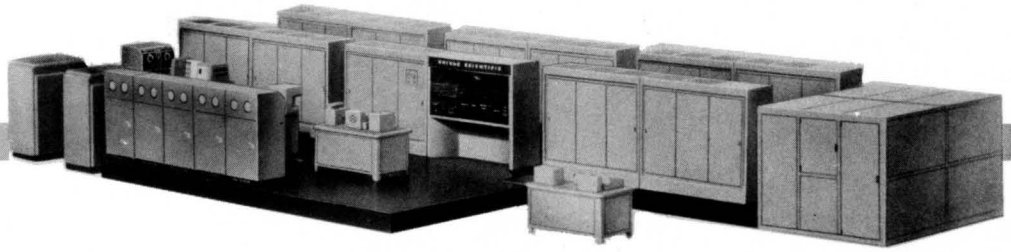
UNICODE



Preliminary Reference Manual

Automatic Coding

for the Univac® Scientific Data Automation System (1103A)

Remington Rand Univac
DIVISION OF SPERRY RAND CORPORATION

INTRODUCTION

UNICODE--NEW PROGRAMMING EASE

In *UNICODE*, automatic programming has reached a new pinnacle of development. *UNICODE* enables the engineer, the physicist, the mathematician--in fact anyone familiar with algebra--to become a programmer...to express his problem concisely in an easy-to-learn form the Univac Scientific 1103-A can use to produce a processing program. *UNICODE* may be employed to code a program for any mathematical problem with a numerical solution. Its best application is in the field of "one shot" problems, that is, problems for which quick solutions are desired and which will be run on the computer a limited number of times. If a problem is to be run many times, the operation time on the computer may warrant a professional programmer's coding to save computer time. But even for such problems, the programmer can reduce his work by having *UNICODE* prepare portions of the program. Problems containing sections requiring many logical operations also can be more efficiently prepared by a professional programmer. But in problems of this type also, these sections can be programmer-coded as subroutines and placed in the *UNICODE* library, with the remainder of the problem then being coded by *UNICODE.* Thus the *UNICODE* library of subroutines is such that it is easy to add, subtract, or change routines, which permits a computer installation to develop a library adjusted to its specific needs. And compatibility with AT-3 or *MATH-MATIC*, the algebraic compiler for Univac II, enables interchange of data input tapes between Univac II and the Univac Scientific.

UNICODE OFFERS TRUE EASE OF EXPRESSION

The *UNICODE* system allows for the expression of any decimal constant. And all constants except those used for incrementing subscripts are converted to floating point numbers with approximately nine decimal digits of significance.

With but few exceptions, any symbol of alphanumeric characters (only one of which must be alphabetic) may be used to express a variable. The *UNICODE* system includes functional notation with exactly the same meaning and notation as that commonly used in mathematical literature except in a few instances. This feature eliminates the necessity of restating an equation in order to use it later in a program. A function may be expressed with as many as four arguments. The arguments may all be subscripted variables, all nonsubscripted variables, or a combination of nonsubscripted and subscripted variables. An

argument of a function may in virtually all instances be written as an expression; an equation may be considered both in a general sense or included as a statement. Parentheses are used in the usual way to order computation in a mathematical expression.

UNICODE INSTRUCTIONS ARE EASY TO LEARN, EASY TO USE

To facilitate use of *UNICODE*, the number of instructions in the system have been kept to a minimum consistent with providing simplicity and flexibility in use and power in effect. The instructions are

Replace-- Substitutes the value of an expression or single variable for the value of a designated variable. Two or more *Replace* instructions may be combined in a single statement by the connecting word *and*.

Compute-- Evaluates a variable using its defining equation. If the variable is written in subscript or functional notation form, all values computed to the maximum number specified by the dimension statement are retained for future reference; if not, only the current value is retained. Two or more *Compute* instructions may be combined in a single statement by the connecting word *and* (equations written in the statement region perform the same function as the *Compute* instruction).

Vary-- Controls the values of a variable used in repetitions of a certain range of statements. It consists of three parts: modify, range, and transfer. Both the modify and range parts may take either of two forms. The connecting word *and* may be used to combine several modify components into one *Vary* instruction when the variables of the modify parts take on their successive values concurrently.

Jump-- Transfers control to another statement.

If-- Transfers control to another statement when a certain condition is satisfied; if the specified condition is not met, continuation to the next statement in sequence is implied.

Resume-- Returns control to a *Vary* instruction so the modifying process is continued from the point at which it was last ter-

minated. If transfer of control to a *Vary* instruction is accomplished by any other means, the *Vary* sequence will begin with initial conditions as written in the *Vary* instructions.

Write-- Records on a specified tape edited information to be printed by the off-line High-Speed Printer. Any result printed will have a fixed point representation unless the decimal point appears outside the range of significant digits. If the number cannot be represented in fixed point form standard scientific notation is used. When a single, subscripted variable appears in the *Write* instruction, subscripts are written automatically. The *Write* instruction can be used in two ways--with one variable, or with two to five variables.

Type-- Enables designation of variables whose values are to be typed out on the Supervisory Control Typewriter; usually used to type out small amounts of information. The variable referenced by the *Type* instruction may be subscripted, single-valued or a function.

Print-- Permits indication of special conditions during the running program. Also used with the Supervisory Control Typewriter, this instruction produces an output identical to that appearing within the parentheses following the word *Print*.

Start-- Indicates the point at which actual execution of a program is to begin; sentences preceding it are not executed.

Stop-- Indicates a point at which execution of the Object (Output) program or subprogram is to be terminated; is translated into an instruction stopping computer operations.

End of Tape--Indicates all pertinent information for the problem contained on a tape has preceded.

Dimension-- Specified number of values of a subscripted variable and/or function that will be retained at any one time. When used with a subscripted variable, this instruction also specifies the largest value the subscript may assume.

Delete-- Eliminates entire *Unicode* lines from the program when correcting errors.

Change-- Replaces the *Sentence* and *Comment* parts of a specified line by the correct *Sentence* and *Comment* (if any), leaving the line number unchanged.

Insert-- Adds new lines at specified points in a *Unicode* program. The programmer need only number the first line to be added.

Sub-- Identifies a subprogram with a pseudo operation (an operation performed by a subprogram written in *Unicode* language) through the pseudo operation symbol used in the program.

Result-- Names the single valued output of the subprogram.

Exit-- Returns control to the sentence of the program referencing the pseudo operation.

UNICODE PERFORMS MANY ROUTINE PROGRAMMING TASKS ITSELF

Many of the time consuming details of programming a problem are eliminated or drastically reduced by *UNICODE*. The pseudo operation, for instance, enables the writing of specialized routines for a particular problem. This makes available as a supplement to the existing library any sequence of instructions that will be used repeatedly in the problem both with or without different operands. The person programming can thus express his main problem in *UNICODE* language without the distraction of excessive detail. A pseudo operation may have any number of operands and parameters so long as the *SUB* sentence in the subprogram has the same number and is written in the same order. The same pseudo operation symbol may appear as often as necessary in any number of equations.

UNICODE numbers all lines not assigned a line number by the person programming the problem. Thus, in his corrections to a *UNICODE* program following a translation by *UNICODE*, the person programming may refer to any assigned line number--including those assigned by *UNICODE*. The person programming may also use machine language in the *UNICODE* source program if he wishes to code a logical portion himself. He also need not subdivide the Object program if it is too large for the machine on which it is to be run--*UNICODE* does this itself automatically if the need arises.

UNICODE reduces the debugging task to a small fraction of what it would be under hand programming. A big factor bringing about this reduction is the

elimination by *UNICODE* of the "housekeeping" phase of programming--the instruction modifications, line designation, program subdivision, and similar "clerical" functions. While the ease of programming with *UNICODE* eliminates most errors, those which still occur are easily located with the aid of the many error checks built into this automatic programming system. Corrections to a *UNICODE* program are easy too. A tape containing the corrected information is mounted on a separate Uniservo magnetic tape unit and correct information merged with the original program to produce the updated program tape.

UNICODE PROGRAMS ARE EASILY PREPARED FOR ENTRY INTO THE COMPUTER

It is not necessary to put the *UNICODE* input program on punched cards and then run them through a card-to-tape converter before compilation--the program can be transcribed directly onto magnetic tape with the Unityper. Each line of the Unityped copy of the *UNICODE* program will contain 120 characters (including fill symbols) representing one blockette of information on magnetic tape. Additional tapes may be used to handle a *UNICODE* program too long for one tape; these tapes are then merged with the original program in the same manner as correction tapes.

The edited record produced by *UNICODE* is also easy to understand. Printed out on the High-Speed Printer as a side-by-side listing, the edited record contains both the input program and the generated Object (output) program. Each *UNICODE* line of the input program is reproduced as it appeared in the Unityped copy, with the addition of the line numbers assigned by *UNICODE*. Immediately following each such line, the corresponding generated machine code is printed, with one machine word per output line. A blank space is left in the side-by-side listing wherever the space symbol appeared in the Unityped input program.

All of these features--and others, too--combined thus make *UNICODE* an extremely useful programming aid--one that not only drastically reduces computer operating expenses but also greatly accelerates new developments in programming research and computer applications.

CONTENTS

GENERAL FORMS

This chapter illustrates the general appearance of the *UNICODE* program, subprogram, and line. Capitalization indicates those sentences which must appear.

PROGRAM Two blockettes (240 characters) in the first block of any program are allowed for the program title. Every *UNICODE* program must be written as follows:

UNICODE Programming Sheet Page_____		
Line Number (Character Positions 1-12)	Sentences and Comments (Character Positions 14-120)	
	UNICODE PROGRAM	1st blockette (must appear)
	Title	4th & 5th blockette of 1st block (if present)
	Equations	Start 2nd block of tape (if present)
	START	Indicates statements follow (must appear)
	Statements	STOP instruction (must appear)
	Subprograms	In programs using pseudo operations
	END OF TAPE	(must appear)

Figure 1 - General Form of a UNICODE Program

SUBPROGRAM A subprogram in a problem indicates use of a pseudo operation. The general form of a subprogram is similar to that of a program except that the first two sentences of a subprogram are reserved for the heading. For clarity in the following figure, assume the pseudo operation symbol is *OPER* and the operands are A and B.

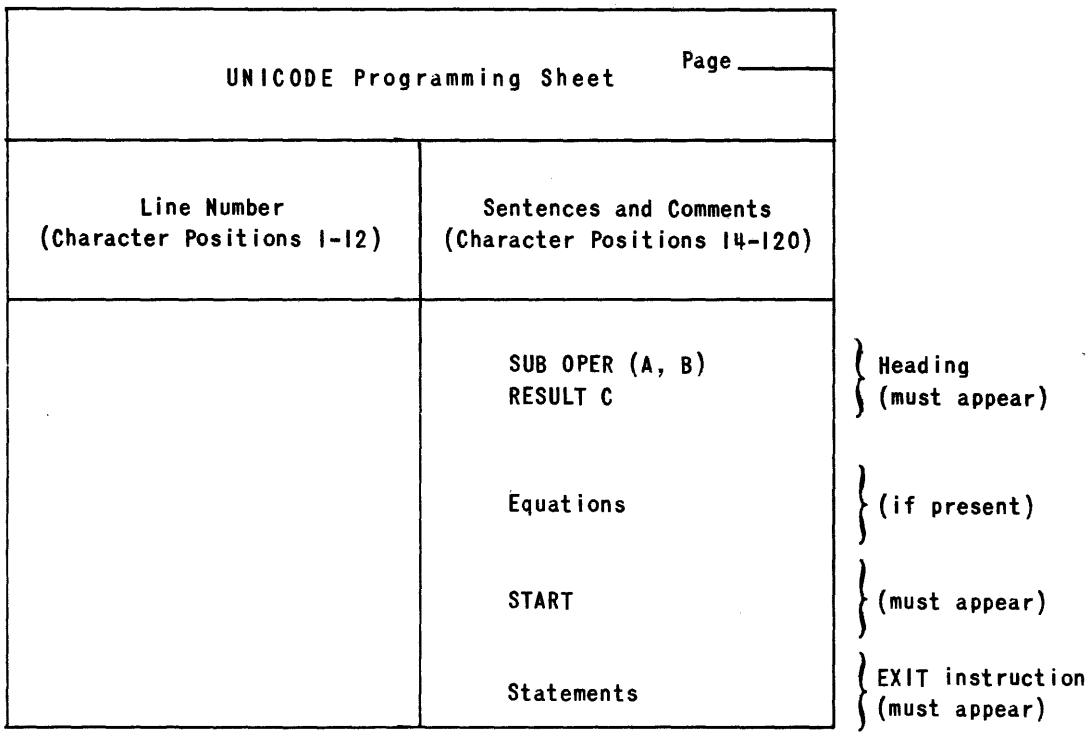


Figure 2 - General Form of a Subprogram

LINE The line is the fundamental unit of the unicode language, consisting of a line number (used to reference the line), a sentence (which is interpreted by *UNICODE*), and comments. The line number and comments are optional.

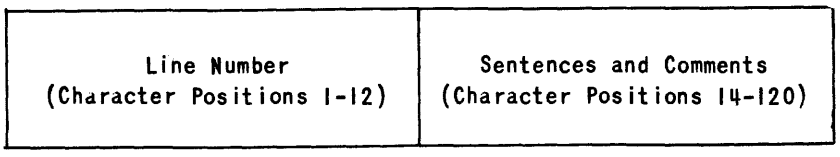


Figure 3 - General Form of a Line

Line Number	Sentences and Comments	
	VARY X 1(0.1) 100 SENTENCE 39 THEN JUMP TO 5Δ.	Λ. is end of sentence symbol
	F(0) = 13Δ. (SET INITIAL CONDITION)	Comment in parenthesis
	COMPUTE Y(I,J)Δ.	

Figure 4 - Examples of 3 Unrelated Lines

Chapter 2

NOTATION

CONSTANTS The *UNICODE* system allows for the expression of any decimal constant. All constants, except those used for incrementing subscripts, are converted to floating point numbers with approximately nine decimal digits of significance. This allows a constant to have *magnitude* between 10^{-38} and 10^{38}, and zero. Since the character combination space period Δ. indicates the end of a sentence, a constant which has no integral part must be expressed as zero point o.___. The characters + - * / denote addition, subtraction, multiplication, and division, respectively.

Examples:

```
123.95
0.12395*10 POW 3
12395*10^-2
```

VARIABLES Any symbol of alphanumeric characters, one of which must be alphabetic, may be used to express a variable. Exceptions are those character combinations which are used elsewhere in the language, such as *SIN, POW, VARY.*

Examples:

```
X
CAT
DS1958
```

SUBSCRIPTS A sequence of symbols or mathematical expressions enclosed within parenthesis immediately following a variable is used to denote the subscripts of a variable. To distinguish between a subscripted variable and functional notation (see next section), the symbol of any variable used in a subscript must have one of the letters, I, J, K, L, or M as its first character. Multiple subscripts on a variable, to a maximum of four, are separated by commas. A subscript may not have a subscript and may take only a positive integral value.

Examples:

```
X(5)
Y(I,J,)
Z(I+3, J+K)
```

FUNCTIONAL NOTATION

Functional notation in the *UNICODE* system has exactly the same meaning and notation as that commonly used in mathematical literature, except that an argument symbol cannot have the letters I, J, K, L, or M as its first character.

Example:

$$F(X,Y,Z,T) = X^2+Y^2+Z^2 - \text{SIN } T$$

When a subscripted variable serves as the argument of a function, parentheses within parentheses are necessary to separate the subscript from the argument.

Example:

$$F(X(I), Y(J)) = X(I) - \text{TAN } Y(J)$$

If a function has subscripted arguments, any future reference to a computed value of the function must be made through particular values of the subscripts. In the above example the function F is defined in terms of the subscripted arguments, X(I) and Y(J). Following computation, a reference to a particular value of the function F might be F(X(5),Y(7)). The reference to a value of F could not be written as F(10.3, 23.4). This restriction applies only to subscripted arguments.

The *UNICODE* system allows a function to be expressed with as many as four arguments. These arguments may be all nonsubscripted variables, all subscripted variables, or a combination of nonsubscripted and subscripted variables. If a subscripted variable is used as the argument of a function, it may not have more than one subscript.

An argument of a function may be written as an expression, except when the function appears on the left of an equality symbol or preceding the word "*BY*" in the "*REPLACE*" instruction. The following are examples of *incorrect* usage of functional notation in the *UNICODE* system:

```
F(X+A-B,Y,Z) = ___.
REPLACE F(X+A-B,Y,Z) BY ___.
```

The *UNICODE* system does not permit direct expression of a function which has another function as its argument.

When two or more functions are defined in terms of the same argument, such as F(X) and G(X), the argument must assume the same values and the same order in the evaluation of the various functions.

Functional notation is convenient in expressing certain problems, but may result in somewhat slower program than if subscript notation is used. The ease of expression attributed to functional notation, however, overbalances this relatively slight time increase, which is required by the manipulation of arguments and their associated function values.

EQUATIONS

An equation in *UNICODE* language is the definition of a single variable on the left of an equality symbol by a mathematical expression on the right. An equation may be considered in a general sense or it may be included as a statement. When used in the general sense the equation can be referenced many times and is written preceding the *START* instruction; when included as a statement, the equation calls for the evaluation of the expression on the right and the substitution of the resulting value for the value of the variable defined. If an equation is written separately from the statements, no computation occurs until the variable is referenced by a statement.

Any variable on the left of an equation must be defined explicitly by the expression on the right; that is, the expression must not contain the variable appearing on the left (unless the subscripts or arguments have different values), nor may it con-

tain any other variable which is dependent upon the variable being defined.

Examples:

```
Y = X  POW 2 - 5 * SIN (A+B)
C(I,J) = A(I,K) + D * B(K,J)
F(X,Y) = X^2 + COS Y
F(X) = F(X-H) + COS X
```

If the variable on the left is subscripted, or is functional notation, an operation symbol may not be included within the subscript or argument. Thus

```
X(I+1) = X(I) - 2
```

illustrates an equation which is *not* permissible since the subscript on the left contains an operation symbol (+). For an equivalent meaning this could be expressed as

```
X(I) = X(I-1) - 2
```

with proper adjustment of I.

HIERARCHY OF OPERATIONS

Parenthesis are used in the usual way to order computation in a mathematical expression. When the order of computation is not specified through the use of parenthesis, however, the operations are performed in the following order:

1. Exponentiation
2. Library Functions and Pseudo Operations
3. Multiplication and Division
4. Addition and Subtraction

In the equation,

```
Z = X + Y^2 /3 * SIN A - B
```

the computation will be carried out as if it had been written

$$Z = X + [(Y^2 /3) * (SIN A)] - B$$

If grouping parentheses are omitted from a consecutive sequence of operations which appear within the same hierarchy, the grouping is understood to be from left to right. Thus, X/Y * Z means ((X/Y) * Z), operating from innermost to outermost parentheses.

Chapter 3

INSTRUCTION REPERTOIRE

To facilitate the presentation and explanation of the *UNICODE* instruction, the following special notation will be used:

1. The letters X, Y, and Z denote *single variables.*
2. The letters a, b, and c denote *expressions.*
3. The letters k, m, and n denote *line numbers.*

REPLACE

The *REPLACE* instruction substitutes the value of an expression or single variable for the value of a designated variable. The previous value of the variable is not retained. When X denotes a subscripted variable or functional notation, the subscripts or arguments *may not* be expressions.

The value of any variable appearing in the expression *a* must be determined prior to the execution of the *REPLACE* instruction. This value may have been obtained as input data, through the computation of an equation, or it may have been assigned a value by a *VARY* instruction or another *REPLACE* instruction.

Example:

```
REPLACE X BY a.
```

In the example, the value of the expression *a* becomes the current value of the variable X. The previous value of X is not retained for future reference.

Two or more consecutive *REPLACE* instructions may be combined in a single statement with the connecting word *AND*.

Example:

```
REPLACE X BY a AND Y BY b.
REPLACE X AND Y AND Z BY a.
```

COMPUTE

The *COMPUTE* instruction evaluates a variable using its defining equation. If the variable is written in subscript or functional

notation form, all values computed, to the maximum number as specified by the dimension statement, are retained for future reference; if not, only the current value is retained.

Before a variable can be computed using its defining equation, all variables which appear in the right hand member must have been previously evaluated; that is, they must havebeen assigned values as input data or by a *VARY* or *REPLACE* instruction, or, if defined by equations, they must have been computed.

Example:

```
COMPUTE Z.
```

Two or more consecutive *COMPUTE* instructions may be combined into a single statement with the connecting word *AND*.

Example:

```
COMPUTE X AND Y AND Z.
```

The computations are performed from left to right. In the example, X is evaluated first, then Y, then Z.

(X = a - Equations written in the statement region perform the same function as the *COMPUTE* instruction. When control reaches such an equation, the variable is evaluated using the right hand expression. An equation appearing with other control statements may not be referenced by a *COMPUTE* statement. See *EQUATIONS* section, Chapter II.)

VARY

The *VARY* instruction controls the values of a variable used in repetitions of a certain range of statements. It consists of three parts: the Modify, Range, and Transfer components.

The *Modify component* may be either of two types:

(1) Increment type

```
VARY X p(q)r ___.
```

In this type the variable X is incremented by q from an initial value of p until $|r - X| < |q|$.

(2) List type

VARY X r,s,t, ___.

In this type the values r,s,t, etc. to a limit of *ten* values are substituted for X in the order stated. In the illustrations above and following X must be a nonsubscripted variable; p,q,r, s,t, and u are nonsubscripted variables or constants.

The *Range component* specifies the statements which are controlled by the *VARY* instruction. If the range of a *VARY* statement A includes another *VARY* statement B, then the complete range of statements for B must be included in the range of A. Reference ***may not*** be made to any statement within the range of a *VARY* instruction from another statement outside the range. This component may also take either of two forms:

(1) ___ SENTENCES k THRU m ___

This form indicates that all statements, from the statement with line number k to and including the statement with line number m, will be repeated each time the variable in the Modify component takes on a new value.

(2) ___ SENTENCE k ___

This is a special case of the above in which the statement with line number k is the only statement subject to repetition. In both cases, the statement numbered k immediately follows the *VARY* statement.

The *Transfer component* specifies the line number of the statement to which control will be transferred upon termination of the *VARY* sequence; that is, when all statements of the range have been executed for all values of the variable in the Modify component. The Transfer component is optionally stated or not stated; however, if it is not specified, control will be transferred to the first statement following the range of the *VARY*

instruction. The Transfer component requires the following form:

___ THEN JUMP TO n.

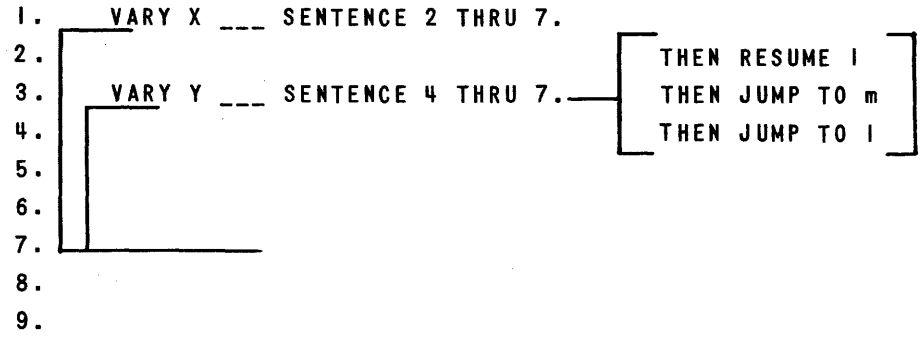
When the ranges of two *VARY* statements extend to exactly the same statement, the Transfer component on the inner *VARY* statement takes an additional option and the ***unstated*** transfer has a different meaning. The Transfer component on the inner *VARY* statement may take one of three forms:

1. THEN JUMP TO m (as before).
2. THEN RESUME k.
3. Not stated.

The *THEN RESUME k* transfer continues the modifying process of the *VARY* statement at k, which ***must*** be the ***preceding*** *VARY* statement. The unstated transfer, instead of its usual transfer to the statement following the range of the *VARY*, automatically causes a "*RESUME*" of the previous *VARY*. Thus, the optional *THEN RESUME k* or the omission of a transfer component on an inner *VARY* statement accomplish the same thing.

If a *THEN JUMP TO k* transfer to the preceding (or any other) *VARY* statement is used, the variable modified by the *VARY* statement at k is reset to its initial value.

Illustration:



The first *VARY* statement (at 1) follows the conventional rule of transfer; i.e., after completion of the *VARY* statement con-

trol is transferred to the statement at 8. The transfer from the second *VARY* could be:

1. Not stated- control transfers to 1, X takes next value.
2. THEN RESUME 1- control transfers to 1, X takes next value.
3. THEN JUMP TO m- control transfers to some statement numbered m.
4. THEN JUMP TO 1- control transfers to 1, resetting X to its initial value.

The connecting word *WITH* may be used to combine several Modify components into one *VARY* instruction when the variables of the Modify components take on their successive values concurrently. The variables of all Modify components take on their initial values the first time through the range, their second values the second time, etc. until the limiting value of *any* of the variables is reached at which time the *VARY* sequence is terminated.

```
VARY X p(q)r WITH Y s(t)u ___
```

Following are examples of the forms which the complete *VARY* instruction may take:

```
VARY X r(s)t SENTENCE k THRU m THEN JUMP TO n.
VARY Y p,q,r,___,s,t, SENTENCE k THEN JUMP TO n.
VARY X r(s)t SENTENCE k THRU m.
VARY Y p,q,r,___,s,t SENTENCE k.
VARY X p(q)r WITH Y s(t)u SENTENCE k THRU m.
VARY X r(s)t WITH Y p,q,___,s,t SENTENCE k.
```

When the starting value of a variable as written in the Modify component is itself a variable, this variable *must assume* a numerical value prior to the time control reaches the *VARY* instruction. In the last example above, r and p must have numerical values prior to the execution of the *VARY*. Further, numerical values *must be established* for *all* other variables in the Increment type of Modify component(s) prior to or during the first execution of the range of the *VARY*. If the Modify component is of the List type, however, a numerical value must be determined only for the next successive variable to be used.

When control is returned to a *VARY* instruction through a *RESUME* instruction, the *VARY* sequence will be continued from that point in the modifying process at which it was last terminated. If transfer of control to a *VARY* instruction is accomplished by any other means, however, the *VARY* sequence will begin with initial conditions as written in the *VARY* instruction.

JUMP

The *JUMP* instruction transfers control to another statement.

Example:

```
JUMP TO SENTENCE k.
```

In the example control is transferred to the statement with line number k, then proceeds in sequence from this statement until interrupted by another transfer of control.

IF

The *IF*, or conditional, instructions transfer control to another statement when a certain condition is satisfied. If the specified condition is not met, continuation to the next statement in sequence is implied. The following examples indicate the *IF* instructions that may be used:

Examples:

```
IF X = Y, JUMP TO SENTENCE k.
IF X NOT = Y, JUMP TO SENTENCE k.
IF X < Y, JUMP TO SENTENCE k.
IF X > Y, JUMP TO SENTENCE k.
IF X < = Y, JUMP TO SENTENCE k.
IF X > = Y, JUMP TO SENTENCE k.
```

In the examples a transfer of control to the statement with line number k is accomplished only if the specified relationship between X and Y exists. X and Y may be subscripted or nonsubscripted variables or functions, but *may not be expressions.*

RESUME

The *RESUME* instruction returns control to a *VARY* statement to continue the modifying process from that point at which it was

last terminated. That is, the variable of the modify component in the *VARY* instruction referenced takes on its *next* value, rather than initial value, prior to entering the range of the *VARY*. The *RESUME* instruction *may refer only* to a *VARY* instruction. In the example control is transferred to the *VARY* instruction with line number k, which contains the modifying process.

Example:

```
RESUME k.
```

The *WRITE* instruction records on a specified tape edited information which is to be printed off-line by the High-Speed Printer. Any result printed will have a fixed point representation unless the decimal point appears outside the range of significant digits. If the number cannot be represented in a fixed point form, standard scientific notation is used; that is, one digit to the left of the decimal point and a power of ten to the right of the digits.

If a *single, subscripted* variable appears in the *WRITE* instruction, the subscripts will automatically be written.

Examples:

```
A.  WRITE R(I,J), ((INVERSE MATRIX)), TAPE 3.
B.  WRITE F(X,Y,Z,T), X,Y,Z,T,
    ((POSITION FUNCTION)) () (DIR. OF FIRE)
    (RIGHT-LEFT) (HORIZ.-VERT.)
    (TIME), TAPE 7.
```

The instruction in example A produces a printed copy in the following format:

```
                    R(I,J)

              INVERSE MATRIX

                  I = I

   1           2           3           4           5
[R(I,1)]    [R(I,2)]    [R(I,3)]    [R(I,4)]    [R(I,5)]

   6           7
[R(I,6)]    [R(I,7)]
```

I = 2

1	2	3	4	5
[R(2,1)]	[R(2,2)]	[R(2,3)]	[R(2,4)]	[R(2,5)]

6	7
[R(2,6)]	[R(2,7)]

-
-
-
etc.

where the brackets [] mean "*the value of*" the variable enclosed, such as, "*the value of the variable R(1,1)*". In the example the order of J is 7. The last subscript written as J above is increased to its limit for each value of the preceding subscript(s) as I above regardless of the storage format within the computer.

When only *one subscripted* variable appears in the ***WRITE*** instruction, it will always be printed in five columns unless the order of the fastest varying subscript is less than 5. When more than one variable appears in the ***WRITE*** instruction, the number of columns is determined by the number of variables. If printed column headings are desired when a function and its arguments are to be printed, these headings are enclosed in parentheses following the title, which is enclosed in double parentheses. The headings may contain a maximum of 18 characters. Where no column heading is desired, a blank set of parentheses is used. To illustrate, Example B would produce a format as follows:

POSITION FUNCTION

X	Y	Z	T
DIR. OF FIRE	RIGHT-LEFT	HORIZ.-VERT.	TIME
-	-	-	-
-	-	-	-
-	-	-	-

Thus, there are two ways of using the ***WRITE*** instruction with one variable or with 2 to 5 variables. No more than five variables may be written in a single ***WRITE*** instruction. With one

variable the values of any subscripts are written automatically and the variable printed in 5 or less columns, depending on the order of the last written subscript. If there are two to five variables, they are printed side by side with one column for each variable. In this case, subscripts or arguments are not printed unless specifically requested in the *WRITE* instructions.

Only one *WRITE* instruction may appear in the range of a *VARY* statement. If the *WRITE* instruction does not appear in the range of a *VARY* statement, all values of the specified variables are recorded on tape.

TYPE

The *TYPE* instruction enables designation of variables whose values are to be typed out on the Supervisory Control Typewriter. This instruction is most frequently used to type out small amounts of information rather than to point long lists of data. The variable referenced by the *TYPE* instruction may be subscripted, a function, or single-valued. If it is subscripted or a function, however, the values of the subscripts or arguments must be determined prior to execution of the instruction.

Example:

```
TYPE (A(I,J), F(X), Y).
```

If the values of I and J were 2 and the value of X was 2.5 when the *TYPE* was executed, the output would be:

```
A(2.2) = [A(2.2)]
F(2.5) = [F(2.5)]
  Y    =   [Y]
```

with the brackets denoting "*the value of.*"

PRINT

The *PRINT* instruction, also used with the Supervisory Control Typewriter, permits indication of special conditions during the running program. The output is always that which appears within the parentheses following the word *PRINT*.

Example:

```
PRINT (A(I,J) SINGULAR).
```

In the example the output would always be *A(I,J) SINGULAR*, whether or not I and J had specific values at the time of execution.

START The *START* instruction indicates the point at which actual execution of a program is to begin. Sentences preceding the *START* instruction are not executed. These sentences usually include the name of the program, those defining equations referred to by a *COMPUTE* instruction, and the *DIMENSION* sentence.

STOP The *STOP* instruction indicates a point at which the execution of the Object program or sub-program is to be terminated. This instruction may appear at several points in a program and is translated into an instruction which stops computer operation. The first *STOP* instruction encountered terminates the program.

END OF TAPE The *END OF TAPE* instruction indicates that all pertinent information for the problem contained on a tape has preceded. It must appear on each tape to be used for a problem.

DIMENSION The *DIMENSION* instruction specifies the *number* of values of a subscripted variable and/or function that will be retained at any one time. The *DIMENSION* instruction if required must appear as the first sentence in the *UNICODE* program. When associated with a subscripted variable, the *DIMENSION* instruction also specifies the largest *value* that the subscript may assume. Every variable written in subscript and/or functional notation form *must* appear in a *DIMENSION* instruction. Thus, in the example below both the subscripted variable Z(I,J) and the function G(X,Y) appear in the *DIMENSION* instruction.

Example:

```
DIMENSION Z(10,8), G(5,3)
```

This example indicates that the subscript I takes on at most 10 different values and J assumes at most 8 different values with the variable Z. Neither subscript I nor J, however, may assume

a value exceeding the dimensions 10 and 8 as stated for the variable. Thus, a reference to Z(12,7) is not permissible.

The example also indicates that the argument X has a maximum of 5 different values retained at any time, and Y has at most 3 values retained. Actually, the arguments X and Y may assume an unlimited number of different values throughout the problem, but only the last 5 values of X and the last 3 values of Y are available for reference. The Differential Equation problem of Appendix B illustrates the use of the *DIMENSION* instruction in this way.

Based on the *DIMENSION* instruction in the example above, 80 computer storage locations are set aside for the variable Z, and 15 for the function G.

SUB-PROGRAM INSTRUCTIONS The instructions *SUB*, *RESULT*, and *EXIT* are used only in *SUB-PROGRAMS* (See Chapter IV).

CORRECTION INSTRUCTION The instructions *DELETE*, *INSERT*, and *CHANGE* are used to make corrections in the *UNICODE* program (See Chapter VII).

PSEUDO OPERATIONS AND SUBPROGRAMS

The pseudo operation enables writing specialized routines for a particular problem. This feature of the *UNICODE* system makes available to the user, as a supplement to the existing Library, any sequence of instructions used repeatedly in the problem, usually with different operands.

SYMBOL The symbol denoting a pseudo operation is identical to that of a variable (six alphanumeric characters of which one must be alphabetic) but is distinguished from a variable in that it appears in the heading of a subprogram. (See below). It is written as an operator for an equation, with its operands and parameters enclosed within parentheses immediately following. The same pseudo operation symbol may appear as often as necessary in any number of equations.

HEADING OF A SUBPROGRAM The heading of a subprogram is made up of two sentences, *SUB* and *RESULT*. Through the *SUB* sentence the subprogram is named with the pseudo operation symbol used in the program. The *RESULT* sentence names the *single-valued* output of the subprogram.

OPERANDS AND PARAMETERS The operands and parameters, which are used only in writing the subprogram, appear following the pseudo operation symbol in the heading. A pseudo operation may have any number of operands and parameters so long as the *SUB* sentence in the subprogram has the same *number*, written in the same order.

EXIT The *EXIT* instruction returns control to the sentence of the program which referenced the pseudo operation.

In the following example, the heading of a subprogram corresponding to a pseudo operation called *DOT* might be:

```
SUB    DOT (G(J), H(J), N)
RESULT R
```

In Figure 1, G(J), H(J), N, and R are "*dummy*" variables in the subprogram *DOT* and will be equated to variables of the problem prior to any execution of the subprogram.

Suppose *DOT* is the symbol for a pseudo operation which evaluates scalar products of vectors of various dimensions, and in which the following two equations occur:

```
X(I) = B(I) N DOT (C(I), A(I),3) - A(I) * DOT (C(I), B(I),3)
Y(I) = E(I) * DOT (F(I), D(I),10) - D(I) * DOT (F(I), E(I),10)
```

The subprogram for *DOT* would then be:

```
LINE NUMBER        SENTENCES AND COMMENTS

                   SUB DOT (G(J), H(J), N)
                   RESULT R
                   START
                   R = 0
                   VARY J1(I) N SENTENCE I
         I         REPLACE R BY R + G(J) * H(J)
                   EXIT
```

When the first term of the equation for X(I) is evaluated, the "*dummy variables*" G and H are equated to C and A and N is set equal to 3. When the second term is computed, G and H are equated to C and B and N is again set to 3. Also, when the first term of Y is evaluated, G and H are equated to F and D respectively and N is set to 10. In each case the single-valued result R is a partial result for X or Y.

DATA

All input variables to the running program which have more than one element are stored on the magnetic drum prior to execution of the problem. Single valued variables remain in high speed (magnetic core) storage throughout the problem.

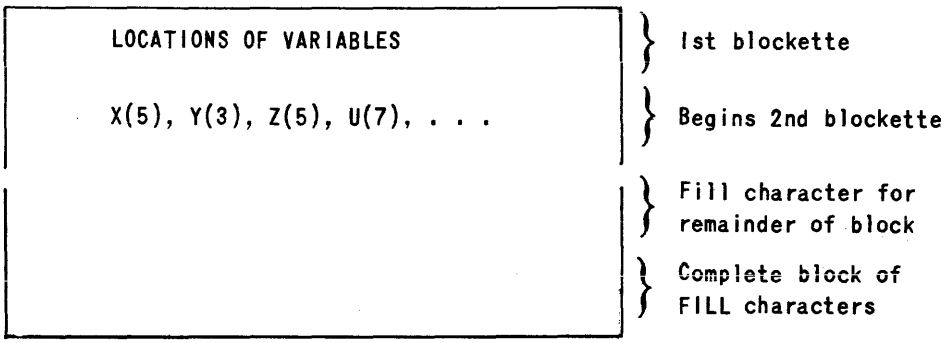
Each segment of the output program has a Preface and a Termination which transfers data between drum and high speed storage during the Interlude between segments (See Appendix C).

DATA PHASE The input data for any problem is read from magnetic tape into high speed storage or onto the drum prior to the execution of the first segment of the output program. To accomplish this the programmer must provide the tape locations of data by placing a directory of information on a specified Uniservo magnetic tape unit.

LOCATION OF DATA The information on this specified tape unit may be considered as a directory of information indicating the tape unit on which the input data is stored. The first blockette of the specified tape will contain the heading:

LOCATIONS OF VARIABLES

Beginning in the second blockette, the variable symbols are written, each followed by a number enclosed in parenthesis which indicates the tape unit containing the values of the variable. Variables are separated by commas. The remainder of the block containing the last item of information and the entire block following are filled out with *FILL* characters.



In the example, the variable X will be found on Uniservo No. 5. This information is used by an automatic read routine which operates prior to the execution of the problem. The arrangement of data on the appropriate tape unit will be described at a later date.

FORMAT

LINE NUMBERING

LINE NUMBERS are used to reference, change, delete, or insert lines in a program. *The programmer must assign positive integral line numbers to those lines which he wishes to refer to in his program.* *UNICODE* will number all lines not assigned a line number by the programmer. If the first line, *UNICODE PROGRAM,* has not been assigned a number, *UNICODE* will number it .1 and continue to number succeeding lines with .2, .3, ..., .9, .10, .11, etc. until the first line number assigned by the programmer is detected. *UNICODE* will add .0 to this line number and continue to assign line numbers in succession from this point, always increasing the number to the right of the right-most decimal point by one. For example, if the first line number detected is 7, *UNICODE* will append to it .0 making it 7.0 and assign succeeding lines the numbers 7.1, 7.2, ..., 7.9, 7.10, etc. until the next line number assigned by the programmer is detected. This process is continued until all lines have been assigned a number.

A line number may be at most twelve characters in length, including decimal points. This means that the integral line numbers assigned by the programmer may be no more than ten characters in length, since the character combination .0 added by *UNICODE* will bring the total to the allowable limit of twelve.

In his corrections to a *UNICODE* program following a translation by *UNICODE,* the programmer may refer to any assigned line number including those assigned by *UNICODE.*

SPACE SYMBOLS

The symbol Λ appearing on the Unityper keyboard is used to denote a space in the *UNICODE* program. In general, the Λ symbol must be used wherever a space would appear in ordinary English word usage, i.e., to separate a word from another word, a word from a variable or constant, a variable from a constant, etc. The space symbol Λ may be used at the option of the programmer wherever its use does not violate any of the following restrictions. The use of more than one space symbol as a separator is also optional. Although the symbol Λ actually appears on the Unityped copy of the input program, only the desired blank

space will appear in the *SIDE-BY-SIDE* listing of the output program. The following restrictions governing the use of the Λ symbol in specific cases ***must be strictly adhered to by the programmer.***

(1) Λ *MUST* be used to separate a standard operation symbol *(SIN, COS, LOG, etc.)* from its operand(s).

Example:

```
X = SINΛY
```

(2) Λ *MUST* be used to separate the exponentiation symbol *POW* from its operands.

Example:

```
Z = XΛPOWΛY
```

NOTE- The exponent always follows the symbol POW; for instance, the mathematical statement of the above equation would be $Z = X^Y$.

(3) Λ *MUST* be included wherever it appears in the list of *UNICODE* instructions presented in Appendix A (*UNICODE LANGUAGE*).

Example:

```
VARYΛXΛp(q)rΛSENTENCEΛKΛTHRUΛMΛ.
IFΛX = Y, JUMPΛTOΛSENTENCEΛK.
```

(4) Λ *MUST* be used together with a period, i.e., in the combination Λ., following the sentence part of each line in a *UNICODE* program.

Example:

```
COMPUTEΛTΛ. (T is temperature of liquid)
```

(5) Λ *MUST NOT* be used as a character in the symbol for a variable, pseudo operation, or standard operation. For example, the character combination MAXΔHT could not be used as the symbol for a variable.

UNICODE PROGRAM

The *UNICODE* (input) program is prepared on standard *UNICODE* programming sheets and then transcribed onto magnetic tape by the Unityper.

The line number (optional) is entered in the area of the programming sheet reserved exclusively for line numbers, the sentence and comment in the area reserved for them. As many lines on the Programming sheet as are necessary may be used to include a complete sentence and its associated comment (optional). The comments, if included, *MUST* be enclosed in parentheses and *MUST* appear immediately following the space-period ∧. combination indicating the end of the sentence. The comment may be as long as desired but *MUST NOT* contain parentheses other than those enclosing the entire comment. Commas, semicolons, and parentheses *MUST* be included where they appear in the listing of *UNICODE* instructions in Appendix A *(UNICODE LANGUAGE)*.

The *UNICODE* program is prepared in accordance with the general forms described in Chapter One and transcribed on magnetic tape. The actual rules and procedures involved in the proper preparation of a *UNICODE* input tape will be explained at a later date, in a supplement intended for use by the Unityper operator.

Each line on the Unityped copy of a *UNICODE* program will contain 120 character positions which represent one blockette on magnetic tape. The first 12 character positions are reserved exclusively for line numbers. The 13th character position contains the space symbol ∧, while the 14th thru 120th character positions are used for sentences and comments. An example of this Unityped input format is shown in Figure 1 of Chapter One.

UNICODE PROGRAM SHEET

PAGE 1

LINE NUMBER 1 12	13	14 SENTENCES AND COMMENTS 120	
		UNICODE∧PROGRAM∧.∧	FIRST BLOCK
		DETERMINATION∧OF∧FORCE∧FOR∧VARYING∧ACCELERATION∧.∧	
		DIMENSION∧F(5)∧.∧	SECOND BLOCK
		F(A) = M * A∧.∧(FORCE = MASS TIMES ACCELERATION WHERE _ _ _)∧	
10		START∧.∧	
11213461111		VARY∧A∧2(2)∧10∧SENTENCE∧1∧THEN∧JUMP∧TO∧3∧.∧	
1		COMPUTE∧F(A)∧.∧	
3		TYPE∧F(A)∧.∧	
		STOP∧.∧	THIRD BLOCK
		END∧OF∧TAPE∧.∧	

Figure 1 - UNITYPED INPUT PROGRAM

SIDE-BY-SIDE LISTING

The information contained in the edited record produced by *UNICODE* is printed out on the High-Speed Printer as a *SIDE-BY-SIDE* listing of the input program and the generated output program. Each *UNICODE* line of the input program is reproduced as it appeared in the Unityped copy, with the addition of the line numbers assigned by *UNICODE*. Immediately following each such line, the corresponding generated machine code is printed, with one machine word per output line.

A blank space is left in the *SIDE-BY-SIDE* listing wherever the symbol Λ appeared in the Unityped input program. A *SIDE-BY-SIDE* listing of the form shown in Figure 2 of Chapter One would be produced as a result of the input program of Figure 1.

UNICODE PROGRAM SHEET

LINE NUMBER		
1 12	13	14 120
.1		UNICODE PROGRAM.
.2		DETERMINATION OF FORCE FOR VARYING ACCELERATION.
.3		DIMENSION F(5)
.4		F(A) = M * A. (FORCE = MASS TIMES ACCELERATION WHERE ...)
		OCTAL MACHINE CODE
		' '
		' '
		OCTAL MACHINE CODE
10.0		START
1121346111.0		VARY A 2(2)10 SENTENCE 1 THEN JUMP TO 3.
		OCTAL MACHINE CODE
		' '
		' '
		OCTAL MACHINE CODE
1.0		COMPUTE F(A).
		OCTAL MACHINE CODE
		' '
		' '
		OCTAL MACHINE CODE
3.0		TYPE F(A).
		OCTAL MACHINE CODE
		' '
		' '
		OCTAL MACHINE CODE
3.1		STOP.
		OCTAL MACHINE CODE
3.2		END OF TAPE.

Figure 2 - SIDE-BY-SIDE LISTING

CORRECTIONS AND TAPE OVERFLOW

Corrections to a *UNICODE* program are made by preparing a Unityped tape which contains the corrected information. This tape is then mounted on a separate Uniservo and the correct information is merged with the original program to produce an updated program tape. The *UNICODE* system incorporates three instructions must have a line number. In the following explanations of the correction instructions the letters k, m, of n will be used to designate a line number.

DELETE

The *DELETE* instruction eliminates entire *UNICODE* lines from the program. In the first example all lines between and including the lines numbered k and m would be eliminated from the program. In the second example only the line numbered k would be eliminated.

Example:

```
(1)  DELETE SENTENCE k THRU m
(2)  DELETE SENTENCE k
```

CHANGE

The *CHANGE* instruction replaces the *SENTENCE* and *COMMENT* parts of a specified line by the correct *SENTENCE* and *COMMENT* (if any), leaving the line number unchanged. The instruction is written in the following form, with the correct *SENTENCE* and *COMMENT* written following the word *TO:*

Example:

```
CHANGE SENTENCE k TO _ _ _ _ _

CHANGE SENTENCE k TO COMPUTE T (Temperature gradient).
```

In the corrected program the line whose number was k would have *COMPUTE T* as its *SENTENCE* and *TEMPERATURE GRADIENT* as its *COMMENT*.

INSERT

The *INSERT* instruction adds new lines at specified points in a *UNICODE* program. The word *INSERT* is written as a complete *SENTENCE* and each line to be added is written as it is to appear in the updated program. If the new line(s) are to be inserted directly following the line whose number is k, then the line number of the first line to be added must be of the form k.i where i is any whole number. The programmer need only number the first line to be added by a given *INSERT* instruction.

Example:

Line Number	Sentence and Comment
(OPTIONAL)	INSERT
6.1.5	COMPUTE X
(OPTIONAL)	REPLACE Y BY X

The example correction would insert the two statements *COMPUTE X* and *REPLACE Y BY X* in the *UNICODE* program directly following the line whose number is 6.1. The line containing the statement *REPLACE Y BY X* will be numbered 6.1.6 by *UNICODE*.

TAPE OVERFLOW

If it is not possible to type an entire *UNICODE* program on a single Uniservo tape, additional tapes may be used. These additional tapes are treated as correction tapes and must be prepared in the form of an *INSERT* correction to the preceding tape; that is, the first line on the additional tape must be an *INSERT* instruction. The second line must be numbered by the programmer as if it were to be inserted directly following the last line on the preceding tape. Also, the programmer must insure that a line number has been assigned to the last line on the preceding tape. The two Uniservo tapes are then merged as in the case of corrections to a *UNICODE* program.

UNICODE--APPENDIX A

UNICODE LANGUAGE

UNICODE INSTRUCTIONS

1. REPLACEΔXΔBYΔAΔ.

2. COMPUTEΔXΔ.

3. X = AΔ.

4. a) VARYΔXΔP(Q)RΔSENTENCEΔNΔTHRUΔMΔTHENΔJUMPΔTOΔKΔ.
   b) VARYΔXΔP(Q)RΔSENTENCEΔNΔ.
   c) VARYΔXΔP,Q,R, ---, S,TΔSENTENCEΔNΔTHRUΔMΔ.
   d) VARYΔXΔP,Q,R, ---, S,TΔSENTENCEΔNΔTHENΔJUMPΔTOΔKΔ.
   e) VARYΔXΔP(Q)RΔWITHΔYΔS(T)UΔSENTENCEΔNΔTHRUΔMΔ.
   f) VARYΔXΔP,Q,R, ---, S,TΔWITHΔYΔR(S)TΔSENTENCEΔKΔ.

5. a) JUMPΔTOΔSENTENCEΔKΔ.
   b) IFΔX = Y, JUMPΔTOΔSENTENCEΔKΔ.
   c) IFΔXΔNOT = Y, JUMPΔTOΔSENTENCEΔKΔ.
   d) IFΔX < Y, JUMPΔTOΔSENTENCEΔKΔ.
   e) IFΔX > Y, JUMPΔTOΔSENTENCEΔKΔ.
   f) IFΔX <= Y, JUMPΔTOΔSENTENCEΔKΔ.
   g) IFΔX >= Y, JUMPΔTOΔSENTENCEΔKΔ.

6. RESUMEΔKΔ.

7. a) WRITEΔX(I,J), ((Title)), TAPEΔLΔ.
   b) WRITEΔF(X,Y,Z,T),X, ---, Z,T,((Title)) (Column Heading) (Col._Hdg) (Col._Hdg) (Col._Hdg) (Col._Hdg), TAPEΔLΔ.

8. TYPEΔ(X(I,J), Y, ---, F(X), Z)Δ.

9. PRINTΔ(--------)Δ.

10. STARTΔ.

11. STOPΔ.

12. ENDΔOFΔTAPEΔ.

13. DIMENSIONΔX(--, --), Y(--, --, --, --), ---, Z(--, --, --)Δ.

SUBPROGRAM INSTRUCTIONS

1. SUBΔPseudo_Operation (X(I), Z(J), -----, Y)Δ.

2. RESULTΔXΔ.

3. EXITΔ.

CORRECTION INSTRUCTIONS

1. DELETEΔSENTENCEΔNΔTHRUΔMΔ.

2. CHANGEΔSENTENCEΔKΔTO ------Δ.

3. INSERT

   } Lines to be inserted written in proper form.

UNITARY OPERATIONS

SYMBOL	OPERATION	EXAMPLE
+ (optional)	plus	+ 157.25
-	minus	157.25
||	absolute value	| X - Y |

BINARY OPERATIONS

SYMBOL	OPERATION	EXAMPLE
+	addition	X+5
-	subtraction	Y-Z
*	multiplication	Y*X*2.5632
/	division	X/7.5
Numerical Superscript	exponentiation (numerical exponent)	Z^5
POW	exponentiation (numerical, literal exponent)	Z POW Y

RELATIONS

SYMBOL	RELATION	EXAMPLE
=	is equal to	Z = X + Y
NOT =	is not equal to	X△NOT = Y
<	is less than	X < Y
>	is greater than	Y > Z
<=	is less than or equal to	X <= Z
>=	is greater than or equal to	Y >= X

UNICODE--APPENDIX B

SAMPLE PROBLEMS

INTEGRATION

The first problem of this section is programmed in four ways:

1. Use of functional notation with equations listed first.
2. No separation of equations but use of functional notation.
3. Use of subscript notation.
4. Elimination of recalculation of previously determined values.

The advantages and disadvantages of these various methods are discussed following the programs.

Problem: Evaluate the following integral by use of Simpson's one-third rule with an interval of one-one hundreth. (Simpson's one-third rule): $(1/3)h\,[y(x) + 4y(x + h) + y(x + 2h)]$; $h = 0.01$.

$$T = \int_0^{\frac{\pi}{2}} \frac{dx}{\sqrt{1 - \frac{\sin^2 X}{2}}}$$

LINE NUMBERS	SENTENCES AND COMMENTS
	UNICODE PROGRAM .
	DIMENSION Y(3) .
	Y(X) = 1/(1 - 0.5 * (SIN X)2)1/2
	U = (1/300) * (Y(X) + 4 * Y(X + 0.01) + Y(X + 0.02)) .
	START .
	T = 0
	VARY X 0(0.02)1.57 SENTENCE 1 THRU 2 .
1	COMPUTE Y(X) AND Y(X + 0.01) AND Y(X + 0.02) AND U .
2	REPLACE T BY T + U .
	TYPE (T) .
	STOP .
	END OF TAPE .

Figure 1 - FUNCTIONAL NOTATION, EQUATIONS WRITTEN FIRST (FIRST METHOD)

LINE NUMBERS	SENTENCES AND COMMENTS
	UNICODE PROGRAM .
	DIMENSION Y(3) .
	START .
	T = 0.
	VARY A 0(0.02)1.57 WITH B 0.02(0.02)1.57 SENTENCE 1 THRU 2. .
1	VARY X A(0.01)B SENTENCE 3 .
3	Y(X) = 1/(1 - 0.5 * (SIN X)^2)$^{1/2}$.
	U = (1/300) * (Y(X) + 4 * Y(X + 0.01) + Y(X + 0.02)) .
2	REPLACE T BY T + U .
	TYPE (T) .
	STOP .
	END OF TAPE .

Figure 2 - FUNCTIONAL NOTATION, EQUATIONS APPEARING AS STATEMENTS (SECOND METHOD)

LINE NUMBERS	SENTENCES AND COMMENTS
	UNICODE PROGRAM .
	DIMENSION Y(3) .
	Y(I) = 1/(1 - 0.5 * (SIN (X + Z))^2)$^{1/2}$.
	U = (1/300)*(Y(1) + 4 * Y(2) + Y(3)) .
	START .
	T = 0 .
	VARY X 0(0.02)1.57 SENTENCE 1 THRU 3 .
1	VARY I 1(1)3 WITH Z 0(0.01)0.02 SENTENCE 2 .
2	COMPUTE Y(I) .
	COMPUTE U .
3	REPLACE T BY T + U .
	TYPE (T) .
	STOP .
	END OF TAPE .

Figure 3 - SUBSCRIPT NOTATION, EQUATIONS FIRST (THIRD METHOD)

LINE NUMBERS	SENTENCES AND COMMENTS
	UNICODE PROGRAM
	DIMENSION Y(3)
	Y(I) = 1/(1 - 0.5 * (SIN (X + Z))^{2})$^{1/2}$.
	U = (1/300) * (Y(1) + 4 * Y(2) + Y(3)) .
	START .
	T = 0 .
	Y(1) = 1.
	VARY X 0(0.02)1.57 SENTENCE 1 THRU 3.
1	VARY I 2(1)3 WITH Z 0.01(0.01)0.02 SENTENCE 2.
2	COMPUTE Y(I).
	COMPUTE U.
	REPLACE T BY T + U.
3	REPLACE Y(1) BY Y(3).
	TYPE (T).
	STOP.
	END OF TAPE.

Figure 4 - SUBSCRIPT NOTATION, EQUATIONS FIRST (FOURTH METHOD)

Examination of the four methods shows that Method 1 (Figure 1) is the most straightforward. It requires the least thought on the programmer's part and does not employ any clever techniques.

Method 2 (Figure 2) must have two *VARY* instructions if the rewriting of the equation for Y is to be avoided, and thus requires more thought than Method 1. It does not, however, require writing a *COMPUTE* instruction.

Method 3 (Figure 3) uses subscript notation and, in order to obtain the proper arguments for the *SIN* function, requires two *VARY* instructions and considerable thought. This method will produce a better object program, however, due to the use of subscripts.

Method 4 (Figure 4) also uses subscript notation and two *VARY* statements, but eliminates the calculation of Y(1) within the *VARY* loops. However, this method requires the knowledge that Y(1) has a starting value equal to 1.

GENERATION OF TABLE INTEGRALS

Whereas the preceding programs compute only one value of the elliptic integral shown, the following program goes a step further and computes an entire table of values.

Problem: Compute a table of values of the elliptic integral

$$F(S, Z) = \int_0^Z \frac{dx}{\sqrt{1 - S^2 \text{ SIN } X}}$$

for $0 \leq Z \leq 90^0$, $0 \leq \arcsin S < 90^0$.

LINE NUMBERS	SENTENCES AND COMMENTS
	UNICODE PROGRAM .
	Y(S,X) = 1/(1 - S² * SIN X)^1/2 .
	U = (1/3000)* (Y(S,X) + 4 * Y(S,X + 0.001) + Y(S,X + 0.002)) .
	S = SIN A
	V = 57.2957795 * A .
	W = 57.2957795 * Z .
	DIMENSIONS Y(1,3) .
	START
	VARY A 0.0872665 (0.0872665)1.570797 SENTENCE 1 THRU 2 .
1	VARY Z 0.0174533 (0.0174533)1.570797 SENTENCE 3 THRU 2 .
3	F = 0 .
	VARY X 0(0.002)Z SENTENCE 5 THRU 4 .
5	COMPUTE S AND Y(S,X) AND Y(S,X + 0.001) AND Y(S,X + 0.002) AND U
4	REPLACE F BY F + U
	COMPUTE V AND W .
2	WRITE F, V, W, ((ELLIPTIC INTEGRALS OF THE FIRST KIND)) (INTEGRAL) (ARCSIN S) (UPPER LIMIT).
	STOP .
	END OF TAPE .

Figure 5 - PROGRAM TO GENERATE A TABLE OF INTEGRALS

In the above program:

0.0174533 radian = 1 degree

1.570797 radian = 90 degrees

Since only 3 values of Y are needed at one time, and only at this time, the dimensions on Y can be kept to a minimum. The table generated will appear as follows (V varying in 5-degree steps and W in 1-degree steps; for each value of V there will be 90 values of W):

ELLIPTIC INTEGRALS OF THE FIRST KIND

F	V	W
INTEGRAL	ARCSIN S	UPPER LIMIT
1.75000000(-2)	5.0000000	1.0000000
3.49000000(-2)	5.00000000	2.00000000
5.24000000(-2)	5.00000000	3.00000000
'	'	'
'	'	'
'	'	'
'	'	'
etc.	etc.	etc.

V is incremented in 5-degree steps and W in 1-degree steps. W goes through 90^{o} for every 5-degree step of V.

DIFFERENTIAL EQUATION

Given the differential equation

$$\frac{dy}{dx} = x^3 - 3xy^3 + 2,\ y(0) = 0$$

find its solution in the interval [0,3] by use of the Runge-Kutta method with an interval H between successive values of x.

LINE NUMBERS	SENTENCES AND COMMENTS
	UNICODE PROGRAM .
	D(X,Y) = X^3 - 3 * X * Y^3 + 2 .
	R1 = D(X,Y) * H .
	R2 = D(X+H/2, Y+R1/2) * H .
	R3 = D(X+H/2, Y+R2/2) * H .
	R4 = D(X+H, Y+R3) * H .
	DY = 1/6 * (R1+2 * R2+2 * R3 + R4) .
	DIMENSION D(1,1) .
	START .
	Y = 0 . (SET INITIAL CONDITION)
	H = 0.01 .
	VARY X 0(H)3 SENTENCE 1 THRU 2 .
1	WRITE Y, X .
	COMPUTE D(X,Y) AND R1 AND D(X+H/2, Y+R1/2) AND R2 AND D(X+H/2, Y+R2/2) AND R3 AND D(X+H, Y+R3) AND R4 AND DY .
2	REPLACE Y BY Y + DY .
	STOP .
	END OF TAPE .

Figure 6 - PROGRAM TO SOLVE A DIFFERENTIAL EQUATION

Instead of using the letter H, its value, in this case 0.01 could be written in the equations and *VARY* sentence. The method shown, however, facilitates the alteration of H if required.

Note the stated dimensions of D. Since only one value of D is needed at a time, each new value of D can replace the previous value. This reduction of storage allotted to the values of D is possible only through the *DIMENSION* statement, and thus through program control. Also, the programmer need not be concerned about destroying the value of X as defined by the *VARY* since actually two locations are kept for X; one of which will be changed only by the vary, and one "working" location stores the value of expressions used as arguments. For example, when computing

```
D(X = H/2, Y+R1/2)
```

the value of X as defined by the vary is added to H/2 and the result is stored in the working location for X, the equation for D referring to the working location of the value of X. Y is handled similarly.

In this problem the *WRITE* sentence is placed following the *VARY* so that corresponding values of X and Y will be printed in two adjacent columns.

UNICODE--APPENDIX C

DETAILS AND TECHNIQUES OF COMPILATION

At the *START* of the Allocator Phase the Uniservos hold information as follows:

Uniservo 1 *UNICODE*
2 Op (Operation) File I of library followed by relatively coded library routines
3 Source program
4 Patch tape of corrections to source program (if any)
5 Op File I of generated routines; List I; relatively coded generated routines
6 Blank
7 Blank
8 Blank

At *COMPLETION* of the Allocator and Processor phases, Uniservos 1 through 5 hold the same information and

Uniservo 6 Op File III
7 Object program tape
8 Edited record (side by side listing prepared in format ready for High-Speed Printer)

All tapes for Univac Automatic Coding program will have the following format:

Block 1 1 - 20 words 21 - 22 23 - 120	all Z's (1st blockette) Tape label all Z's
Block 2 1 - 2 words 3 - 120	Entry label designating type of information to follow directly in block 3 all Z's
Block 3 1 - m words	Indicative information (start) . .
Block n	. . . . . last word of indicative information. fill with Z's to end of block
Block n+1 1 word 2 3 - 120	END△OF End of Entry Lable △ ENTRY all Z's

Repetition of Block 2 through Block n+1 (as necessary)
2 Final Blocks written all Z's

LIST 1 FORMAT

List 1 consists of the call words, one per location in the *u* position of all the selected *library routines* needed for solution of a given problem. This list is generated during translation and stored following Op File I of the generated routines on Uniservo 5. If no library routines are required, only the lead and final sentinels appear on Uniservo 5 with no indicative information of List 1 intervening.

The count of the total length of List 1 is generated and placed in *u* position of the location directly following the word containing List Δ1 (Entry label) on the tape. This is done during translation.

Block A	1st word 2nd 3-120	LIST Δ 1 length all Z's
	op	u v
Block A + 1	1 2 3	call word call word call word

(fill remainder of unfilled block with Z's)

OP FILE 1 FORMAT

Word	Position	
1	u	call word
1	v	number of lines this item in this Op File
2	v	total number of lines for this routine in running program including temporaries
3 3+j	u u	call word for cross reference (if any) one per location

Hence the length of the Op File for each item is (2+n) where n is the number of cross references.

CALL WORDS

Call Words are represented by 5 octal digits; the first always being a 7. The second digit categorizes the type of routine; the third through fifth digits, the assigned routine number.

	70xxx	Statements
	71xxx	Equations
	72xxx	Pseudo operations
	73xxx	Library routines
	74xxx	Statements of a pseudo operation
	75xxx	Vary statement - main program only
Unsubscripted (any single-valued variable)	76xxx	Output from equation subroutines (generated intermediate results - 1 output per equation).
Subscripted variables	77xxx	Multiple word data groups

LIBRARY AND GENERATED SUBROUTINE FORMAT

	1 u	Call word (unique for each routine)
	1 v	Number of lines of prelude and routine
Do not appear in running program	2	Number of lines subject to address modification
	3	Number of unmodifiable constants
	4	Number of temporary storage required
	5	Number of inputs
Prelude	6	Number of outputs
	7 8	} Name of routine 12 characters
	9	MJ to START
	10	RJ Diagnostic
	11	MJ EXIT
Routine in Standard USE Format	. . . .	Output lines as required Input lines as required 1st instruction in main body . . . instructions . . Relative constants (if any) Unmodifiable constants (if any)

Block	Word		
1	1- 20	all Z's	leading sentinel
	21	ΔΔLIBΔ	
	22	ΔTAPEΔ	
	23-120	all Z's	
2	1	ΔΔΔOPΔ	
	2	FILEΔ1	
	3	number of words in entire library Op File 1	
	4-120	all Z's	
3	1	1st word of indicative information library Op File 1	
n-1 block	1	indicative information (continued) fill remainder of block with Z's	
nth block	1	ENDΔOF	
	2	ΔENTRY	
	3-120	all Z's	
n+1st block	1	ΔΔLIBΔ	
	2	SUBRTN	
	3-120	all Z's	
n+2nd block	1	Library prelude . . Library routine . . .	(1st routine) Each prelude must start in a new block
n+mth block	1	Library prelude (last routine) . . .	
n+m+1 block	1	ENDΔOF	
	2	ΔENTRY	
	3-120	all Z's	
n+m+2 block		all Z's	final sentinel
n+m+3 block		all Z's	

Block	Word		
1	1- 20	all Z's	leading sentinel
	21	ΔΔGENΔ	
	22	ΔTAPEΔ	
	23-120	all Z's	
2	1	ΔΔΔOPΔ	
	2	FILEΔ1	
	3	Number of words in subroutine Op File 1	
	4	Number of lines of data 76-----type	
	5-120	all Z's	
3	1	1st word of indicative information	
	.	. subroutine Op File 1	
	.	.	
	.	.	
	.	.	
	.		
n-1 block	1	indicative information Op File 1 (continued)	
	.		
	.		
	.		
	m	last word of indicative information	
	m through 120	all Z's	

Note:

if m = 119 Z's appear in 120 only
is m = 120 no Z's in n-1 block
nth block as follows:

1	ENDΔOF
2	ΔENTRY
3-120	all Z's

n	1	ENDΔOF	
	2	ΔENTRY	
	3-120	all Z's	
n+1st block	1	LISTΔ1	
	2-120	all Z's	
n+2nd block	1	indicative information List 1 (if any)	
	.		
	.		
	.		

r-1 block	1	continuation of List 1
	s	last entry in List 1
	s-120	all Z's
rth block	1	END△OF
	2	△ENTRY
	3-120	all Z's
r+1st block	1	△SUBRO
	2	UTINES
	3-120	all Z's
r+2nd block	1	start of prelude for first subroutine
	.	
	.	
	.	
	.	
	.	
	.	
	.	
	.	
	.	
	.	
r+kth block	1	Prelude (last subroutine)
		Subroutine (last)
r+k+1st		
	1	END△OF
	2	△ENTRY
	3-120	all Z's
r+k+2		all Z's
r+k+3		all Z's

Each prelude must start first subroutine in a new block; remainder of block filled with Z's.

2 full blocks of Z's as final sentinel

Directory 1 - Locates Op File I on magnetic drum.
2 word item - an item for each routine preceded by length of Directory 1
1 u - call word
2 v - magnetic drum address of location of first word of Op File I

length
1 u
2 v
1 u
2 v
etc.

Directory 2
2 words only
1 u - magnetic drum address of first statement Op File I
2 u - magnetic drum address of the Op File I of the item following last statement Op File I

Directory 3 - 1 word for each segment
1 u - magnetic drum location for start of Op File III for this segment
1 v - number of words in Op File III for each segment

Directory 4 - A list of flagged jumps to *other* segments only
1 u - call word
word 2 First and second octal digits (operating position) - IP flag recorded during Phase III
Third octal digit - blank
Fourth and fifth octal digits - segment number jumping from
Sixth and seventh octal digits - segment number jumping to
Eighth - twelfth (v position) - high speed storage running in new segment

Directory 5 - magnetic drum assignments of data - 2 word items
1 u - call word
2 v - magnetic drum address assigned

List 1
- Call words of the needed library routines for this specific problem

Op File I - 1 u - call word
1 v - number of lines this item this Op File
2 v - total number of lines in subroutine running program
3 u
⋮
3+j cross reference call words, one per location (if any)

Op File IIa - Segment description (Intermediate Allocator results)

Op File IIa-6
-5 } - Reserved for ease in transforming Op File IIa to Op File III

-4 - Total number lines in statements and routines + 1 (in v)

-3 - Address for insertion of IP for this segment

-2 - Call word for first statement of next segment

-1 - IP for natural connection to next segment

Op File IIa - Start of explicit information

1 u call word
1 v
} 2 word items

1 u
1 v
}

Op File IIb - List of all jumps

1 word items { 1 u - call word } grouped by segment

Op File III

Op File III-6 - Segment number

-5 - Length of Op File III (in u, no j)

-4 - Total number lines in statements and routines + 1

-3 - Address for insertion of Ip command

-2 - Call word for first statement of next segment

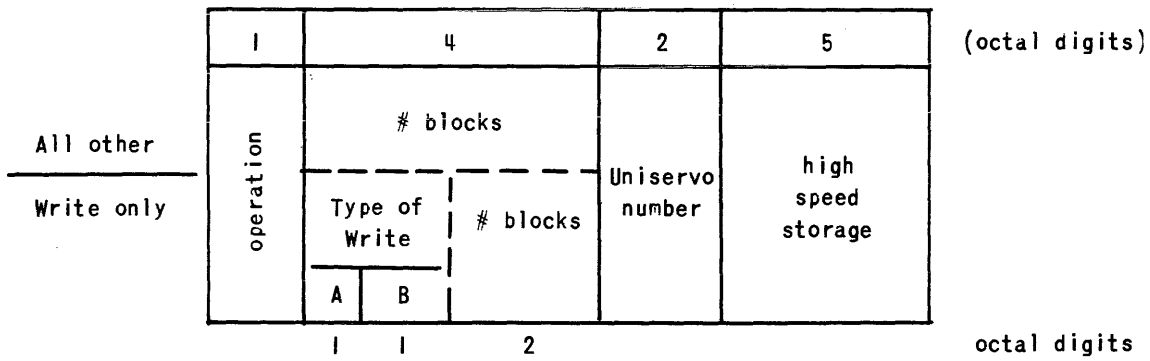
-1 - IP for natural connection

Op File III - 1 u - call word
2 v - high speed storage running address
2 op - flag for cross-reference jumps (if any)
2 u - if data: number of lines of data
if flag: segment number jumped to and
segment number jumped from

} Start of explicit information

} Written in Phase III

CODE WORD FOR GENERALIZED TAPE HANDLER



Octal Digit A specifies blockette spacing

A = 1 - 0 inches blockette space
2 - 0.1 inches blockette space
4 - 1.0 inches blockette space

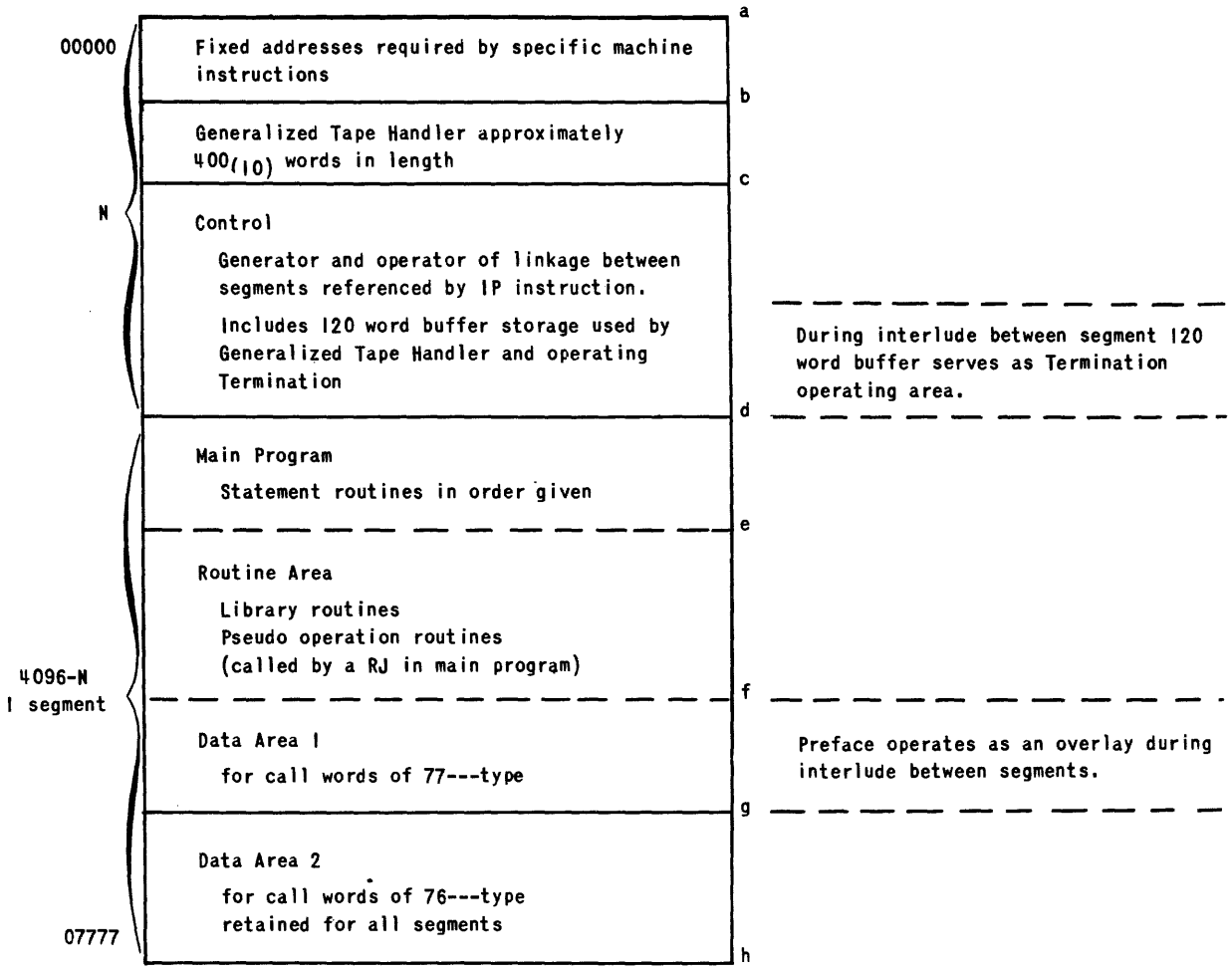
Octal Digit B specifies block spacing and density

B = 0 - 128 line/inch density, 1 inch block space
1 - 50 line/inch density, 1 inch block space
2 - 128 line/inch density, 2.4 inch block space
3 - 50 line/inch density, 2.4 inch block space

Operation Code

0 No operation - ERROR
1 Rewind
2 Rewind with inter-lock
3 Move Forward
4 Move Backward
5 Read Forward
6 Read Backward
7 Write

FIGURE I - High Speed Storage Memory Layout During Running of Object Program (illustrated for Univac Scientific (Model 1103A) with one bank of magnetic core storage)



a - b	Fixed length all problems
b - c	Fixed length all problems
c - d	Fixed length all problems
d - e	Variable length as determined by Allocator
e - f	Variable length as determined by Allocator
f - g	Variable length as determined by Allocator
g - h	Variable for each different problem; but once set by Allocator is of fixed length for all segments of the same problem
N	Fixed length a through d for all problems

CONTROL SECTION DURING RUNNING PROBLEM

The Control section of coding is called into operation whenever an IP command is encountered during the running problem. The Control section, which resides and operates in high-speed storage throughout the running problem, has the ability to recover the IP instruction and extract its coded information from bit positions 0 - 29. This information (segment from, segment to, and high-speed storage of segment to) is supplied to a generator which produces necessary coding for:

1. Calling in the Termination routine for the appropriate segment (to 120 buffer)
2. Executing this Termination routine from 120 buffer
3. Returning to the Control section for the appropriate tape movement to obtain the needed segment and its Preface.
4. Reading in the segment and Preface (as a continuous unit)
5. Executing the Preface
6. Transferring control to desired high-speed storage in the new segment

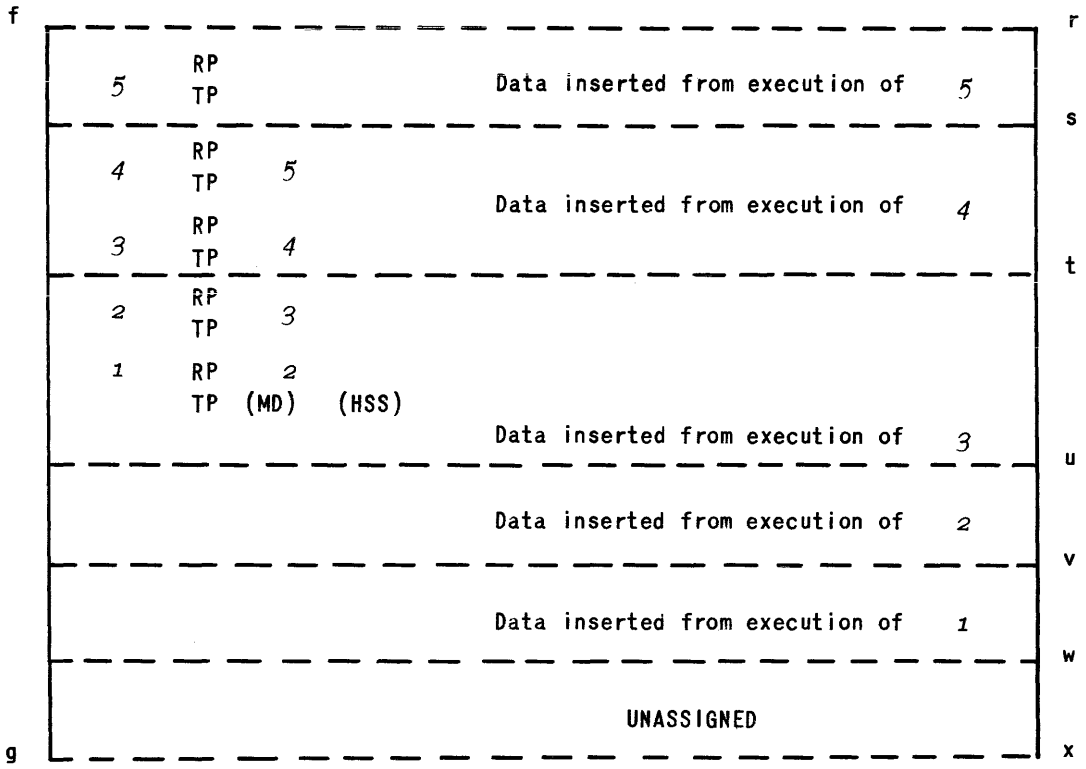
THE INTERLUDE BETWEEN SEGMENTS

The Interlude is divided into 3 natural phases--all under direction of the Control section. The first phase reads the appropriate Termination coding from tape into the 120 buffer storage and execute the Termination instructions. The second phase reads the instructions for the next segment to be executed and its Preface instructions from tape into high speed storage. The third phase executes the Preface instructions from the Data Area 1 where they are located. Their execution results in an overlay of themselves, filling Data Area 1 with the new information needed for the segment to be executed.

The Preface is read into memory directly following the Routine area in ascending address locations. The last pair of Repeat Transmit (75-11) orders will be executed first. Their execution will transfer data from the magnetic drum to locations in high speed storage preceding Data Area 2. If the segment length is exactly 4096-N there will be no unused memory locations during running and hence Data Area 1 will be firmly packed between the Routine area and Data Area 2. If the segment length is less than 4096-N, which will be the usual case, the unused portion of memory will be found immediately preceding Data Area 2. The Repeat Transmit instructions, which operate during the interlude between segments, are executed in reverse; that is from decreasing absolute address locations. By use of the Repeat Transmit, the first *n* data words are inserted in successively increasing memory addresses starting at location [Data area 2 - (n + unused storage)]. Thus no preface instructions can be overlaid before they have been executed. The items of Data Area 1, 77 ----type data, are always at least two words in length where as the Repeat Transmit for each data group is exactly two in length.

Preface and Termination routines are called from tape whenever an IP instruction is encountered. The Termination is called first (under direction of the Control section) and is executed from the 120 word buffer available within the Control section. It is not an overlay. When all Termination instructions have been executed, machine control is returned to the Control section of coding which reads in the appropriate portion of tape containing Segment i and Preface Segment i. (See Object Program Format, Figure III) The Preface is an overlay of Data Area 1 and is transferred into high speed storage with the segment instructions for which it is designed. The Preface is then executed and machine control returned to the appropriate instruction of the segment now residing in high speed storage.

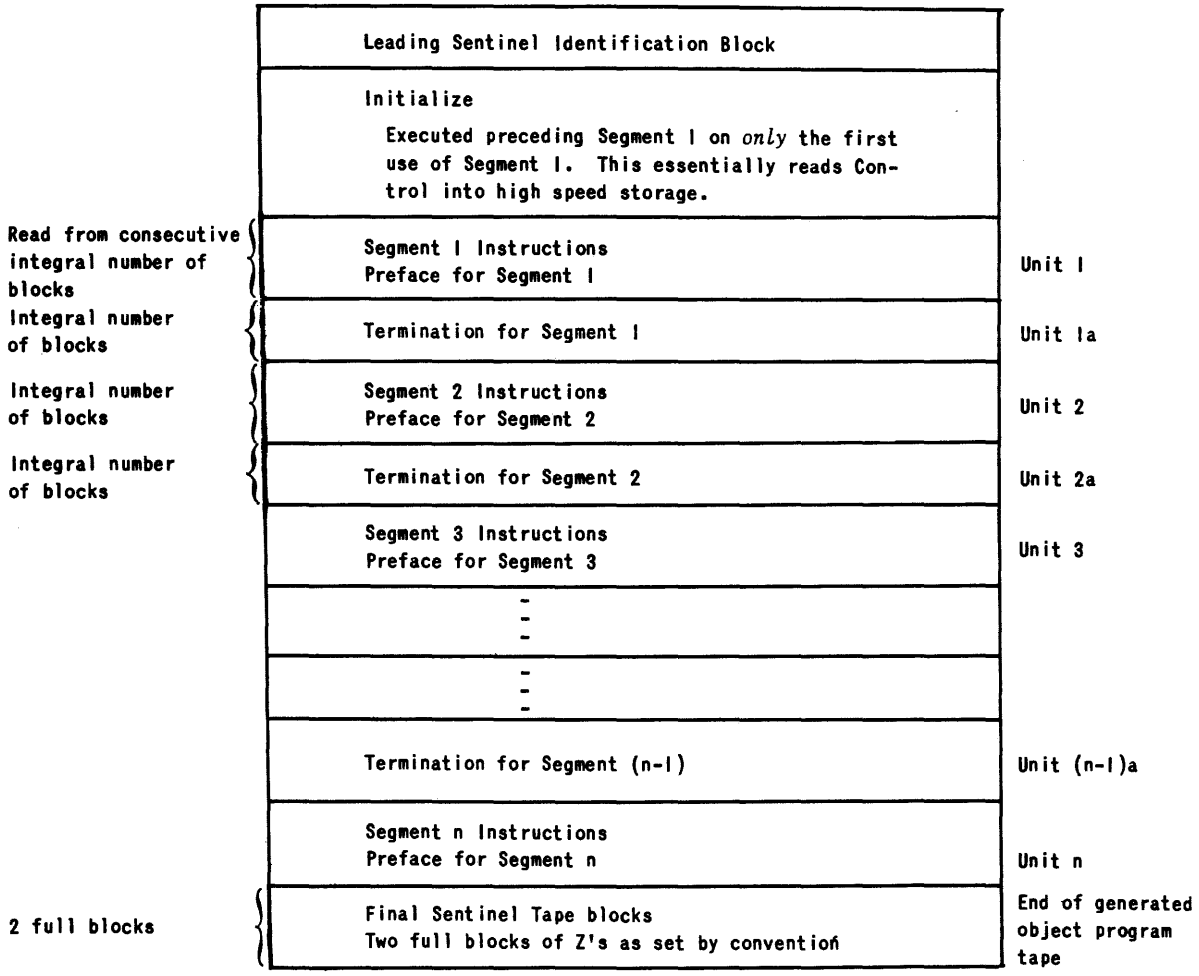
FIGURE II - Preface Layout During Running of Object Program
Kenlargement of f through g of Figure I)



The storage area shown in Figure II has been reserved by the Allocator for the data required (five 77---type groups) of an illustrative segment. The execution of Preface Instruction pair 1 inserts data as shown from v to w. Similarly pair 2 from u to v; pair 3 from t to u; pair 4 from s to t; pair 5 from r to s.

Note that the operation of any pair cannot overlay any instructions not yet executed. When all pairs have been executed, the data is firmly packed starting at position r. The area w thru x represents unassigned storage for this particular segment as previously determined by the Allocator.

FIGURE III - Object Program Format as Stored on Uniservo



UNICODE--APPENDIX D
ALLOCATOR AND PROCESSOR DESCRIPTION

The main purpose of the Allocator is to determine the most effective segment lengths into which the object program is to be divided. When segment lengths have been determined, instructions for orderly handling of these coding groups are generated. The Processor then appropriately modifies the stretches of coding, collects and arranges them systematically into a meaningful object program tape.

Input to Allocator
- Uniservo No. 2: Complete Op File I for library routines
- Uniservo No. 5: Op File I for generated routines; List 1

Output from Allocator
- Uniservo No. 6: Op File III by segment

Input to Processor
- Uniservo No. 2: Library routines (coded relative to 01000)
- Uniservo No. 5: Generated routines (coded relative to 01000)
- Uniservo No. 6: Op File III by segment

Output from Processor
- Uniservo No. 7: Object program tape
- Uniservo No. 8: Edited record for High-Speed Printer

PHASE I. (ALLOCATOR)

Phase I prepares two directories using Op File I of the generated routines on Uniservo No. 5 and Op File I of the library routines on Uniservo No. 2. All items of Op File I on the generated routine tape are first read into high speed storage and then transferred to the magnetic drum. Directory I is then constructed by making an entry for each item placed on the magnetic drum. The first word of this entry contains the call word for this item in the *u* position; the second contains the locating magnetic drum address for this item in the *y* position.

When Op File I of the generated routine tape has been completely read into high speed storage, List 1 (a listing of all library routines required for the problem prepared during translation) is read into high-speed storage.

(List 1 is stored following Op File I of the generated routine tape.) Next Op File I of the library tape is read from tape and checked for the occurrence of the items of List 1. When an item of List 1 is found in the library Op File I, the Op File for this item is placed on the magnetic drum and an entry is made in Directory 1.

Directory 2 consists of only two words. The first word holds the magnetic drum address of the first statement Op File; the second contains information relating to the magnetic drum address of the last statement Op File. This two-word directory is prepared concurrently with Directory 1.

PHASE II (ALLOCATOR)

Phase II uses Directories 1 and 2 to divide the problem into efficient running segments, producing Op File IIa and IIb on tape for each segment.

Using the first word of Directory 2 (location of Op File I for the first statement) as the initial point, Op File I for each statement is processed in sequence. A sub-tally of the total number of lines of coding required for each current statement and its necessary cross references is maintained. This in turn updates a master tally for the segment containing the accumulated total number of lines needed for all statements and their required cross reference routines. After processing each complete statement, the master tally is checked to determine if it is within the prescribed limits (4096 - N; where N is the length of the Control section). If it has exceeded the set limits, the sub-tally is subtracted from the master tally and this becomes the length of the segment. If a single statement and its necessary cross references exceeds 4096 - N further processing is required. This has not been included at the present stage of development of *UNICODE*.

The last statement processed which exceeds the set limit, becomes the first entry in the following segment. Processing continues entering each item in turn onto Op File IIa, using the length 4096 - N as a limit for each segment. Whenever cross references to other statements (open jumps) are recognized, these call words are entered into Op File IIb. Thus Op File IIb is a listing of jump cross reference call words for each segment. When sufficient statements for one segment have been processed and their call words entered into Op File IIa and IIb (as needed), these files are written on tape ready for use in Phase III. The process is repeated, building Op File IIa and IIb for each segment, using the second word of Directory 2 to indicate when the last statement in Op File I has been processed.

PHASE III (ALLOCATOR)

Purpose:

1. Builds Op File III for each segment and writes on tape.

OP FILE III (2-word items)

	op	u	v
1.		call word	
2.			high speed storage running location
		or	
1.		call word	high speed storage running location in another segment (if the end point of one-way jump is in another segment)
2.		segment number from / segment number jumped to	
		or	
1.		call word	high speed storage running location (if call word is of the form 77xxx which refers to a group of data, e.g. $x_1, x_2, \ldots x_{10}$. or if call word is of the form 76xxx, which refers to unsubscripted data, e.g. as x, y, etc.)
2.		number of data	

2. Generates the necessary instructions to manipulate data between segments during the running program. These instructions are called:

   a) The Preface - transfers 77xxx type data to its running location in high speed storage.

   b) The Termination - transfers up-dated 77xxx type data to its designated locations on magnetic drum.

   The Preface and Termination instructions operate in high speed storage during the interlude between two segments. After generation of these instructions (but during this phase) the Preface and Termination are put on the drum.

Input:

Phase III receives as input (from Phase II):

1. Op File IIa - call words of routines and data in segment.

2. Op File IIb - call words of end points of *all* one way jumps.

These files are on Uniservo tape by segment.

Output:

From Phase III then, its output consists of:

1. Op File III by segments on tape.

2. Preface and Termination for each segment on drum.

Procedure:

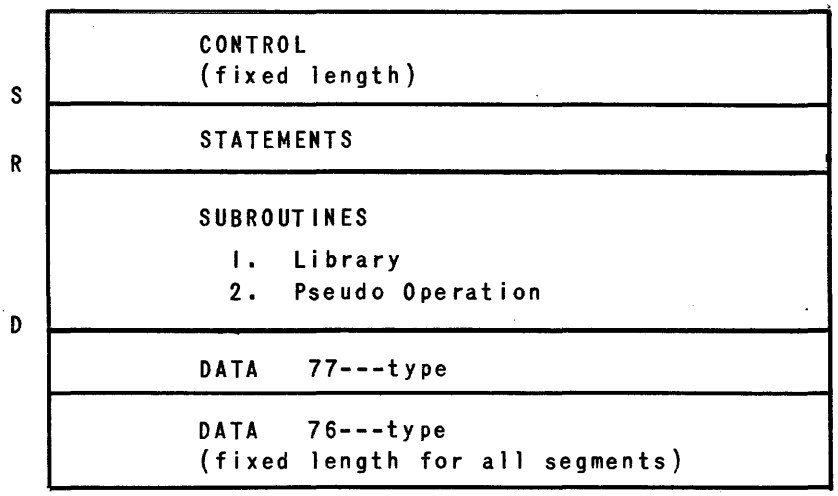
Read Op Files IIa and IIb into high speed storage one segment at a time. Then compare each call word in Op File IIb against the entire Op File IIa for this segment to determine if the end of the jumps (which are actually the words in IIb) appear in the same segment. If equality is not met, the call word from IIb is entered in IIa, thus increasing the length of Op File IIa. Each new entry into IIa at this time is accompanied with the flag 14 in the operation position of the next word. Thus, each new entry in IIa is an entry of two words. Each time an entry is made in Op File IIa the call word from IIb is also placed in another list, called Directory 4, which will be used only during this phase. Each entry in Directory 4 is also a 2-word entry, consisting of the call word in the first word and the segment number in the second word. An item in Directory 4 at this time looks like this:

	on	u		v
1st word	00	call word		00000
2nd word	00	segment 0 number	00	00000

The above procedure is followed until all the call words in Op File IIb have been checked against Op File IIa for one segment. Each call word in Op File IIa is checked to determine the type of routine or data to which it refers:

70xxx	statement
71xxx	equation
72xxx	pseudo operation
73xxx	library routine
74xxx	statement of a pseudo operation
75xxx	vary statement (main program only)
76xxx	single word data items
77xxx	multiple word data groups

The determination of the type of routines used in the segment, along with the number of lines in the running routine (available in Op File IIa), enable assignment now of the actual operating addresses according to the high speed storage layout:



Because Control is fixed in length, *S* can be exactly located. A separate tally of statement lengths during Phase II permits exact determination of *R*. The accumulated tally of total statement and subroutine lengths plus one determines *D*. (The plus one accounts for the single generated instruction needed to provide continuity between sequential segments.) With these starting points *S*, *R*, and *D*, assignment of memory locations in a forward direction can be made according to the category determined by the call word.

The number of lines of data, when the call word is 76--- or 77---, is also used to fill in the *u* portion of data items in Op File IIa. At this time, Op File IIa is beginning to resemble the new Op File III which is actually an expanded and completed Op File IIa.

Upon completion of the foregoing process for each segment, that segment's Op File III (formerly Op File IIa) is written on the drum and Directory 3, containing one word for each segment, is constructed in the following format:

op	u	v
	MD location of 1st word of Op File III	# of words in Op File III for this segment

Thus the first word in Directory 3 refers to the first segment, the second word, the second segment, etc.

When Op File III for the last segment has been written on the drum, Op File III is in its final form for all items except those referring to jumps to other segments. Because Directory 4 is actually a combined listing of these call words for all segments, the items of Directory 4 are used to search against Op File III (by segment) and fill in Directory 4 with the segment number where the call word is found and the operating address of the routine during execution. This continues until all the entries in Directory 4 have been processed. A complete Directory 4 item is of the form:

op	u		v
	call word		
14	segment from	segment to	high speed storage running address in segment to

The second word of the above item in Directory 4 is filled into Op File III (one segment at a time) in its appropriate place to complete Op File III. While each segment is in high-speed storage at this time, the instructions for data manipulation are generated and stored on the drum.

The instructions for data manipulation are prepared from Op File III. Each multiple word data group is assigned an area on the magnetic drum starting with 40001. When the call word for a data group (77xxx type) has been assigned, a magnetic drum location entry into Directory 5, the listing of MD data assignments, is made. Using Op File III information for each 77xxx type call word, the repeated TP's are then generated. When this listing is complete, the *w's* of Repeat orders are determined and recorded, in reverse direction for Preface and forward for Termination. The *w's* for the Preface are fixed, relative high speed storage running locations because they are generated at a point during compilation when the exact starting address of Data Area 1 (77 - - - type) is known. Since the length of the Preface is known when Termination *w's* are written, they also are assigned fixed addresses in the 120 buffer area within the Control section. The completed Preface and Termination instructions for

each segment are stored on the magnetic drum and are available there during the Processor phase.

Generation of instructions to read the Uniservo containing input data to the appropriate magnetic drum regions via high speed storage buffer is also prepared during Phase III. These instructions are written on tape and operate from high speed storage preceding the first read-in of the running program. A list of all data symbols, along with their 77 - - - type call word constructed when call words were assigned during a previous phase, is utilized for the generation of these instructions.

This phase is complete when Op File III by segments has been written on tape.

PHASE IV (PROCESSOR)

The Processor assembles the required subroutines for each segment, using as input the Op File III for each segment together with the library and generated subroutines with their preludes. As each subroutine is processed the relatively coded addresses are changed to the proper machine coded operating addresses. Cross reference call words are replaced by the necessary machine coding to accomplish the cross reference, depending on whether the reference is "within a segment" or "from one segment to another." When all the routines for one segment have been processed, the segment together with its Preface and Termination is transferred to Uniservo tape. This tape, containing all the segments in sequence, is the Object Program tape. Following is a more detailed description of the methods used in modifying the relative coding.

In the initial stage of the Processor the Op File III for the segment to be processed is read from tape into high speed storage. When this transfer has been completed, the first subroutine is read from the Generated tape into the tape image in high speed storage. Tape handling is then temporarily suspended and the actual processing begun. The call word for the subroutine is checked against those listed in Op File III to determine if the subroutine is referenced in this particular segment; the word following the call word to determine if it has a flag indicating a cross reference to another segment. If the call word is listed in the Op File III and is not flagged, the subroutine will then be processed. If the subroutine is not to be processed at this time, the next subroutine will be read into the tape image and the procedure repeated.

The first line to be processed in all cases is the entrance line of the subroutine. Following the modification of this line, each line subject to address

modification is processed in order, beginning with the line indicated by the line count of the tape image. Each relative address is processed, depending on the nature of the coding, to obtain the proper machine coded address.

Those addresses coded relative to 01000 are modified by changing their reference from 01000 to the high speed storage operating location of the subroutine in which they appear. Those addresses which reference another subroutine contain a call word having an octal 7 digit in the leftmost octal position of the address. With one exception such an address is modified by replacing the call word by the high speed storage running address of the referenced subroutine. This running address is obtained from the word following the call word in OP File III. The one exception to this method of modifying a cross reference is that in which the cross reference is to another segment. A reference to another segment occurs as a one way unconditional jump and is modified by replacing the entire line of coding by an Interpret instruction which furnishes the Control section with the information necessary to accomplish the desired cross reference. This Interpret instruction is obtained from the word following the call word in Op File III, and contains the segment number from which the jump is made, the segment number to which the jump is made, and the high speed storage running address in the latter segment. When a reference is made to a line of another subroutine other than the entrance line, the line to be modified contains the call word of the referenced subroutine. This line is followed by a special line of coding containing the location of the referenced line relative to the entrance line of the subroutine. In a reference of this type, the address is modified as is any other cross reference within the same segment to obtain the running address of the subroutine. The special line of coding is then applied to produce the running address of the referenced line within the subroutine.

The lines which have been modified are transferred in groups to successive locations in the drum image until all the words subject to address modification have been processed. The constants for this routine are then transferred to consecutive locations following the last modified line in the drum image, and the drum image address is advanced to allow for the temporary storage locations required.

Each generated subroutine and library routine required for the particular segment is processed in this manner. When all the required routines for a segment have been assembled and processed, the entire drum image load with the proper Preface and Termination is transferred to the Output tape to form a segment of the final running program. The Library and Generated tapes are then rewound and the processing of the next segment is begun.

Each succeeding segment is processed in exactly the same way until all the segments have been processed and written on the Output tape. The final running program is then available on Uniservo tape and control is transferred to the Side-by-Side Listing section of the Automatic Coding Program.

UNICODE--APPENDIX E
GLOSSARY

Alphabetic Character - Any of the letters of the alphabet A through Z.

Alphanumeric Character - An alphabetic or numeric character.

Argument - A variable which may take on any value.

Binary Operation - An operation applied to two operands.

Body - The statements of a program or subprogram collectively.

Character - Any of the letters of the alphabet, numeric digits, or other typewritten representations available.

Coding - A sequence of machine instructions.

Comments - The optional, uninterpreted part of a line.

Equation - The definition of a variable, subscripted or not, by an expression to the right of the equality symbol.

Heading of a Subprogram - The first two sentences of a subprogram.

High Speed Storage - Rapid access magnetic core storage of the computer consisting of a minimum of 4096 storage locations.

Input Program - See *UNICODE* Program.

Instruction - See *UNICODE* Instruction.

Library - See *UNICODE* Library.

Line - The fundamental unit of a program written in *UNICODE* language.

Line Number - A number used to reference a line.

Magnetic Drum - Medium access storage of the computer consisting of 16,384 storage locations.

Numeric Character - Any of the digits 0 through 9.

Operand - A variable, constant, or expression.

Object Program - The set of computer instructions produced by *UNICODE*.

Output Program - See Object Program.

Preface - The coding necessary to transfer data from magnetic drum to high speed storage prior to the execution of a segment.

Pseudo Operation - An operation performed by a subprogram written in *UNICODE* language.

Segment - A high speed storage load of coding and the data referenced by this coding.

Segmentation - The process of separating the object program into segments.

Sentence - That part of a line interpreted by *UNICODE* to produce coding, or to indicate special conditions.

Source Program - See *UNICODE* Program.

Statement - A sentence following the *START* instruction which controls the sequencing of a program.

Subprogram - A group of sentences which perform a pseudo operation.

Subscript - A variable which may take on positive integral values only.

Symbol - Any combination of six or less characters.

Termination - The coding necessary to transfer data from high speed storage to magnetic drum following the execution of a segment.

Unary Operation - An operation applied to a single operand.

UNICODE - Automatic coding system for the Univac Scientific (Model 1103A).

UNICODE Instruction - Any command which can be interpreted by *UNICODE*.

UNICODE Language - The language which can be interpreted to produce an object program.

UNICODE Library - A collection of frequently used routines such as sine, cosine, and square root.

UNICODE Program - A sequence of equations and instructions written in *UNICODE* language.

UNICODE--APPENDIX F

UNIVAC SCIENTIFIC (MODEL 1103A) SYSTEM CHARACTERISTICS AND INSTRUCTION REPERTOIRE

WORD CHARACTERISTICS, ADDRESSES, REGISTERS, TYPEWRITER CODE

INSTRUCTION WORD CHARACTERISTICS

An instruction word consists of three parts: a 6-bit operation code, a 15-bit first execution address (u-address), and a 15-bit second execution address (v-address). The sequence is operation code, u-address, v-address.

In some instructions, u-address is replaced by jn or jk, in others v-address is replaced by k. The functions of j, n, and k are:

j—a one-digit octal number modifying the instruction ($j=u_{14}, u_{13}, u_{12}$).

n—a four-digit octal number designating number of times instruction is to be performed ($n=u_{11}, u_{10}, \ldots\ldots u_0$).

k—a seven-bit binary number designating number of places word is to be shifted to left ($k=v_6, v_5, \ldots v_0$ except in Left Transmit where $k=u_6, u_5, \ldots u_0$).

WORD EXTENSION CHARACTERISTICS *

D(u)—a 72-bit word whose right hand 36 bits are (u) and whose left-hand 36 bits are same as left-most bit of (u).

S(u)—same as D(u) except left-hand 36 bits are zero.

D(Q)—a 72-bit word whose right-hand 36 bits are (Q) and the left-hand 36 bits are same as left-most bit of (Q).

S(Q)—same as D(Q) except left-hand 36 bits are zero.

$D(A_R)$—a 72-bit word whose right hand 36 bits are (A_R) and whose left-hand 36 bits are same as left-most bit of (A_R).

$S(A_R)$—same as $D(A_R)$ except left-hand 36 bits are zero.

L(Q)(u)—a 72-bit word whose left-hand 36 bits are zero and whose right-hand 36 bits are bit by bit product of corresponding bits of (Q) and (u).

L(Q)'(v)—same as L(Q)(u) except right-hand 36 bits are bit by bit product of corresponding bits of (Q)' and (v).

*Brackets denotes contents of, as (A) means contents of A.

REGISTERS

CONTROL REGISTERS

PCR (Program Control Register)——Stores instruction during its execution, consists of:

- MCR (Main Control Register)—Stores 6-bit operation code
- UAK(U-Address Counter) ——Stores 15-bit u-address
- VAK (V-Address Counter) ——Stores 15-bit v-address.

PAK (Program Address Counter)—Additive counter to generate successive addresses for obtaining program instructions.

SAR (Storage Address Register) ——Stores address during a reference to storage; lower seven stages serve as shift counter during shifting of A or Q.

INPUT-OUTPUT REGISTERS

IOA —— 8-bit in-out register.
IOB ——36-bit in-out register.
TR ——36-bit in-out tape register.
TWR—— 6-bit out typewriter register.
HPR —— 7-bit out high-speed punch register.

ARITHMETIC SECTION REGISTERS

A ——72-bit accumulator with shifting properties.
A_R——right-hand 36 bits of A.
A_L——left-hand 36 bits of A.
Q ——36-bit register with shifting properties.
X ——36-bit exchange register.

ADDRESSES

ALLOCATIONS

MC—00000—07777
(4,096 36-bit words)
00000—17777
(8,192 36-bit words)
00000—27777
(12,288 36-bit words)
Q—31000—31777
(1 36-bit word)
A—32000—37777
(1 72-bit word)
MD—40000—77777
(16,384 36-bit words)

FIXED

F_1——00000 or 40001
F_2——00001
F_3——00002
F_4——00003

TYPEWRITER CODE (OCTAL)

LETTERS

UC	LC	No.	UC	LC	No.
A	a	30	N	n	06
B	b	23	O	o	03
C	c	16	P	p	15
D	d	22	Q	q	35
E	e	20	R	r	12
F	f	26	S	s	24
G	g	13	T	t	01
H	h	05	U	u	34
I	i	14	V	v	17
J	j	32	W	w	31
K	k	36	X	x	27
L	l	11	Y	y	25
M	m	07	Z	z	21

SIGNS

–	–	56	(	,	46
.	=	44	)	.	42
/	+	54	–	|	50

NUMBERS

UC	LC	No.	UC	LC	No.
1	1	52	6	6	66
2	2	74	7	7	72
3	3	70	8	8	60
4	4	64	9	9	33
5	5	62	0	0	37

FUNCTIONS

SPACE	04
BACK SPACE	61
CAR. RETURN	45
TABULATOR	51
STOP	43
SHIFT UP	47
SHIFT DOWN	57
COLOR SH.	02
CODE DEL.	77

REPERTOIRE OF INSTRUCTIONS

01 POLYNOMIAL MULTIPLY (FPuv).....$(u)(Q) + (v) \longrightarrow Q$ *

02 INNER PRODUCT (Fluv).....$(u)(v) + (Q) \longrightarrow Q$ *

03 UNPACK (UPuv).....$(u)_M \longrightarrow u$, $(u)_C \longrightarrow v_C$, sign bits transferred to u_C, zero transferred to v_M *

04 NORMALIZE PACK (NPuv).....$(v)_C$ and $(u)_f$ are normalized, rounded and packed into u *

05 ROUND OPTION (FRj-).....if j=0, round; if j=1, no round *

11 TRANSMIT POSITIVE (TPuv).....$(u) \longrightarrow v$

12 TRANSMIT MAGNITUDE (TMuv)..... $|(u)| \longrightarrow v$

13 TRANSMIT NEGATIVE (TNuv)..... $(u)' \longrightarrow v$

14 INTERPRET (IPxx).....(PAK)$\longrightarrow F_1$, take (F_2)

15 TRANSMIT U-ADDRESS (TUuv)..... $(u_{15-29}) \longrightarrow v_{15-29}$

16 TRANSMIT V-ADDRESS (TVuv)..... $(u_{0-14}) \longrightarrow v_{0-14}$

17 EXTERNAL FUNCTION (EF-v).....Select External Equipment and Perform (v)

21 REPLACE ADD (RAuv).....$(u) + (v) \longrightarrow u$

22 LEFT TRANSMIT (LTjkv).....Shift (A) by k; j=0, $(A_L)_f \longrightarrow v$; j=1, $(A_R)_f \longrightarrow v$

23 REPLACE SUBTRACT (RSuv)..... $(u) - (v) \longrightarrow u$

27 CONTROLLED COMPLEMENT (CCuv)..... $(u) \oplus (v) \longrightarrow u$

31 SPLIT POSITIVE ENTRY (SPuk).....$S(u) \longrightarrow A$, shift (A) by k

32 SPLIT ADD (SAuk)..... $(A) + S(u)$, shift (A) by k

33 SPLIT NEGATIVE ENTRY (SNuk)..... $[S(u)]' \longrightarrow A$, shift (A) by k

34 SPLIT SUBTRACT (SSuk)..... $(A) - S(u)$, shift (A) by k

35 ADD AND TRANSMIT (ATuv)..... $(A) + D(u) \longrightarrow v$

36 SUBTRACT AND TRANSMIT (STuv)..... $(A) - D(u) \longrightarrow v$

37 RETURN JUMP (RJuv).....(PAK)$\longrightarrow u_v$, take (v)

41 INDEX JUMP (IJuv).....$D(u) - 1 \longrightarrow A$; $(A)_f \geq 0$, $(A) \longrightarrow u$ and take (v)

42 THRESHOLD JUMP (TJuv)..... $(u) > (A)$, take (v)

43 EQUALITY JUMP (EJuv)..... $(u) = (A)$, take (v)

44 Q-JUMP (QJuv).....(Q) -, take (u); (Q) +, take (v); (Q) left 1

45 MANUALLY SELECTIVE JUMP (MJjv)....j = 0, take (v); j = 1, 2, 3 and MJS = j, take (v)

46 SIGN JUMP (SJuv)..... (A) -, take (u); (A) +, take (v)

47 ZERO JUMP (ZJuv)..... $(A) \neq 0$, take (u); (A) = 0, take (v)

51 Q-CONTROLLED TRANSMIT (QTuv)..... $L(Q)(u) \longrightarrow v$

52 Q-CONTROLLED ADD (QAuv)..... $(A) + L(Q)(u) \longrightarrow v$

53 Q-CONTROLLED SUBSTITUTE (QSuv).... $L(Q)(u) + L(Q)'(v) \longrightarrow v$

54 LEFT SHIFT IN A (LAuk)..... $D(u) \longrightarrow A$, shift (A) by k, $(A)_f$ u

55 LEFT SHIFT IN Q (LQuk)..... $(u) \longrightarrow Q$, shift (Q) by k, $(Q)_f \longrightarrow u$

56 MANUALLY SELECTIVE STOP (MSjv)....j = 0, stop; j = 1, 2, 3, and MSS = j, stop

57 PROGRAM STOP (PS--).....Stop and Indicate

61 PRINT (PR-v).....Typewriter performs code in v_{0-5}

63 PUNCH (PUjv).....Punch (v_{0-5}); j = 1, also 7th level

64 ADD (FAuv)..... $(u) + (v) \longrightarrow Q$ *

65 SUBTRACT (FSuv)..... $(u) - (v) \longrightarrow Q$ *

66 MULTIPLY (FMuv)..... $(u)(v) \longrightarrow Q$ *

67 DIVIDE (FDuv)..... $(u) \div (v) \longrightarrow Q$ *

71 MULTIPLY (MPuv)..... $(u)(v) \longrightarrow A$

72 MULTIPLY ADD (MAuv)..... $(A)_i + (u)(v) = (A)_f$

73 DIVIDE (DVuv)..... $(A)_i \div (u) = (Q)$, $[(A)_f = + R]$; $(Q) \longrightarrow v$

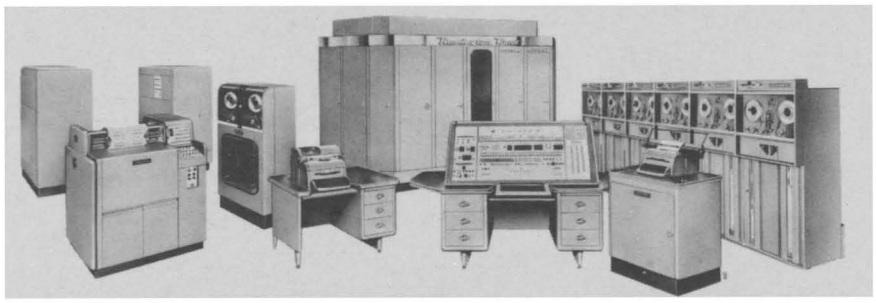
74 SCALE FACTOR (SFuv).....D(u) in A, shift A by 36, shift A until $A_{34} \neq A_{35}$, (SK)$\longrightarrow v_v$

75 REPEAT (RPjnw).....w$\longrightarrow$v of (F_1), Execute NI n times, jump to F_1

76 EXTERNAL READ (ERjv).....j = 0, (IOA)$\longrightarrow$v; j = 1, (IOB)$\longrightarrow$v

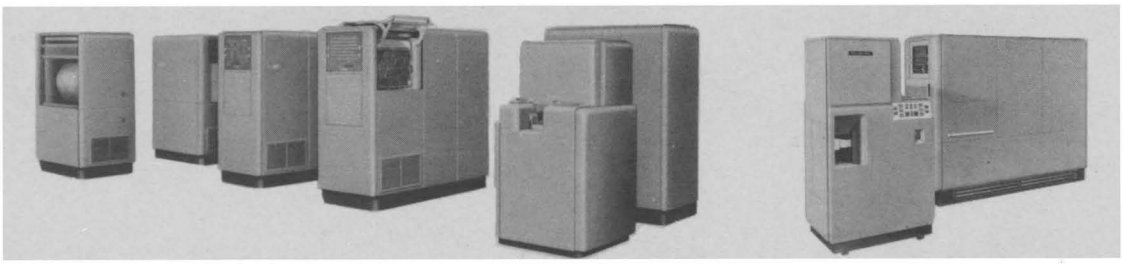
77 EXTERNAL WRITE (EWjv).....j = 0, $(v)_{0-7} \longrightarrow$IOA; j = 1, (v) $\longrightarrow$IOB

* Floating point arithmetic



Univac II Systems • For data-automation which involves large volumes of input and output.

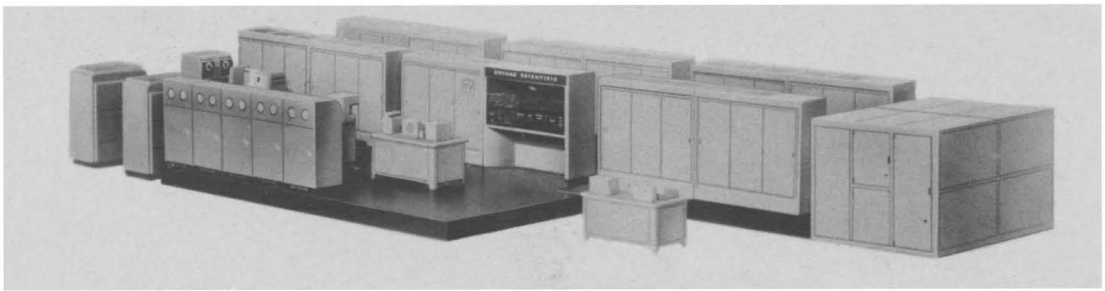
THE UNIVAC® FAMILY



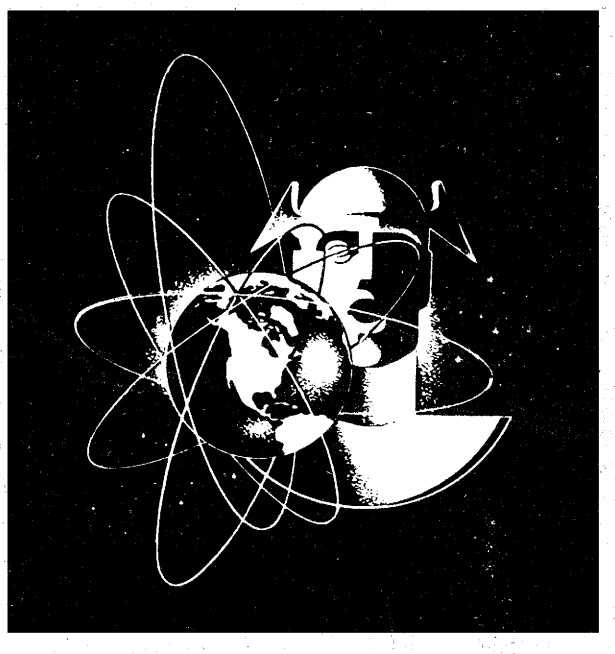
Univac File-Computer • For instantaneous random access to large-scale internal storage—plus computation.

Univac 60 & 120 Computers • For speeding and simplifying the procedures of punched-card systems.

OF ELECTRONIC COMPUTERS



Univac Scientific Systems • For complex and intricate computations of engineering and research.



UNIVAC®—The FIRST Name in Electronic Computing Systems

U 1451 PRINTED IN U.S.A.