RECOMP II USER'S PROGRAM NO. 1117

PROGRAM TITLE:	AFCOR ALGEBRAIC COMPILER MANUAL
PROGRAM CLASSIFICATION:	Executive and Control
AUTHOR:	BROADVIEW RESEARCH CORPORATION
PURPOSE:	To translate Algebraic and control statements into symbolic coding which can then be assembled into machine language by AFAR (A Symbolic Assembly Program).
DATE:	May 1961

Published by

RECOMP User's Library

at

AUTONETICS INDUSTRIAL PRODUCTS
A DIVISION OF NORTH AMERICAN AVIATION, INC.
3400 E. 70th Street, Long Beach 5, Calif.

FOREWORD

This manual describes the operation and use of the algebraic compiler prepared under contract AF 23(601)-2857 for the Aeronautical Chart and Information Center, U.S. Air Force, by Broadview Research Corporation.

The operation and use of the symbolic assembler, which can be used to produce machine-language programs, is described in BRC 161-9-Rev., AFAR Symbolic Assembler Manual.

The flow charts and coding for the compiler and assembler programs appear in the following documents:

BRC 161-11-I, AFAR Symbolic Assembler Flow Charts

BRC 161-11-II, AFAR Symbolic Assembler Coding

BRC 161-14-I, AFCOR Algebraic Compiler Flow Charts

BRC 161-14-II, AFCOR Algebraic Compiler Coding

TABLE OF CONTENTS

LIST OF TABLES

Section I
INTRODUCTION

This manual describes the AFCOR compiler designed and programmed by Broadview Research Corporation for the RECOMP II computer under a contract with the Aeronatuical Chart and Information Center (ACIC).

With the AFAR assembly program, which was prepared under the same contract, a complete system is available to produce object programs for a variety of scientific computing applications from an easily written source language.

The source language is patterned after the IBM FORTRAN* compiler language but contains several variations which reflect the scope and character of the RECOMP II digital computer.

The complete compiler-assembly system is designed for use with the Systematics card-to-paper-tape converter at ACIC.

The over-all system operation consists of the following steps:

1. Punch source program in cards
2. Convert cards to source program paper tape
3. Generate symbolic paper tape object program, using the one-pass AFCOR compiler
4. Generate absolute machine language paper tape object program in two passes of the AFAR** assembly program.

This manual defines and illustrates the source language and the operation of the compiler and object programs.

---

* FORTRAN, copyrighted by International Business Machines Corporation, New York, N.Y.

** The use and operation of the AFAR assembly program is described in BRC 161-9 produced under the subject contract in October, 1960.

Section II
ELEMENTS OF THE AFCOR LANGUAGE

CONSTANTS

In the AFCOR language, fixed point constants are distinguished by the absence of a decimal point. They have from one to six decimal digits. Hence, 999999 is the maximum fixed point integer acceptable to the AFCOR compiler. Every fixed point integer is stored in a single full RECOMP word.

Floating point constants always have a decimal point. They have from one to twelve decimal digits. Every floating point constant is stored in two RECOMP words. Examples of floating point constants are:

```
43.2
57.1700321
1.
.0001
0.732168
.5
```

VARIABLES

Two modes of variables correspond to the two modes of numerical constants. Variables have from one to five alphabetic characters. The last character must not be F since names ending in F are reserved for library functions. The use of all symbolic names beginning with JJ and RR is restricted to the compiler. The modes of variables are specified by the first letter of their alphabetic names. The first letter of a fixed point variable must be one of the letters from I through N. The first letter of a floating point variable must be one of the letters from O through Z.

Apart from the fixed and floating point modes, there are two general kinds of variables. These are subscripted and nonsubscripted variables.

At any given time, a nonsubscripted variable has only one numerical value, either a fixed point integer or a floating point value, depending on the first letter of its name. Thus, a nonsubscripted variable name defines a storage area (one or two words) that contains a single numerical value. The contents or value of such a variable can either be input or computed by means of a formula.

A subscripted variable defines an array of several values. Every subscripted variable must be defined in a DIMENSION statement, which must precede all executable source statements, that contains the maximum value of each subscript. The array can be one- or two-dimensional, that is the values can have one or two subscripts.

A one-dimensional subscripted variable can be defined for many different purposes, for example:

1. To represent a vector
2. To list individual but related elements
3. To define a sequence of numbers, such as a set of points at which an algebraic expression should be evaluated

A two-dimensional subscripted variable can be thought of as a matrix or an array in two dimensions. It is a convenient way, for example, of representing tables of functions of two variables.

For all subscripted variables, the subscripts must always take positive nonzero values and be either fixed point constants or nonsubscripted, fixed point variables. Examples of variables are shown in the following tabulation.

Specification	Mode	Remarks
KOUNT	Fixed point integer	Nonsubscripted
KAY(3)	" " "	One-dimensional subscripted
XVAL	Floating point	Nonsubscripted
Y(I)	" "	One-dimensional subscripted
X(4,IVAL)	" "	Two-dimensional subscripted
Z (10,10)	" "	" "
ZZ (I,J)	" "	" "

EXPRESSIONS AND FORMULAS

The AFCOR compiler provides the programmer a means of specifying computations in a language closely paralleling standard algebraic notation. The four arithmetic operations have the following symbols:

Addition	+
Subtraction	-
Multiplication	*
Division	/

These symbols are used to connect constants, variables, and functions to form expressions. Left and right parentheses are allowed to group subexpressions and to designate the desired hierarchy or order of computation. The mode of a formula is determined by the mode of each of its elements. Mixed formulas with both floating and fixed point elements are strictly prohibited and are signalled as errors by the compiler. Fixed point division, of course, truncates.

In the AFCOR language, the general form of a formula statement (or equation) is:

Variable = Expression

Examples of legal formulas are shown in the following tabulation:

Legal Formula Statements
ZB = 4.*T(4)
X = 3.172*Z
Y(K,2) = W + T(K)
VALUE = 1000.0
POINT = ROOTF(X,Y,Z)
U(2) = -.1
SU = 1.4142*(-Q-R*(Q-R))
W = PWERF (10.,SAM)/(ZB*ZB-1.0)+R
IJKL = 2*IJKL
JKL = IJKL/2
JUMP = (N+M)/4 - INCRF (7,M)*10
K(L) = 3
LA = 3 + 4*(KAY/(1-JAY))
N(4) = M(1)

One exceptional use of the minus sign, called the unary minus, is allowed. This indicates a change in the sign of the following variable (or subexpression) rather than true subtraction from a previous number. Unary plus signs are not allowed.

Examples of illegal formulas with an explanation of the errors involved are shown in the following tabulation:

Illegal Formula Statements	Reasons
I = W	Mixed modes
KI = +4	Unary plus not allowed
X = 3.(Q)	Should be 3.*Q
X = 3*Q	Mixed modes (3 lacks decimal point required in all floating point numbers)

Illegal Formula Statements	Reasons
2.*ZR = ROOTF(X,Y,Z)	The left side may be only a variable, not an expression
X = P*((Q*(R-T)/V+X)	There must be the same number of left and right parentheses
TEMP = -1400.+/(3.*ZAF)	Two arithmetic operation symbols in succession

FUNCTIONS

The AFCOR compiler source language provides the capability of defining functions that can be added to generated symbolic coding. In this way, library routines programmed externally can be used in conjunction with the object program.

The LIBRARY statement is included in the source language to define the symbolic name of functions (or subroutines) and to define the kind of calling sequence which the compiler must generate for each appearance of a function in an expression.

In a LIBRARY statement, the name of a function is followed by parentheses enclosing a parameter list and a result indication. Several functions can be defined in a single LIBRARY statement by using commas to separate different specifications.

The name of a function can be any combination of from one to five alphabetic characters, the last of which must be F in order for the compiler to distinguish between subscripted variables and functions when translating expressions.

The parameter list must contain from one to eight designators constructed according to the following rule. The parameter list can contain: either (a) only one element, 0, or (b) any combination of the digits 1 through 7 and the asterisk, but none may appear more than once.

The parameter list of a function in the LIBRARY statement

designates where the specific arguments shall be placed relative to generated transfers to the function subroutine. A typical function-defining statement is:

```
LIBRARY, ROOTF(1,2,3,*), DLTAF (*,*), PWERF (*,1,*), INVRF(0,0)
```

When a function is specified in a source language expression, that is, when it is an element of the right-hand side of a formula, the name of the function is written, followed by an argument list enclosed in parentheses. The argument list contains variables or constants in one-to-one correspondence with the proper parameter list of a LIBRARY statement.

If $p_1 \ldots p_k$ are the nonzero designators of the parameter list, and $a_1 \ldots a_k$ are the variables or constants in the argument list and k is equal to or less than 8, and c equals the number of words required to contain each argument, that is, c equals 1 for fixed point integers and 2 for floating point numbers, then, in general, the value of the argument, a_i, is stored in the p_ith group of c word(s) following the transfer, if p_i is not an asterisk.

It is extremely important to note that all the constants and/or variables in a particular argument list must be of the same mode, that is, they must either be all floating point or all fixed point. There is no restriction within the compiler against using the same function with an all floating-point-mode argument list in some instances and with an all fixed-point-mode argument list in others. Every externally programmed function routine must be able to differentiate between the two kinds of calling sequences if both modes are to be used in a source language program.

Floating Point Arguments

The following are examples of particular specifications using

the functions defined in the sample LIBRARY statement above:

```
ROOTF (PA,PB,PC)
DLTAF (TIME)
PWERF (10.,R)
INVRF
```

The first three examples have parameter lists containing floating point variables or constants only which satisfy the fundamental requirement of mode agreement. The fourth example requires no arguments since its parameter list is zero.

There is one built-in floating point function that need not be defined in a LIBRARY statement. The built-in function, FSQF, calculates the square root of its floating point argument.

```
T = FSQF (4.7318631)
ROOT = FSQF (ZBAR)
```

The following schematic illustrates the fact that if the argument list is in floating point mode, the transfer that is generated is made to appear in the left half of a word whose address is even. The next three half words are skipped so that the first argument's location will always be in the form of XXXE.0, where E denotes an even octal digit.

If, according to the result indication, the result of a function is to be stored in one of the seven storages immediately following the transfer, then such a reservation will be assigned by the compiler. If numbers are stored following a transfer in the object program, then the subroutine can pick up the arguments from the calling sequence by use of the X register which is set by the transfer.

When the argument list is in floating point, two words are reserved for each argument. The first word contains the normalized fraction; the second, the binary exponent or characteristic.

Schematic

Example: Object Program corresponding to ROOTF (PA,PB,PC)

Location	Contents	Parameter No.
XXXE.0	TRA ROOTF	
.1	Not used	
XXXF.0	Not used	
.1	Not used	
XXXG.0	Storage for PA,	1
.1	fraction portion	
XXXH.0	Storage for binary	
.1	exponent of PA	
XXXI.0	Storage for PB,	2
.1	fraction portion	
XXXJ.0	Storage for binary	
.1	exponent of PB	
XXXK.0	Storage for PC,	3
.1	fraction portion	
XXXL.0	Storage for binary	
.1	exponent of PC	

The numbers (or asterisk) of a parameter list need not be in any special order, nor is it mandatory not to skip certain numbers. The latter will merely have the effect of generating enough zero words to fill the void(s) left by the omitted number(s) of the argument list.

If an asterisk appears in a parameter list or the result indication, the corresponding value from the argument list will be placed in:

1. A and R registers if the mode of the argument list is floating point
2. A register if the mode is fixed point

Fixed Point Arguments

In dealing with functions of floating point arguments, two full computer words are reserved in the calling sequence for each argument and result except when the corresponding element of the parameter list is 0 or an asterisk.

When the argument list in a function specification is in fixed point integer mode, only one full word is reserved for each fixed point constant or variable. Therefore, the compiler generates the transfer instruction in the next left half word.

In general practice, each function subroutine will be programmed externally (as opposed to being generated by AFCOR) to handle only one kind of calling sequence, that is, either all floating point or all fixed point. Special subroutines that can accept either type of calling sequence will generally be constructed so that by examining the value of a calling sequence argument, the mode can be distinguished, for example:

```
LIBRARY, POLYF (*,1,2,3,4,5,6,7,*)
KAPPA = POLYF (-5,KX,KA,KB,KC,KD,KE,KF)
ZEE = POLYF (4.,ZX,ZA,ZB,ZC,ZD,ZE,0.)
```

As a subroutine POLYF would require the following abilities in order for all three of the above statements to appear in one source program:

1. The first argument always appears in the A register. If its sign is minus, it is taken to mean that the arguments are all fixed point integers, hence the absolute value of the integer in the A register is regarded as the degree of a polynomial to be evaluated. The word following the transfer, corresponding to the designator in the parameter list having the value 1, is the value of the independent variable, KX in this

example. The remaining values are coefficients of a fifth degree polynomial; each occupies one word of the calling sequence. In this case, the routine performs the calculations in fixed point arithmetic and returns with the answer in the A register.

2. If the sign of the A register is plus, then A and R, in floating point, specify the degree of the polynomial desired. The subroutine then obtains the floating point independent variable, ZX, from the two words following the transfer. The remaining values are floating point coefficients of a fourth degree polynomial; each occupies two words. Since this example specifies only a fourth degree polynomial, the coefficient of ZX^5, which normally occupies the seventh pair of words after the transfer, is not used by the subroutine. However, the mistake of writing an argument list shorter than the parameter list must not be made. Therefore, a dummy value of zero (floating point : 0.) is given.

Section III
AFCOR SOURCE LANGUAGE STATEMENTS

The statements of a source program can be classified into the following categories:

1. Formulas
2. Specification
3. Control
4. Input-output

This section deals with specification and control statements. Formula statements were discussed in the previous section, and input-output statements are discussed in the next section.

The AFCOR language has six different kinds of specification statements. They provide for the definition of subscripted variables, the definition of functions, the generation of an end stop for the object program, the setting of the START1 and START2 buttons, and a signal to the compiler indicating the end of the source language.

There are eight different kinds of control statements, providing for testing, branching, transferring, looping, and intermediate program stops.

There are seven different kinds of input-output statements providing for input from Baudot paper tape, typewriter, or console and for output via paper tape punch, typewriter (or both), or by decimal display in the Nixie tubes.

As shown in Table 1, standard mnemonics are used to define the general form of each kind of statement. A two digit integer code is associated with each kind of statement. These code numbers

never appear in the source language but are generated internally by the compiler and are used for display purposes if an error in compilation necessitates a stop. With these numbers and another set of "kind of error" codes which are also displayed, any error in a source program can be located.

In defining the general form of statements, those parts which are not required are underlined; all other parts are mandatory.

Table 1

SYMBOLS USED IN EXPLAINING SOURCE LANGUAGE STATEMENTS

Generic Symbol	Definition	Restrictions	Examples	Baudot Mode
n	Statement number	1-5 digits written continuously with no embedded blanks; unsigned	15 1000 00088	fs
v	Variable	Fixed pt. integer or floating pt. mode. Subscripted or not. Name 1-5 characters not ending with F	PVAL K(3) X(1,30)	ls (Name)
i	Fixed point variable, non-subscripted	Name begins with I-N, does not end with F; not subscripted	J NUMBER	ls
k	Fixed point constant	1-6 digits; no embedded blanks; unsigned (positive)	99703 318 0	fs
m	i or k	Should only assume positive nonzero values		
w	Sense switch	One of the letters B,C, or D		ls
j	Display digit	Unsigned single digit (0-9)		fs
s	Subscripted variable	Fixed or floating pt. mode	V(K) Z(14,11)	ls
x	Floating point variable	Subscripted or not	V(K) QINC P(2)	ls

Table 1

SYMBOLS USED IN EXPLAINING SOURCE LANGUAGE STATEMENTS
(Continued)

Generic Symbol	Definition	Restrictions	Examples	Baudot Mode
fn	Function	Name 1-5 characters, ending in F. Must be followed by parentheses enclosing parameter list and result indication	FSQRF(a,r)[1/] DLTAF (a,r) ROOTF(a,r)	ls (Name)
p	Parameter list	Up to 7 numbers[2/], not including last, enclosed in parentheses, indicating where to place arguments with respect to transfer to function routine	(1,2,*) (*,*) (3,1,4,3) (0,0)	fs
r	Result indication	Indicates where (1-7 or *) to store answer in calling sequence (if r = 0, no answer is obtained from the function routine); always follows parameter list	See above example; the last character before right parenthesis is "r"	fs
a	Argument list	Variables or constants listed, in 1-1 correspondence to parameter list, as arguments or "inputs" for a function subroutine	(X,Y,Z) (17.4,Y) (KAY,2)	ls or fs
q	n or i			fs or ls

1/ (a,r) indicates argument list and result indication which must be enclosed in parentheses following function name.

2/ In addition, the asterisk (*) symbol may be used to indicate that an argument or a result is to be placed in the A register (or A and R registers in the case of floating point variables). The numbers must be in the range 1-7. Zero denotes the vacuous state (no argument list or no result).

Table 2

SPECIFICATION STATEMENTS

Statement Code No.	General Form	Examples
18	DIMENSION, S_1 (K_1, K_2), . . .	DIMENSION, U(2), T(4,8), K(10), POINT(12,13)
	Remarks: Every one-dimensional variable must be listed by name, followed by the maxi-maximum value of the subscript, enclosed in parentheses. Every two-dimensional variable must contain the maximum values of the two subscripts, enclosed in parentheses and separated by a comma. Values assumed by subscripts must not exceed their respective maxima during execution of an object program. Such an excess results in an incorrect data address. No testing for this type of error is performed. Similarly, subscripts must not take on a value less than or equal to 0. Dimension statement must precede all executable statements.	
20	LIBRARY, fn(a,r), . . .	LIBRARY, ROOTF(1,2,3,*), INVRF (1,0)
	Remarks: See "Functions". Must precede all executable statements.	
19	n STOP j	1313 STOP 9
	Remarks: The computer comes to a final halt with J displayed in the Nixie tubes. Execution of the object program cannot be resumed by pressing START. If no j part is specified, the display in the Nixie tubes is not altered by the execution of this statement.	
15	n START 1(n_1), 2(n_2)	START 1 (20)
	Remarks: Transfers to the statement numbers enclosed in parentheses can be compiled in either locations 0001.0 or 0002.0 or both. By this means, multiple start or restart paths can be selected by pressing the START1 or START2 buttons. The START3 button cannot be set by the source program. The START statement must precede all executable statements.	
17	END	END
	Remarks: Signals end of source program input to compiler. No executable object instructions corresponding to this statement are compiled. All statements must precede the END statement.	

Table 3

CONTROL STATEMENTS

Statement Code No.	General Form	Examples
16	n PAUSE j	111 PAUSE 7
	Remarks: The computer comes to a halt in the execution of the object program, displaying the digit j preceded by 3 decimal points. If j is not present, no display is given, so that any previous displayed information remains in the Nixie tubes. Pressing START will cause control to proceed to the next executable statement.	
14	n ASSIGN (i) n	ASSIGN (ISW) 111
	Remarks: Statement number n is assigned to the nonsubscripted fixed point variable enclosed in parentheses. Used in conjunction with an "assigned" GO TO control statement.	
03	n GO TO q	17 GO TO (NSW) GO TO (304)
	Remarks: Causes transfer of control to the statement number indicated by the value of q. If q is not fixed point constant, it must be a nonsubscripted fixed point variable preset by an ASSIGN statement.	
04	n IF (v) n_1, n_2, n_3	IF (WB(K)) 401,402,402
	Remarks: The value of the variable in parentheses, which can be subscripted, is tested. If it is zero, control is transferred to statement n_2. If v is not zero, then control is transferred to n_1 for negative values or n_3 for positive values.	
13	n IF SENSE w, n_1, n_2	937 IF SENSE [illegible], 940,950
	Remarks: If the sense switch designated by w, a single letter, is ON, control is transferred to statement number n_1. Otherwise (OFF), control is transferred to statement n_2.	

Table 3

CONTROL STATEMENTS (Continued)

Statement Code No.	General Form	Examples
01	$\underline{n_1}$ DO n_2 i : m_1, m_2, $\underline{m_3}$	60 DO 65 IM : 1,JMAX

Remarks: This statement sets up a fixed point counter which controls the repeated execution of all statements between the DO statement and statement n_2, which must be a CONTINUE statement appearing later in the source program. The number of times the "loop" is executed is determined by the following. If m_3 is not present, a value of 1 is used for m_3.

(1) The loop is executed for i = m_1

(2) $m_1 = m_1 + m_3$

(3) If m_1 is now greater than m_2, control proceeds to the next executable statement following statement number n_2

(4) Otherwise, the loop is again executed: repeat from step 2

Statements between the DO and the continue can supersede the nominal loop control outlined above. The counter i is normally but not necessarily a subscript of variables in formulas that are evaluated in the loop. At any given time, the value of i is available, in the same sense as a fixed point integer variable, for testing or computation. It is possible to construct "nests" of DO loops, provided each DO statement is paired with its own unique CONTINUE statement. It is the responsibility of the programmer to design source programs so that m_1, m_2, and m_3 have meaningful values, since the object program cannot check for errors. Incorrect m-values could cause the program to "hang up" in an unending loop or to malfunction in some other way during execution.

Table 3

CONTROL STATEMENTS (Continued)

Statement Code No.	General Form	Examples
02	n CONTINUE	65 CONTINUE

Remarks: This statement causes the compiler to generate the object instructions that compute the next value of i and test whether to execute the DO loop again or to continue to the next executable statement. Each DO must have a unique CONTINUE associated with it; there is no other legal use of this statement.

Table 4

INPUT-OUTPUT AND FORMULA STATEMENT CODE NUMBERS

Statement Code No.	General Form
05	n READ TAPE, input list
06	n PUNCH n, output list
07	n TYPE n, output list
08	n DISPLAY, v
09	n PUNCH AND TYPE n, output list
10	n READ TYPER, input list
11	n READ CONSOLE, input list
12	n FORMAT (. . .)
00	n v = expression

Section IV

INPUT-OUTPUT AND FORMAT STATEMENTS

INPUT STATEMENTS

There are three sources of data input for the RECOMP II:

1. Paper tape
2. Typewriter
3. Console

There are three corresponding source language statements:

1. READ TAPE
2. READ TYPER
3. READ CONSOLE

Each of the above statements is followed by a list that indicates which variables are to be read from the specified input unit.

Elements of the input list are separated by commas. Except for HEDnnn:k, which is allowed only in the READ TAPE statement, the following list illustrates every possible form of the elements of input lists:

```
X
X(2)
X(I)
X(I,J)
X(2,J)
X(I,6)
X(3,4)
X(I,J), I:2,6,J:1,10
X(J),J:12,15
```

HEDnnn:k specifies that words in the RECOMP II alphanumeric (F) mode (8 Baudot characters per word) are to be read in and stored. These alphanumeric words are available for print-output if their label, HEDnnn:k, is included in an output list. Some typical alphanumeric labels are:

HED000 : 2
HED8 : 3
HED12 : 4
HED775 : 8

A discussion of the method of storing two-dimensional variables is necessary to understand the restrictions on elements of an input or output list. If X is a two-dimensional variable whose dimensions are defi ned in a DIMENSION statement as (5,7), the following array is defined:

	Column						
Row	1	2	3	4	5	6	7
1	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}	X_{16}	X_{17}
2	X_{21}	X_{22}	X_{23}	X_{24}	X_{25}	X_{26}	X_{27}
3	X_{31}	X_{32}	X_{33}	X_{34}	X_{35}	X_{36}	X_{37}
4	X_{41}	X_{42}	X_{43}	X_{44}	X_{45}	X_{46}	X_{47}
5	X_{51}	X_{52}	X_{53}	X_{54}	X_{55}	X_{56}	X_{57}

An individual element of the array is specified by writing X(I,J), where:

$$1 \leq I \leq I_{max} = 5$$
$$1 \leq J \leq J_{max} = 7$$

In general, the (I,J)th element X(I,J) is stored beginning with the word whose address is:

$$X + 2 * \left[(I - 1) * J_{max} + (J - 1) \right]$$

Since elements of a rectangular array (two-dimensional variable or matrix) are stored by rows, X(1,2) is in X + 2. That is, all the values constituting the first row are stored consecutively, beginning with the first storage reserved for the array. This fact is a derivative of the general AFCOR rule that the second (rightmost) subscript is varied while holding the first subscript (leftmost) fixed at its initial value; after the left subscript advances to

the next value, the right subscript again varies over the entire range (from 1 to J_{max}) as defined in the DIMENSION statement.

The simplest way to input a two-dimensional array completely and continuously is to write only the name of the variable without subscripts. For example:

```
READ TAPE, X
```

would bring into memory 5X7 or 35 floating point numbers from paper tape. The thirty-five numbers constitute a single group punched on tape by rows. The whole group is preceded by an N control character and followed by:

```
L00030
S
```

The same result would be obtained by writing:

```
READ TAPE, X(I,J), I:1,5,J:1,7
```

For each element of an input list, one transfer to the input routine is generated. If an element requires more than one value to be input, all the values denoted by the element must occupy consecutive storages since the RECOMP II reads into consecutive memory locations. On tape, there must be a new location setting (L00030) and a start code (S) following each number or set of numbers corresponding to individual elements of an input list. However, after each group of header words, called in by the appearance in the list of an element of the form HEDnnn:k, the location setting must be L00031.

It is not legal to write:

```
READ TAPE, X(I,J),J:1,7,I:1,5
```

because the subscripts are varied in reverse order. It is also not legal to write:

```
READ TAPE, X(I, J),I:1,5
```

```
READ TYPER, X(I,J),J:1,7
```

because the variation of both subscripts must be stated, and it can be stated only by specifying constants. Thus, the following statement is illegal:

```
READ CONSOLE, X(I,J),I:1,N,J:1,M
```

Since reading of a group of numbers is accomplished by automatically storing successive numbers in consecutive words in memory, it is never possible to specify a range of the rightmost subscript other than the complete range (1 to J_{max}). Thus, where X has dimensions (5,7), the statement:

```
READ TAPE, X(I,J);I:2,4,J:1,4
```

is illegal, but the statement:

```
READ TAPE X(I,J),I:2,4,J:1,7
```

is legal, because an input statement can generally specify any subset of consecutive full rows of a two-dimensional array.

Examples of input statements are:

```
READ TAPE, KAY(K),K:1,14
READ CONSOLE, Z,Y,WW(I),I:1,7
READ TYPER, TA, TB, TC, TD
READ TAPE, HED47:3, HED48:5, PARAM(I,J),I:2,7,J:1,5
READ CONSOLE, MOP, MIP, MAP, ZOP, ZIP, ZAP
READ TYPER VALUE (3, INDEX), INDEX: 1,7, VALUE (1,1),
          VALUE (2,1), LA, LB, LC, T(I), I:1,4
          QAK(13), QAK(5), WRN
```

The usual RECOMP II format letter codes, N or F, must appear on the tape before each such number or set of numbers. Details of Baudot paper tapes, such as the above, and spacing requirements must conform to the rules set down in the Recomp II Operating Manual published by Autonetics Industrial Products. After the numbers are input from either the typewriter or the console, the START3 button must be pressed to transfer control to the input routine. This is done automatically when input is read from paper tape by

means of a new location setting after each number or set of numbers.

OUTPUT STATEMENTS

There are four possible output statements:

1. TYPE n, . . .
2. PUNCH n, . . .
3. PUNCH AND TYPE n, . . .
4. DISPLAY, v

All the above statements, except DISPLAY, are followed by an output list, indicated by the ellipses above. The rules for output lists are the same as those for input lists. The DISPLAY statement, however, is followed by only a single variable.

It it possible to input or output an entire array without specifying the subscript range. Assume that subscripted variables are defined by the following:

DIMENSION, XVAL(4,10), KSET(17)

Then, the statement :

TYPE 909, KSET, XVAL

will type seventeen floating point numbers from the seventeen word storage area KSET, followed by forty floating point numbers from the eighty word storage area XVAL. The format of the typed numbers is determ ined by the configuration of the FORMAT statement number 909.

The FORMAT statement enables the design of a variety of output formats. The TYPE statement must always have an associated FORMAT statement number. The PUNCH statement, when used in association with a format, produces a paper tape suitable for listing on a Flexowriter. Without a format number, a paper tape acceptable as a subsequent input tape in the RECOMP II N or F modes is punched. In the PUNCH and TYPE statement, a format number which is

applicable to typing only, is required. The punching from such a statement will always be in the N or F mode. Hence, this statement does not correspond to a simultaneous punch-and-type operation.

FORMAT STATEMENTS

Every format statement must have a statement number. The specifications are enclosed in parentheses and can be any combination of the following four kinds:

1. kIw
2. kEw.d
3. kFw.d
4. kH . . .

I format

The I format provides a means of outputting fixed point integer variables. Each I specification is followed by w, an integer of one or two digits specifying the width of the desired field; k denotes the number of such I fields. The field prints with a decimal point in the rightmost print position, preceded by the decimal digits to which the value is converted and by a leading minus sign if applicable. Extra print positions in the field to the left of the value print as spaces.

E format

The E format provides a means of outputting floating point variables in "scientific notation" in the form of a fraction < 1.0 and a decimal exponent. This format is preferred to the F format when the magnitude of the numbers requires an excessive number of point positions and prevents the following restriction from being satisfied.

The parameters w and d must be chosen so that two inequalities are satisfied:

$$1 \leq d \leq 12$$
$$w \geq d + 6$$

The value prints a plus or minus sign, a decimal point, d places of significant figures, a space, another sign, and two digits indicating by what power of ten the preceding value should be multiplied.

F format

The F format provides a means of outputting floating point variables to the desired number of decimal places. The general form of F format specifications is kFw.d, and where w specifies the width of the field, d specifies the number of places to the right of the decimal point, and k states the number of such F fields.

If e represents the decimal exponent of a value x to be printed, that is:

$$x = c * 10^{e}, \quad c < 1.0$$

then two inequalities must be satisfied:

$$w \geq d + e' + 2$$
$$d + e' + 2 \leq 16$$

where $e' = e$ if $e \geq 0$, otherwise $e' = 0$.

H format

While the I, E, and F formats control the method of printing numerical values, it is possible to include alphanumeric data in a format. If k characters of alphanumeric format are desired, they are included in the format, preceded by kH. Variable alphanumeric information, such as for headings, may be output by specifying the HEDnnn:k label in the appropriate output list. In other words, the H format is used only to output permanent built-in alphanumeric information.

Each typing output statement may specify only 1 line of printing unless the FORMAT contains T and the typewriter switch is set to interpret the tab character as a carriage return. A carriage

return is given automatically as the first operation of an output statement. Thus, the programmer must take care not to specify, in an output FORMAT, more print positions than are actually available.

If a field is too small for the listed value, the entire field will be printed with asterisks (*) rather than with a truncated value.

Provision for Tabulating

Any of the four output format specifications can be written with provision for tabbing before printing by preceding the I, E, F, or H character by the letter T. By setting the T-CR switch on the typewriter to CR, multiple lines of output may be printed from a single format statement.

Form of Displayed Numbers

The general form for a floating point value is:

± XXXXXXXXXX S YY

Ten significant decimal places are given, preceded by a sign. This is to be regarded as a fraction, i.e. a decimal point between the leading sign and the leftmost digit is understood. A sign expressed as 0 (negative) or 1 (positive) follows set off by a blank Nixie tube on each side. The last two Nixie tubes (YY) express the magnitude of the power of ten by which the preceding decimal fraction is to be multiplied. If YY > 99, two decimal points are displayed in the YY positions.

The general form for a fixed point value is:

± XXXXXXXXXXXXXX

The sign precedes 14 decimal places, and a decimal point is at the far right.

Example of Formats of Floating Point Numbers

Assume that the variable X is stored in the RECOMP II at locations 1220.0 and 1221.0. Assume that these storage words contain the following numbers written in command format:

C(1220.0) = +4701011-0000000

C(1221.0) = +0000000-0000031

Format Specification	Printout of Value of X
F14.10	b78.0317993164
F14.4	bbbbbbb78.0318
F11.9	**********
F7.0	bbbb78.
E14.8	b.78031799b+02
E12.2	bbbbb.78b+02
E13.9	**********

where b represents a blank space.

Example of Format of Fixed Point Integers

Assume that the variable I is stored in location 1734.0;

C(1734) = -0000000-0000061

Format Specification	Printout of Value of I
I4	b-13
I1	*

Section V
SAMPLE PROGRAM

PROBLEM

To find the roots of f(x), where f(x) is a function* computed by the PVALF subroutine that requires four arguments:

x in A and R registers
t in the two words following the transfer
s in the next two words
r in the following two words

The result is in the A and R registers on return from the subroutine.

Assuming that roots are desired in a variety of neighborhoods, a set of up to twenty-five starting points for the Newton-Raphson method is to be input. If less than twenty-five starting guesses are desired, the remaining values are input as +0.0. On finding a starting guess equal to +0.0, the program is to stop, displaying 9. Several sets of twenty-five initial guesses can be input in successive runs through the program.

METHOD OF SOLUTION

The Newton-Raphson method should be employed to iterate for a root based on each starting guess. The i-th root x_i is computed by using:

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}$$

*Note that the exact nature of the function f(x) is not at stake here. The discussion in this section is applicable for any function. All that is needed is some specific subroutines to evaluate it and its derivative in any particular application. In this example, these subroutines have been arbitrarily named PVALF and DRIVF.

where x_o is the starting guess and f'(x) represents the derivative of the function f(x).

The derivative is computed by the DRIV subroutine which requires three arguments:

x in A and R registers

t in the two words following the transfer

r in the next two words

The result is in the A and R registers on return from the subroutine.

If convergence to a root is not obtained after ten iterations, computations are abandoned, and a suitable message is typed. The criterion for convergence is that successive roots agree within 10^{-5}.

When testing the difference between successive roots, the absolute value is required. Thus, if:

$$\left| x_i - x_{i-1} \right| < 10^{-5}$$

convergence has been obtained. For this purpose, an absolute value function, ABSF, must be used.

SOURCE PROGRAM

```
      DIMENSION, XARG (25)
      LIBRARY, PVALF (*,1,3,2,*), DRIVF(*,1,2,*),ABSF(*,*)
  3   READ CONSOLE, T,S,R
  5   READ TAPE, XARG
      DO  10  I : 1,25
      XR = XARG(I)
      IF (XR) 4,9,4
  4   DO  6  INDEX : 1,10
      XROOT = XR - PVALF (XR,T,R,S)/DRIVF(XR,T,R)
      TEMP = XROOT - XR
      TEMP = ABSF (TEMP) - .00001
      IF (TEMP) 8,7,7
  7   XR = XROOT
  6   CONTINUE
      TYPE 19, XARG(I)
 10   CONTINUE
```

```
 9      PAUSE  9
        IF SENSE D, 3, 5
 8      TYPE 17, XARG (I), XROOT
        GO TO 10
17      FORMAT ( 2 TE 20.5)
19      FORMAT ( 1 TE 20.5, 20H     NO CONVERGENCE)
        END
```

Notes on Running This Particular Source Program

When the compiled and assembled object program has been loaded into the RECOMP II, the START button is pressed. The computer will then stop in order to input the values of T, S and R. This is done as follows:

1. Press KEYBOARD FILL button.
2. Press N (number button).
3. Enter inputs (3 mixed numbers). Each number must be input in the following form: $\pm$ XXX.XXX ENTER. The number of digits before and after the decimal point must be at least one. After each number, the ENTER button must be pressed.
4. After entering the third number, namely R, it must be made certain that the paper tape is ready in the photo-electric reader. Then the START3 button is pressed.*

The program then proceeds to read 25 (or less) numbers into the XARG block. The set of numbers must be followed on tape by L00030 and an S code in order to cause the proper conversion process to be performed. Recall also, from the earlier discussion of the approach to this solution of this problem that the last of the XARG block of numbers must be +0.0 if there is to be less than twenty-five numbers in any particular set. This type of consideration is, of course, known to the program designer or programmer and is covered in his operating instructions for his particular problem.

For each of the nonzero numbers of a set of starting points

*Equivalent to the appearance of L00030 followed by Start Code S after data on paper tape.

(i.e., the XARG block) a line of output is typed, showing the root unless convergence was not attained within 10 iterations. After using up a set of XARG points or finding a zero value, the machine stops with the digit nine displayed in the Nixie tubes. At that time sense switch D may be set:

ON, to require new coefficients T, S, and R to be input via console

OFF, to use the same coefficients

In either case, if the START button is pressed, the program will continue, beginning a new series of computations.

If convergence is not obtained, the starting guess is printed, followed by a NO CONVERGENCE comment. If convergence is obtained, the computed root prints to the right of the starting guess.

A listing of the generated object program appears in Appendix A.

Section VI
COMPILER OPERATION

The following steps summarize the solution of a problem by an AFCOR program:

1. Analyze problem and method of solution.
2. Code AFCOR source program
3. Punch program on EAM cards
4. Convert program to paper tape via Systematics converter
5. Compile symbolic object program (AFCOR)
6. Assemble object program, including necessary function subroutines and input-output routines (AFAR)
7. Read assembled object program tape into RECOMP II
8. Ready paper tape input, if any, and set appropriate sense switches and tab settings
9. START

PUNCHING THE SOURCE PROGRAM

The format of punched cards acceptable to the compiler is free form. Punching can begin or end in any card column from 1 to 68. Spacing between characters can be specified in any form that makes the source program easily readable.

Each statement must begin on a new card, but can extend to more than one card. No more than 180 card columns can be used for one statement. The end of a statement is indicated by punching two consecutive dollar signs (3-8-11 punch).

In arranging the source deck for conversion to paper tape via the Systematics converter, the LIBRARY statement, all DIMENSION statements, and the START statement must precede any executable

source statements. The last card must be an END statement.

OPERATION OF THE COMPILER

Sense switch settings are made according to one of the alternatives shown in the following tabulation:

	B	C	D
UP (off)	Produce symbolic object program according to setting of sense switch D	Type each source statement	Punch symbolic object program tape for assembly
DOWN (on)	Do not punch, and do not type the symbolic object program	Do not type source statements	Punch and type symbolic object program

Read compiler into computer; ready source program tape in reader; press START1 button; compilation will proceed.

If no errors have been detected upon completion of the generation of the symbolic object program, the compiler halts at L00020.

Assembling the Object Program

The compiler does not generate:

1. Symbolic input-output routines
2. Function subroutines

Generally, the input-output routines will be preassembled at some locations in high memory. The symbolic references to locations in the I-O routines can be defined by SYN cards to avoid the unnecessary process of reassembling them for each particular object program.

Similarly, preassembled function subroutines can be defined in the symbolic object program by using SYN cards. In either case, however, it is permissible to add the symbolic tapes to the symbolic object program at the time of assembly. The last instruction assembled should be HAL +8 so that the object program tape will

transfer control to L00040, the beginning location of all object programs.

Running the Object Program

The following general procedure summarizes the running of all object programs.

1. Set T-CR switch to appropriate position
2. Set tabular stops on typewriter if required
3. Set sense switches appropriately
4. Load object program tape (stops at L00040)
5. Ready data tape in reader, if required
6. Press START

APPENDIX A

The following is an example of the typed listing from a compilation run. The source program is the one explained in Section V. Note that both the source statements and the corresponding symbolic coding is listed. In addition to the inclusion of the symbols defined by the source language (variable and function names and statement numbers) there are internally generated symbols (beginning with DD, JJ and RR) and certain symbolic addresses associated with the I-O routines. (AF, AA, etc.) If other symbolic coding is to be assembled along with an object program, care must be taken to avoid duplication of the symbols beginning with DD and A; source symbols beginning with JJ and RR must always be avoided.

```
                OCT   7700003760001
                TRA   GA
                NOP
                TRA   GB
                NOP
                TRAL          +2
                TRAL          +11

      DIMENSION, XARG (25)

      LIBRARY, PVALF (*,1,3,2,*), DRIVF (*,1,2,*), ABSF (*,*)

  3   READ CONSOLE, T, S, R

      3         BSS   BSS
                NOP
                TRA   AC
                CLA   T
                STO           +1
                NOP
                TRA   AC
                CLA   S
                STO           +1
                NOP
```

```
            TRA   AC
            CLA   R
            STO           +1

5     READ TAPE, XARG

      5     BSS   BSS
            NOP
            TRA   AA
            CLA   XARG
            STO           +1

      DO 10 I: 1,25

            NOP
            CLA   DDDDR   -2
            STA   10
            CLA   JJDDJ
            STO   I
            TRA   DDDDR   +6
            CLA   DDDDR
            CLA   DDDDR
      DDDDR CLA   I
            ADD   JJDDJ
            STO   I
            SUB   JJDDN
            TZE   DDDDR   +6
            TPL   10      +1

      XR = XARG (I)

            CLA   I
            ALS           22
            ADD   DDDDF
            TRA   DDDDF   +2
      DDDDF CLA   XARG    -1
            CLA           +0
            STA   DDDDC
            NOP
      DDDDC FCA
            FST   XR
```

```
      IF (XR) 4, 9, 4

              CLA   XR
              TZE   9
              TPL   4
              TMI   4

4     DO 6 INDEX:  1, 10

      4       BSS   BSS
              NOP
              CLA   DDDDK    -2
              STA           6
              CLA   JJDDJ
              STO   INDEX
              TRA   DDDDK    +6
              CLA   DDDDK
              CLA   DDDDK
      DDDDK   CLA   INDEX
              ADD   JJDDJ
              STO   INDEX
              SUB   JJDDT
              TZE   DDDDK    +6
              TPL   10       +1

      XROOT = XR - PVALF (XR, T, R, S) / DRIVF (XR, T, R)

              FCA   T
              FST   DDDDZ    +1
              FCA   R
              FST   DDDDZ    +3
              FCA   5
              FST   DDDDZ    +2
              FCA   XR
              TRA   DDDDZ
              BSS   T
      DDDDZ   TRA   PVALF
              BSS   T        3
              FST   RRDDL    +0
              FCA   T
              FST   DDDDW    +1
              FCA   R
```

```
            FST   DDDDW    +2
            FCA   XR
            TRA   DDDDW
            BSS   T
DDDDW       TRA   DRIVF
            BSS   T        2
            FST   RRDDL    +1
            FCA   RRDDL    +0
            FDV   RRDDL    +1
            FCA   XR
            FSB   RRDDL    +0
            FST   XROOT

      TEMP = XROOT - XR

            FCA   XROOT
            FSB   XR
            FST   TEMP

      TEMP = ABSF (TEMP) - .00001

            FCA   TEMP
            TRA   DDDDH
            BSS   T
DDDDH       TRA   ABSF
            FSB   RRDDY
            FST   TEMP

      IF (TEMP) 8, 7, 7

            FCA   TEMP
            TZE   7
            TPL   7
            TMI   8

7     XR = XROOT

      7     BSS   BSS
            FCA   XROOT
            FST   XR
            TRA   6        +1
```

```
      TYPE 19, XARG (I)

              CLA   19
              TRA   AFZ
              CLA   I
              ALS             22
              ADD   DDDDP
              TRA   DDDDP     +2
      DDDDP   CLA   XARG      -1
              CLA             +0
              STA   DDDDP     +4
              TRA   AF
              CLA
              STO             +1

10    CONTINUE

      10      BSS   BSS
              TRA   10        +1

9     PAUSE 9

              DSD   DDDDQ
              HTR   DDDDQ     +2
      DDDDQ   OCT   +6314476000000

      IF SENSE D, 3, 5

              TSD   3
              TRA   5

8     TYPE 17, XARG (I), XROOT

      8       BSS   BSS
              CLA   17
              TRA   AFZ
              CLA   I
              ALS             22
              ADD   DDDDQ
              TRA   DDDDQ     +2
      DDDDO   CLA   XARG      -1
              CLA             +0
```

```
                STA     DDDDQ     +4
                TRA     AF
                CLA
                STO               +1
                NOP
                TRA     AF
                CLA     XROOT
                STO               +1

      GO TO 10

                TRA     10

17    FORMAT (2TE 20.5)

      17        OCT     +6004001200013

19    FORMAT (1TE 20.5, 20 H        NO CONVERGENCE )

      19        OCT     +6002001200012
                OCT     -7620410204116
                OCT     -3140435414756
                OCT     -0253202616037

                END

      XAR       BSS     XARG      +25
      RRDDL     BSS     RRDDL     +2
      T         BSS     T         +1
      S         BSS     S         +1
      R         BSS     R         +1
      XR        BSS     XR        +1
      XROOT     BSS     XROOT     +1
      TEMP      BSS     TEMP      +1
      JJDDJ     DEC               1
      JJDDN     DEC               25
      JJDDT     DEC               30
      RRDDY     FLD               .00001
      GA        TRA
      GB        TRA
```

APPENDIX B

COMPILATION ERRORS

During compilation, any of a number of errors in the source progam may be detected. As each error is encountered, a unique code is displayed in the Nixie tubes and compilation is temporarily stopped. Each code is keyed below to a specific error.

Error Code	Nature of Error
Formula Translator	
00 001	Mixed mode in formula
00 002	Illegal variable name (A through H)
00 003	Undefined function name
00 004	Improper function list
00 005	Illegal delimiter
00 006	Illegal consecutive delimiters
00 007	Improper number of delimiters
00 008	Incomplete expression
00 009	Function or expression to left of equal sign
Do	
01 001	Missing colon
01 002	Missing comma
01 003	Improper terminating statement

Error Code	Nature of Error
Continue	
02 001	No statement number
Assigned Go To	
03 001	Missing right parenthesis
03 002	Improper variable
If	
04 001	Missing left parenthesis
04 002	Missing right parenthesis
04 003	Missing comma
Display	
08 001	Missing comma
08 002	Missing left parenthesis for subscript
Format	
12 001	No statement number
12 002	Missing left parenthesis at beginning of list
12 003	Illegal element
12 004	Illegal conversion: Not E, F, I, H or T

Error Code	Nature of Error
12 005	No character specification before H
12 006	Zero before H
12 007	Repeated tabs (i.e. nT) not followed by I, E or F
12 008	Element does not end with comma, right parenthesis or period
12 009	Illegal element or comma missing between two elements
12 010	Improper period
12 011	Missing right parenthesis at end of list
12 012	Constants larger than 999999
If Sense	
13 001	Missing comma
13 002	Improper transfer specification
Assign	
14 001	Missing left parenthesis
14 002	Missing right parenthesis
14 003	Improper variable or constant
Start	
15 001	Specified start location not 1 or 2
15 002	Missing left parenthesis
15 003	Missing right parenthesis
15 004	Improper transfer specification

Error Code	Nature of Error
	Dimension
18 001	Missing comma
18 002	Improper variable
18 003	Missing left parenthesis
18 004	Improper dimension, singly subscripted variable
18 005	Improper dimension, doubly subscripted variable
18 006	Missing between major and minor dimensions
18 007	Dimension table full
	Library
20 001	More than one LIBRARY statement in program
20 002	Improper list element
20 003	Incomplete list
20 004	List too long
20 005	Statement improperly constructed
20 006	Too many functions defined
	Input-Output, Variables
22 001	No comma between elements
22 002	Improper element
22 003	Missing right parenthesis after subscript
22 004	Undefined subscripted variable

Error Code	Nature of Error
22 005	Missing comma in singly subscripted element
22 006	Improper delimiter after singly subscripted element
22 007	Improper limits on doubly subscripted element
22 008	Ambiguous range on subscripts
22 009	Missing right parenthesis after doubly subscripted element
22 010	Improper major dummy variable
22 011	Missing comma in doubly subscripted element
22 012	Improper minor dummy variable
22 013	Doubly subscripted, singly dimensioned variable
Input-Output, Headings	
23 001	Missing colon
23 002	Identifier larger than 100
23 003	Illegal I/O statement for heading
23 004	Header storage table full
Miscellaneous	
24 001	Binary to baudot conversion attempted on number larger than 999999
25 001	Non-numeric character found in baudot constant
26 000	Illegal subscript in dimensioned variable

Error Code	Nature of Error
	Statement Scanner
30 000	Improper character
30 001	More than 5 characters in name
30 002	Subscripted variable not defined in DIMENSION statement
30 003	More than 13 characters in floating constant
30 004	Two decimal points in floating constant
30 005	More than 6 characters in fixed constant
30 006	More than 4 characters in statement number
30 007	Improper first letter for name
30 008	Function not in arithmetic on LIBRARY statement
30 009	Variable table full
30 010	Two equal signs in statement
30 011	Constant table full
30 012	Illegal statement type

APPENDIX C

INPUT-OUTPUT ROUTINES

Since the compiler must be capable of handling a variety of input and output functions, it would be ineffiecient to generate a specialized symbolic routine for each input or output specification. Instead, the burden of these procedures can be shifted by compiling only the necessary linkages in object programs to a set of generalized subroutines existing apart from the compiler. These routines are termed the input-output package.

The linkages produced by the compiler are transfer instructions (with and without parameter words) to symbolic locations defined in the input-output package. These locations must also be defined when the object program is assembled. There are two ways to accomplish this definition:

Assemble a symbolic version of the input-output package along with the object program

Define those critical locations when assembling the object program by a set of SYN* cards referring to a preassembled version of the package.

The latter method is clearly the more desirable.

Therefore, it is advised that the input-output package be assembled previously at the extreme high end of memory to allow

*See BRC 161-9-Rev., the AFAR Assembly Manual, pages 20 and 21, for a description of the SYN pseudo operation.

object programs as much area as possible (object programs always begin at location 0). Using the assembly listing of the package, a set of SYN cards can be specified by defining the following location symbols: AA, AB, AC, AD, AE, AEZ, AF, AFZ, AG, AGZ, BCH, BEH and BFH. The SYN cards can be converted to paper tape via the Systematics converter. This tape should be used whenever any object program is assembled. Finally, before running any object program, the assembled package should be read into the computer.

If it is desirable to assemble the input-output package from its symbolic form with the object program, the SYN tape must not be used, however, nor should any other assembled version of the package be used when running that object program.