~~	~~One Fixed and Two Floating~~	~~Floating~~	~~Solves a system of linear equations by Crout's method.~~
~~FLEJF~~	~~3~~	~~One Fixed and Two Floating~~	~~Floating~~	~~Solves a system of linear equations~~
FLOATF	1	Fixed	Floating	Converts fixed-point *arg* to floating point

Function Name	Number of Arguments	Mode of		Operation Performed
		Arguments	Function	
FMDNF	2-255	One Fixed; the others Floating	Floating	Computes the median of a set of numbers
~~FREFALF~~	~~5~~	~~Floating~~	~~Floating~~	~~Computes real root of $f(x)$ by regular falsi~~ method~~, where $arg_1 \leq x \leq arg_2$~~
~~FSIMPF~~	~~5~~	~~One Fixed and Four Floating~~	~~Floating~~	~~Computes $f(x)$ according to Simpson's Rule, where $arg_1 \leq x \leq arg_2$~~
FTENXF	1	Floating	Floating	Computes the value 10^{arg}
F2XF	1	Floating	Floating	Computes the value 2^{arg}
INTF	1	Floating	Floating	Computes the integral part of *arg*
XINTF	1	Floating	Fixed	
FINTLF	3	One Fixed; Two Floating	Floating	Interpolates within a set of points by Lagrange's formula
LOGF	1	Floating	Floating	Computes the value $\log_e arg$
LOG10F	1	Floating	Floating	Computes the value $\log_{10} arg$
FLOG2XF	1	Floating	Floating	Computes the value $\log_2 arg$
MAXF	2-30	Floating	Floating	Selects the argument with the largest value
XMAXF	2-30	Fixed	Fixed	
MAX0F	2-30	Fixed	Floating	
XMAX0F	2-30	Fixed	Fixed	
MAX1F	2-30	Floating	Floating	
XMAX1F	2-30	Floating	Fixed	

Function Name	Number of Arguments	Mode of		Operation Performed
		Arguments	Function	
MINF	2-30	Floating	Floating	Selects the argument with the smallest value
XMINF	2-30	Fixed	Fixed	
MIN0F	2-30	Fixed	Floating	
XMIN0F	2-30	Fixed	Fixed	
MIN1F	2-30	Floating	Floating	
XMIN1F	2-30	Floating	Fixed	
MODF	2	Floating	Floating	Produces Integral Remainders: MODF (arg_1, arg_2) = arg_1-INTF (arg_1/arg_2)* arg_2
XMODF	2	Fixed	Fixed	Same operation as above, using XINTF
RAND1F	One dummy fixed-point argument		Floating	Generates positive fractional random numbers
FNRANDF	2	Floating	Floating	Generates a single normally-distributed number. arg_1 = Mean, $arg_2 = \sigma$
SINF	1	Floating	Floating	Computes SIN(arg) in radians
SIN1F	1	Floating	Floating	Same as SINF
ASINF	1	Floating	Floating	Computes SIN^{-1} arg
ASIN1F	1	Floating	Floating	Same as ASINF
SIGNF	2	Floating	Floating	Transfers sign of arg_2 to $\lvert arg_1 \rvert$
XSIGNF	2	Fixed	Fixed	Same operation as above
SQRTF	1	Floating	Floating	Computes $\sqrt{arg}$
SQRT1F	1	Floating	Floating	Same as SQRTF

Function Name	Number of Arguments	Mode of		Operation Performed
		Arguments	Function	
FCSQRTF	2	Floating	Floating	Computes $\sqrt{arg}$, where *arg* is a complex number
NROOTF	2	Floating	Floating	Computes $\sqrt[arg_2]{arg_1}$
TANF	1	Floating	Floating	Computes TAN *(arg)* in radians
TAN1F	1	Floating	Floating	Same as TANF
ATANF	1	Floating	Floating	Computes $\text{TAN}^{-1}arg$
ATAN1F	1	Floating	Floating	Same as ATANF
TANHF	1	Floating	Floating	Computes TANH *(arg)*
XFIXF	1	Floating	Fixed	Converts floating-point *arg* to fixed point (Same as ~~XINFT)~~ XINTF)

For additional information on any of the above functions, the respective subroutine descriptions should be consulted.

FUNCTION SUBPROGRAMS

These are functions which cannot be defined by a single ALTAC statement, and are not used frequently enough to warrant inclusion on a library tape. A Function Subprogram is a source program, the first statement of which is a FUNCTION statement.

FUNCTION

The FUNCTION statement is the first statement of a Function Subprogram, and it identifies the function that is being defined. The general form of this statement is:

GENERAL FORM	EXAMPLES
FUNCTION $f(a_1, a_2, \ldots)$ where f is a function name that obeys rule 2 on page 45, and each a is a formal parameter.	FUNCTION HMEAN (A, B, C) FUNCTION FACTOR(PAR1, PAR2) FUNCTION INDEX (X, Y, Z)

RETURN

The RETURN statement is the last *executed* statement in a subprogram, and it returns control to the calling program. It is used in both Function and Subroutine Subprograms, and it logically precedes the END or COMPLETE statement which indicates the physical end of the subprogram (see page 59). The general form of the RETURN statement is:

GENERAL FORM	EXAMPLE
RETURN	RETURN

Defining and Calling a Function Subprogram

As is the case with Arithmetic Statement Functions and Library Functions, a Function Subprogram may be called by any expression in the main program which contains its name.

The value of the function that is returned to the calling program may be defined by means of an arithmetic statement or by an input order statement. For example, if the following subprogram

LOCATION	ALTAC STATEMENT
	FUNCTION INDEX (X,Y,Z) $
12	FORMAT (I5)$
	IF (Z) 1,1,2 $
1	INDEX = 3*X+Y**2 $
	RETURN $
2	READ 12, INDEX $
	RETURN $
	END $

is called by a program containing the statement,

```
IVALUE=INDEX(SUPPLY, DEMAND, CREDIT) $
```

the value of the function INDEX would be defined by the statement

```
INDEX=3*SUPPLY+DEMAND**2 $
```

or by the statement

```
READ 12, INDEX $
```

depending on whether the value of CREDIT is not or is greater than zero, respectively.

When a formal parameter in a FUNCTION statement is an array name, the corresponding argument in the calling statement must also be an array name. Each such array name must be defined in a DIMENSION or TABLEDEF statement in its respective source program, and all but the last dimension must correspond.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is a source program, the first statement of which is a SUBROUTINE statement. Subroutine subprograms differ from functions in two basic ways:

- Unlike a function which may be called by any expression containing its name, a subroutine subprogram can only be called by a CALL statement (see page 54).
- A function produces only a single result; a subroutine subprogram can produce more than one result.* Each result corresponds to a formal parameter of the subroutine.

SUBROUTINE

The SUBROUTINE statement is the first statement of a subroutine subprogram, and it identifies the subroutine that is being called. The general form of this statement is:

GENERAL FORM	EXAMPLES
SUBROUTINE $f(a_1, a_2, \ldots)$ where f is the name of a subroutine subprogram (see page 46), and each a is a formal parameter. The formal parameters, and the parentheses enclosing them, may be omitted from the SUBROUTINE statement.	SUBROUTINE CALC (A, B, ANS1, ANS2) SUBROUTINE RATE(PAR1, PAR2, RESULT) SUBROUTINE TREND (A, B, C, D, E) SUBROUTINE INPUT

When a formal parameter in a SUBROUTINE statement is an array name, the corresponding argument in the CALL statement (see below) must also be an array name. Each such array name must be defined in a DIMENSION or TABLEDEF statement in its respective program, and all but the last dimension must correspond.

An example of the use of the SUBROUTINE statement is presented on page 55.

* A subroutine can also be made to perform an operation and not produce a result. In this case the arguments following the subroutine name are omitted.

CALL

The CALL statement is used to call the Subroutine Subprogram whose name appears in the statement. The general form of the CALL statement is:

GENERAL FORM	EXAMPLES
CALL $f(a_1, a_2, \ldots)$ where f is the name of a subroutine subprogram (see page 46) and each a is an argument of one of the forms indicated below. The arguments may be omitted when corresponding to a SUBROUTINE statement with no formal parameters.	CALL CALC(X, Y, SOL, SIG) CALL RATE (RISK, CAPITAL, GAIN) CALL TREND (TEMP, PRESS, WIND, PRECIP, FORCAST) CALL FACTOR (WEIGHT, FUEL, THRUST) CALL INPUT

An argument appearing in a CALL statement may be in any of the following forms:

- Fixed-or floating-point expressions
- Names of arrays
- Hollerith fields

The use of a Hollerith field as an argument of a subprogram is presented below. The other types of arguments listed above were discussed on page 46.

The arguments in the CALL statement must be presented in the same order, number, form, and mode as the corresponding formal parameters in the SUBROUTINE statement.

Hollerith Arguments

Hollerith arguments may be used by a program to define a Hollerith field internally (i.e., without the use of an input statement).

The Hollerith argument in the calling statement must be of the following general form:

$$nH\ldots\ldots$$

where n is any unsigned decimal integer greater than zero. The n alphanumeric characters following the H will be translated by ALTAC into TAC word constants (W/........), eight characters per word. If n is not a multiple of eight, the unused right-most part of the last word will be filled with spaces. A word of 48 one bits will follow the last word.

A word containing the starting location of the Hollerith information is the argument transmitted to the subprogram. The corresponding formal parameter should be the name of an array that appears in a DIMENSION statement in the subprogram.

The following example illustrates the use of the CALL, and SUBROUTINE statements:

Assume A and B are two single-dimensioned arrays of 100 elements each. Define a third array C, such that for n=1,2,..., 100

$$C_n=0 \quad \text{If } A_n=0 \text{ or } B_n=0 \text{ or both}$$

otherwise

$$C_n = |A_n-B_n|$$

The necessary coding could be of the form:

LOCATION	ALTAC STATEMENT
	•
	DIMENSION A(100),B(100),C(100)$
	•
	•
	•
12	CALL CALC (A,B,C)$
	•
	•

and the subroutine could have been written as:

LOCATION	ALTAC STATEMENT
	SUBROUTINE CALC (S,T,U) $
	DIMENSION S(100),T(100),U(100) $
	DO 5 I=1,100 $
	U(I)=0. $
	IF(S(I))3,5,3 $
3	IF(T(I))4,5,4 $
4	U(I)=ABSF(S(I)-T(I)) $
5	CONTINUE $
6	RETURN $
	END $

Statement 12 in the main program transfers control to the subroutine CALC. After array C is formed, statement 6 of the subroutine returns control to the main program at the first statement following statement 12. Note that formal parameters S,T, and U in the subprogram are dimensioned, and are of the same mode, order, and number, as arguments A,B, and C in the main program.

Chapter VIII
ADDITIONAL FEATURES OF THE ALTAC SYSTEM

This chapter discusses TAC coding within an ALTAC program, the IDENTIFY statement, the I Card, Remarks Cards, and the COMPLETE and END statements.

TAC CODING WITHIN AN ALTAC PROGRAM

TAC coding in the standard TAC format may be included in an ALTAC program in either of two ways:

1. By writing the ALTAC statement

   STARTTAC $

   immediately *before* the TAC coding, and the statement

   ENDTAC $

   in columns 17-22, immediately *after* the TAC coding.

   All TAC coding between these statements is unprocessed by ALTAC and are passed on as part of the TAC program that results from the ALTAC Translation.

2. By writing a T in column 9 (column 1 when in FORTRAN format) of every TAC instruction inserted. ALTAC replaces the T in the label field with a space character, and then interprets columns 9-80 literally.

   An instruction with a T in the label field must never appear between the statements STARTTAC and ENDTAC, otherwise a label field error will be indicated by the TAC Assembler.

If the first executable statement of an ALTAC source program is a TAC instruction, this instruction, and the STARTTAC statement preceding it (if any), must be preceded by a CONTINUE statement.

IDENTIFY

The IDENTIFY statement is used to:

- Identify the format of the source program

- Indicate to the ALTAC Translator the size (amount of core storage) of the computer on which the object program will be run.
- Specify the least amount of COMMON storage that must be reserved.*

The general form of the IDENTIFY statement is:

GENERAL FORM	EXAMPLES
IDENTIFY *Type, mK, nW* where *Type*, *mK*, and *nW* are optional parameters, which are explained as follows.	IDENTIFY A, 32K, 1200W IDENTIFY F, 16K, 800W

PARAMETER	EXPLANATION
Type	"*Type*" may be A or F, indicating that the source program is in ALTAC format or in FORTRAN format.~~*~~ If statements in ALTAC format and FORTRAN format are mixed within a program, an IDENTIFY statement with the appropriate *type* parameter must precede each change in format. If the *type* parameter is omitted from the IDENTIFY statement, ALTAC assumes that the program is in ALTAC format.
mK	This parameter defines the memory size of the Philco 2000 computer on which the object program will be run. *m* may be 8, 16, or 32, denoting 8,192, 16,384 or 32,768 words of memory respectively. If the same size memory is to be used in both the compilation and run phase, this parameter may be omitted. A program that is compiled for a Philco 2000 with a larger memory may run on a Philco 2000 with a smaller memory; however, a program that is compiled for a Philco 2000 with a smaller memory may *not* run on one with a larger memory.
nW	This parameter specifies the least number of words of COMMON storage which must be contained in the program to be compiled.*

* This need only be specified when deviating from the standard mode (relocatable) of compilation (see the Philco 2000 Operating System Manual, TM-23). In this case, the first source program must make provision for the largest amount of common storage required for the entire program.

I CARD

The I Card is the first physical card of a program, and it identifies the program. The general form of this card is:

L	LOCATION	ALTAC STATEMENT
I		SAMPLEΔPROGRAMΔΔ • •

An I is written in the label column (column 9) while a name (e.g., SAMPLE ΔPROGRAM ΔΔ) identifying the program is written in columns 17-32. The name is comprised of all 16 characters (spaces included) in these columns, and is *not* terminated with a dollar sign.

REMARKS CARD

An "*" in the label field of a card indicates that all information on that card is to be interpreted as remarks, and does not affect the compilation. A Remarks Card must not appear between cards of another statement.

COMPLETE OR END

Either the COMPLETE or the END statement is used to signal to the ALTAC Translator the end of the program being compiled. The COMPLETE or END statement must be the last physical statement in the source program. The general forms of these statements are:

GENERAL FORMS	EXAMPLES
COMPLETE	COMPLETE
END	END

Appendix A
TABLE OF PHILCO CHARACTERS

PHILCO CHARACTER	OCTAL CODE	HOLLERITH PUNCH
0	00	0
1	01	1
2	02	2
3	03	3
4	04	4
5	05	5
6	06	6
7	07	7
8	10	8
9	11	9
@	12	8-2 ①
=	13	8-3
;	14	8-4
≡	15	8-5 ①
&	16	8-6 ①
'	17	8-7
+	20	12
A	21	12-1
B	22	12-2
C	23	12-3
D	24	12-4
E	25	12-5
F	26	12-6
G	27	12-7
H	30	12-8
I	31	12-9
n ②	32	12-8-2 ①
.	33	12-8-3
)	34	12-8-4
%	35	12-8-5 ①
?	36	12-8-6 ①
"	37	12-8-7 ①

PHILCO CHARACTER	OCTAL CODE	HOLLERITH PUNCH
-	40	11 or 8-4 ①
J	41	11-1
K	42	11-2
L	43	11-3
M	44	11-4
N	45	11-5
O	46	11-6
P	47	11-7
Q	50	11-8
R	51	11-9
⌉	52	11-8-2 ①
$	53	11-8-3
*	54	11-8-4
<	55	11-8-5 ①
#	56	11-8-6 ①
⊔	57	11-8-7 ①
Blank (space)	60	Blank
/	61	0-1
S	62	0-2
T	63	0-3
U	64	0-4
V	65	0-5
W	66	0-6
X	67	0-7
Y	70	0-8
Z	71	0-9
|	72	0-8-2 ①
,	73	0-8-3
(	74	0-8-4
>	75	0-8-5 ①
:	76	0-8-6 ①
e ②	77	0-8-7 ①

① Multiple punched.

② These two characters are not acceptable ALTAC characters, and are included here only to show the complete character codes.

Appendix B
SUMMARY LIST OF ALTAC STATEMENTS

This appendix provides a convenient reference to all ALTAC statements discussed in the manual.

STATEMENT	TYPE	PAGE REFERENCE
$v = e$	Arithmetic	13
GO TO n	Control	15
GO TO m or GO TO $m, (n_1, n_2, \dots, n_k)$	Control	15
ASSIGN n to m or ASSIGN (n) to m	Control	16
GO TO $(n_1, n_2, \dots, n_m), i$	Control	16
IF (e) n_1, n_2, n_3 or IF $(e_1) : (e_2), S$	Control	17
SENSE LIGHT i	Control	18
IF (SENSE LIGHT i) n_1, n_2	Control	19
IF (SENSE SWITCH i) n_1, n_2	Control	19
IF (SENSE BIT i) n_1, n_2	Control	19
IF OVERFLOW n_1, n_2	Control	20
DO n $i = m_1, m_2, m_3$ or DO (n) $i = m_1, m_2, m_3$	Control	20
CONTINUE	Control	23
PAUSE n	Control	23
STOP	Control	23
EQUIVALENCE $(v_1, v_2, v_3, \dots), (v_k, v_{k+1}, \dots), \dots$	Specification	26
COMMON $v_1, v_2, v_3, \dots$	Specification	27
DIMENSION $v_1, v_2, v_3, \dots$	Specification	25
TABLEDEF $v_1, v_2, v_3, \dots$	Specification	28
READ n, *List*	Input	32
READ TAPE i, *List*	Input	33
READ INPUT TAPE i, n, *List*	Input	32
PRINT n, *List*	Output	33
PUNCH n, *List*	Output	34

STATEMENT	TYPE	PAGE REFERENCE
PUNCH OUTPUT TAPE *i, n, List*	Output	34
WRITE TAPE *i, List*	Output	35
WRITE OUTPUT TAPE *i, n, List*	Output	35
END FILE *i*	Output	35
RUNOUT *i*	Input/Output	36
BACKSPACE *i*	Input/Output	36
REWIND *i*	Input/Output	37
LOCKOUT *i*	Input/Output	37
FORMAT $(d_1, \ldots, d_n)$	Input/Output	37
FUNCTION $f(a_1, a_2, \ldots)$	Subprogram	51
SUBROUTINE $f(a_1, a_2, \ldots)$	Subprogram	53
CALL $f(a_1, a_2, \ldots)$	Subprogram	54
RETURN	Subprogram	51
IDENTIFY *Type, mK, nW*	Compiler Control	58
COMPLETE	Compiler Control	59
END	Compiler Control	59

INDEX